



Edição 147 :: R\$ 14,90

 DEVMEDIA

NoSQL: Modelando os dados de suas aplicações  
Prepare os dados de suas soluções Java  
para o Apache Cassandra

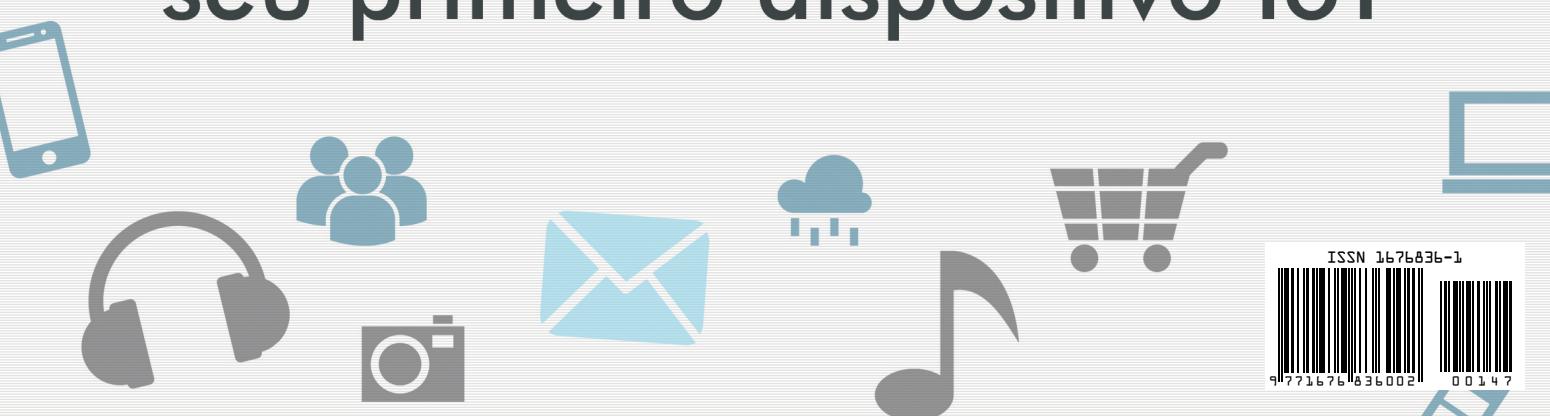
Construindo microserviços seguros  
Aprenda como mitigar  
os riscos através da autenticação

Injeção de dependências no Android  
Reduza o acoplamento e facilite  
a manutenção do código

# INTERNET das COISAS

A decorative icon set for IoT concepts, including a blue speech bubble, a grey paperclip, a grey speech bubble, and a teal speech bubble.

## Saiba o que é e construa seu primeiro dispositivo IoT

A decorative icon set showing various IoT applications: a smartphone, headphones, a person icon, a camera, an envelope, a cloud with rain, a shopping cart, a musical note, and a computer monitor.

Testes unitários, de integração  
e de aceitação  
Como implementá-los com as  
ferramentas JUnit, Hamcrest e Concordion

DevOps: conheça os papéis  
do desenvolvedor  
O que fazer para entregar software  
com qualidade e eficiência



# MVP

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

APlique esse investimento  
na sua carreira...

E mostre ao mercado  
quanto você vale!

CONFIRA TODO O MATERIAL  
QUE VOCÊ TERÁ ACESSO:

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

POR APENAS  
**R\$ 69,90\*** mensais

\*Tempo mínimo de assinatura: 12 meses.



PRA QUEM QUER EXIGIR  
MAIS DO MERCADO!

 **DEVMEDIA**



Edição 147 • 2016 • ISSN 1676-8361

## EXPEDIENTE

### Editor

Eduardo Spínola ([eduspinola@gmail.com](mailto:eduspinola@gmail.com))

**Consultor Técnico** Diogo Souza ([diogosouzac@gmail.com](mailto: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](http://www.devmedia.com.br/central), ou se preferir entre em contato conosco através do telefone 21 3382-5038.

### Publicidade

[publicidade@devmedia.com.br](mailto: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](http://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](http://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 do tipo Mentoring

## 20 – Spring Boot: Como desenvolver microserviços seguros

[ Diego Ernesto Rosa Pessoa ]

Artigo no estilo Solução Completa, Conteúdo sobre Novidades

## 29 – Primeiros passos no mundo da Internet das Coisas – Parte 1

[ Rômero Ricardo de Sousa Pereira ]

Artigo no estilo Solução Completa, Conteúdo sobre Boas Práticas

## 42 – DevOps: Conheça os papéis do desenvolvedor

[ Pedro E. Cunha Brigatto ]

Conteúdo sobre Boas Práticas, Conteúdo sobre Eng. de Software

## 57 – Testes unitários, de integração e de aceitação na prática

[ Daniel Medeiros de Assis ]

Conteúdo sobre Boas Práticas

## 68 – Injeção de dependências em aplicações Android

[ Sergio Eduardo Dantas de Oliveira ]



### 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](http://www.devmedia.com.br/javamagazine/feedback)

# REVISTAS DIGITAIS



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

**São mais de 4.000 artigos publicados!**

Uma verdadeira biblioteca online perfeita para seus estudos!



Para mais informações :

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

 **DEVMEDIA**

# Apache Cassandra: Modelando os dados de sua aplicação NoSQL

Aprenda nesse artigo como organizar os dados de suas aplicações Java para persistência no Apache Cassandra

**A**tualmente, grande parte dos sistemas já opera através da Internet. Como consequência disso, eleva-se a quantidade potencial de usuários e passa-se a expor as limitações das tecnologias tradicionais. Em muitos casos essa exposição se deu pelo fato dos sistemas terem apresentado um crescimento bastante elevado do número de acessos, o que culminou em um aumento exponencial do volume de dados a tal ponto que os bancos relacionais passaram a ter dificuldades em processar as requisições com um tempo de resposta satisfatório. A solução, então, seria escalar o banco de dados verticalmente, adicionando mais recursos de hardware numa mesma máquina, de forma a garantir um desempenho aceitável para o sistema. Entretanto, os custos com isso podem se tornar proibitivos, assim como em algum momento o limite dessa escalabilidade pode ser alcançado. Diante disso, os bancos de dados relacionais se tornaram um gargalo na arquitetura desses sistemas.

Essas limitações fizeram com que os pesquisadores buscassem alternativas para melhorar o desempenho e, a partir daí, criaram opções de replicação de dados dos bancos relacionais em vários nós (mestre-escravo, mestre-mestre) e os particionamentos vertical e horizontal. Dessa forma, foi criada a possibilidade de se escalar horizontalmente um banco de dados relacional. Entretanto, os pesquisadores notaram que para grandes volumes de dados, o custo de se manter uma estrutura de hardware para escalar horizontalmente de forma satisfatória era proibitivo. Esse elevado custo se dava por conta das características ACID (Atomicidade,

## Fique por dentro

O Apache Cassandra é um dos melhores bancos de dados de arquitetura distribuída, principalmente para projetos de Big Data, nos quais se espera alta disponibilidade, escalabilidade linear e capacidade de operar em múltiplos centros de processamento simultaneamente, como um único repositório de dados. Devido a essas características, esta solução NoSQL está sendo adotada cada vez mais em sistemas de escala global, que requerem uma distribuição geográfica em várias localidades. Entre esses sistemas, estão os de processamento de transações online de larga escala, como ocorre com a loja virtual da Amazon.

Os tradicionais bancos de dados relacionais não lidam tão bem com os requisitos de sistemas dessa magnitude. Apesar disso, no exterior, assim como no Brasil, o Cassandra não é tão popular quanto os bancos relacionais, mas tem sido reconhecido como um banco promissor, capaz de ocupar um espaço que as soluções padrão não atendem com excelência. Sabendo disso, vamos explorar esse assunto neste artigo, abordando primeiramente um ponto fundamental: a modelagem de dados para o Cassandra.

Consistência, Integridade e Disponibilidade) do banco de dados. Para garantir essas propriedades, o banco terminava por fazer pesquisas em todos os nós do cluster de dados a fim de realizar as operações de JOIN, precisava fazer leituras (muitas vezes em vários nós) antes de escrever ou atualizar os dados, entre outros detalhes. Todo esse comportamento levou a um custo muito alto para se realizar consultas, aumentando o tempo das mesmas de tal forma que se tornaram inviáveis. Nesse momento, negócios que necessitavam de respostas rápidas, principalmente os de

operações críticas, começaram a sofrer com essas dificuldades e tiveram que buscar alternativas.

A opção que se encontrou foi baseada no teorema de CAP, o qual conceitua que é impossível, para um sistema distribuído, garantir as características de consistência (só existe um único valor em todo o cluster para um mesmo registro), disponibilidade (é possível executar operações com sucesso a qualquer momento/tempo razoável) e tolerância a falhas (sistema continua a operar mesmo se um nó tiver falha de rede). Portanto, só é possível construir sistemas distribuídos que atendam a no máximo duas das características do teorema de CAP, sendo necessário, portanto, flexibilizar as regras de armazenamento de dados em troca de maior escalabilidade e performance.

Esse conceito é válido até mesmo para os bancos de dados relacionais, quando se opta por escalá-los horizontalmente, pois se houver falha de rede em algum nó, algumas consultas podem não ser executadas. Obviamente, com o aumento do volume de dados, se faz necessário escalar horizontalmente, a fim de aumentar a capacidade de processamento e armazenamento, diminuir custos (nós simples em vez de máquinas poderosas) e aumentar a disponibilidade. Essa necessidade fez com que surgessem os bancos de dados NoSQL, como o Cassandra, o qual faz uso do conceito de consistência eventual, garantindo apenas a disponibilidade e tolerância a falhas.

A consistência eventual significa que se nenhuma atualização for feita a um registro a partir de determinado momento, eventualmente, ou seja, quando todas as atualizações anteriores tiverem sido replicadas em todos os nós do cluster, todos os acessos àquele registro retornarão o valor mais atualizado. O problema é que antes que o dado mais atualizado esteja disponível em todos os nós do cluster de dados, não necessariamente o valor retornado a uma consulta será o mais atualizado, pois a replicação dos dados demora um certo tempo para ocorrer, criando uma janela de inconsistência. Por exemplo, se um cluster Cassandra possui três nós e houver uma atualização em um registro trocando a letra "A" por "B", enquanto o valor B não for atualizado em todos os nós, uma consulta pode consultar um nó em que o valor ainda seja "A" (a escolha do nó é aleatória), pois a replicação não terminou, ocasionando uma inconsistência.

O Cassandra possui formas de contornar essa limitação ao possibilitar a configuração do nível de consistência. Entretanto, não chega a garantir a consistência, como nos sistemas relacionais. Dessa forma, esta solução NoSQL é a melhor opção para situações em que se tenha um grande volume de dados, necessite de alta disponibilidade, tolerância a falhas e que não seja necessário trabalhar sempre com o dado mais recente. Um bom exemplo de aplicação que pode tirar proveito dessa característica é o carrinho de compras da Amazon. O modelo de negócio da empresa assume que é melhor pedir desculpa ao usuário por um eventual erro, no caso de qualquer inconsistência (nos dados, compra efetuada de forma errada, falha na compra, etc.), do que arcar com os custos de garantir a consistência com um banco de dados relacional. Quando ocorre esse tipo de situação, a empresa pede desculpas e oferece

uma série de vantagens ao cliente, como bons descontos em produtos ou créditos para serem gastos na loja. Dessa forma, além de incentivar uma nova compra, a empresa tem o custo de operação diminuído significativamente ao adotar essa estratégia. O "custo" dos incentivos aos clientes que passaram pelo problema é diluído pelo maior volume de compras que o sistema pode lidar.

Além disso, existem vários casos de sucesso da utilização do Cassandra, em vários ramos de negócio, como, por exemplo: catálogo de produtos, redes sociais, detecção de fraudes e aplicações analíticas no geral. Devido a isso, tem sido adotado por milhares de empresas de diferentes áreas, como a Amazon (comércio eletrônico), eBay (comércio eletrônico), Netflix (serviço de assinatura de filmes e séries de TV), Facebook (rede social), CERN (centro de pesquisa nuclear), FedEx (logística), Globo.com (portal de notícias), Microsoft (software), Credit Suisse (banco de investimento) e até mesmo a NASA (agência espacial estadunidense).

Como era de se esperar, o Cassandra não foi escolhido por acaso. Ele é altamente escalável, possui uma arquitetura P2P tolerante a falhas, um modelo de dados versátil e flexível e uma linguagem de consulta com baixa curva de aprendizado. Todas essas características fazem com que o Cassandra seja o repositório perfeito para aplicações que precisam estar sempre disponíveis e que operam com grandes volumes de escrita e leitura de dados. Com esta solução NoSQL é possível atender a milhões de transações por segundo, em grandes volumes de dados, fazendo uso de milhares de servidores.

No entanto, um dos grandes desafios que novos projetos encontram ao adotar o Apache Cassandra é que a modelagem de dados é bem diferente. As abordagens tradicionais se baseiam em bancos relacionais e já possuem uma metodologia bem estabelecida, fruto de décadas de pesquisas. Por sua vez, por ter uma abordagem diferente e recente, existem poucas metodologias para a modelagem de dados não-relacionais. A primeira tentativa e a mais utilizada até o momento, foi criada por três pesquisadores da Wayne State University, entre os quais se destaca Artem Chebotko, arquiteto de soluções da DataStax, uma empresa de software que fornece uma versão comercial do Apache Cassandra. Essa abordagem que será demonstrada ao longo deste artigo.

O processo de modelagem relacional é bastante focado nos dados, pois procura entender e organizar os dados de forma relacionada, de tal forma a minimizar a redundância e duplicação dos mesmos. Além disso, as consultas feitas ao banco de dados, a princípio, não interferem no processo de modelagem, ou seja, não se modela pensando nas consultas que a aplicação deseja fazer. Durante esse processo, a análise e otimização de consultas é uma atividade muitas vezes não executada, uma vez que se tem uma linguagem poderosa como a SQL que ajuda na obtenção dos dados. Tudo isso resulta em um projeto que ajuda a evitar a duplicação, impondo regras bastante restritas sobre os dados, mas que não é otimizado para realizar consultas que exigem processamento em grandes de volumes de dados e curto tempo de resposta.

Em contrapartida, a forma de estruturação e armazenamento de dados do Cassandra foi projetada com o intuito de prover uma

# Apache Cassandra: Modelando os dados de sua aplicação NoSQL

performance superior para as consultas que a aplicação precisa executar. Dito isso, a modelagem de dados para o Cassandra se inicia com as consultas, pois o que se mais objetiva é a agilidade na realização das operações em detrimento da consistência dos dados. Ao contrário do que prega a cartilha da modelagem relacional, para o Cassandra a regra é encadear os dados pertencentes a uma mesma consulta e desnormalizar as entidades relacionais, com o objetivo de que consultas complexas possam ser executadas acessando uma única tabela. Por conta disso, é bastante comum que o mesmo dado seja armazenado em múltiplas tabelas, a fim de suportar várias formas de consultas, o que resulta na duplicação de dados.

Portanto, é preciso entender como modelar a estrutura de armazenamento dos dados da aplicação no Cassandra, levando em consideração as consultas que serão executadas. Para isso, existe um processo de modelagem padrão bem definido, o qual será demonstrado ao longo do artigo, partindo das etapas de identificação do fluxo da aplicação, de forma simultânea com a modelagem do modelo conceitual, culminando no modelo físico, o qual será utilizado para criar o esquema de banco de dados desejado.

## O Modelo de Dados do Cassandra

Um esquema de banco de dados no Cassandra é denominado *keyspace*. Este nada mais é do que uma estrutura de dados na qual todos os outros objetos do banco

de dados residem, além de uma série de atributos de configuração que não são o foco deste artigo. Conforme a **Figura 1**, dentro do *keyspace* são definidas uma série de tabelas para armazenar os dados de uma aplicação, e mais adentro, pode-se perceber que cada tabela possui linhas e colunas. A essa estrutura dá-se o nome de modelo de dados, o qual representa os mecanismos internos que o Cassandra faz uso, no intuito de armazenar os dados.

### Tabelas (Famílias de Colunas)

Uma tabela, ou família de colunas no Cassandra, é a estrutura de dados onde são armazenados os dados da aplicação. Neste modelo, cada tabela é formada por um conjunto de linhas, e cada linha, por um conjunto de colunas. Ainda no Cassandra, cada tabela pode ou não ser subdividida em vários subconjuntos de linhas, chamados de partições, os quais podem ter apenas uma única linha ou mais. Além disso, podem existir três tipos de chaves ou identificadores em cada linha: chaves de partição (*partition keys*), chaves de ordenação (*clustering keys*) e chaves primárias (*primary keys*). Entretanto, as chaves de partição e primária são obrigatórias (sempre existem, ainda que as duas sejam a mesma), enquanto as de ordenação são opcionais.

A chave de partição é um conjunto de uma (chave simples) ou mais colunas (chave composta) de uma linha que identifica de qual partição (subconjunto) da tabela a linha faz parte. No entanto, não basta

apenas identificar a partição à qual a linha pertence, pois, o Cassandra não saberia em que ordem as linhas devem ser armazenadas em cada partição. Desse modo, esse é o propósito da chave de ordenação, a qual também é definida por um conjunto de uma ou mais colunas cujos valores servem como critério de ordenação. Por sua vez, a junção entre as chaves de partição e de ordenação identificam unicamente uma linha em uma partição, sendo assim chamada de chave primária. Devido a essa característica, existem partições que só podem ter uma linha, porque enquanto a chave da partição é obrigatória, a de ordenação é opcional. Assim, na ausência da chave de ordenação, o Cassandra assume que a chave de partição também é a chave primária da linha, e nestes casos, como não haveria como identificar mais de uma linha, a partição poderá ter no máximo uma linha.

A definição da estrutura da tabela, na linguagem utilizada para operar o Cassandra, a CQL (*Cassandra Query Language*), especifica um conjunto de colunas e uma chave primária, ou seja, para se criar uma tabela no Cassandra, é necessário apenas informar o nome e o tipo de dado (primitivos: *int*, *text* ou compostos/coleções: *set*, *list* ou *map*) de cada coluna e quais são as colunas que compõem a chave primária. Dessa forma, todas as linhas da tabela terão a mesma estrutura, ainda que em partições diferentes. Além disso, também é possível definir uma coluna com o tipo de dados *counter*, o qual é um tipo especial, usado para manter um contador distribuído, a fim de realizar operações de agregação em um cluster de máquinas como, por exemplo, contar o número de visitas realizadas a um site. No entanto, se houver uma coluna do tipo *counter*, todas as outras que não são desse tipo devem fazer parte da chave primária, pois só assim o Cassandra consegue realizar as operações de agregação. Ademais, similarmente à coluna de tipo *counter*, existem outros tipos de coluna, como a opção estática (*static*), a qual apresenta o mesmo valor em todas as linhas de uma partição. Obviamente, esse tipo de coluna só faz sentido numa tabela com partições com múltiplas linhas.

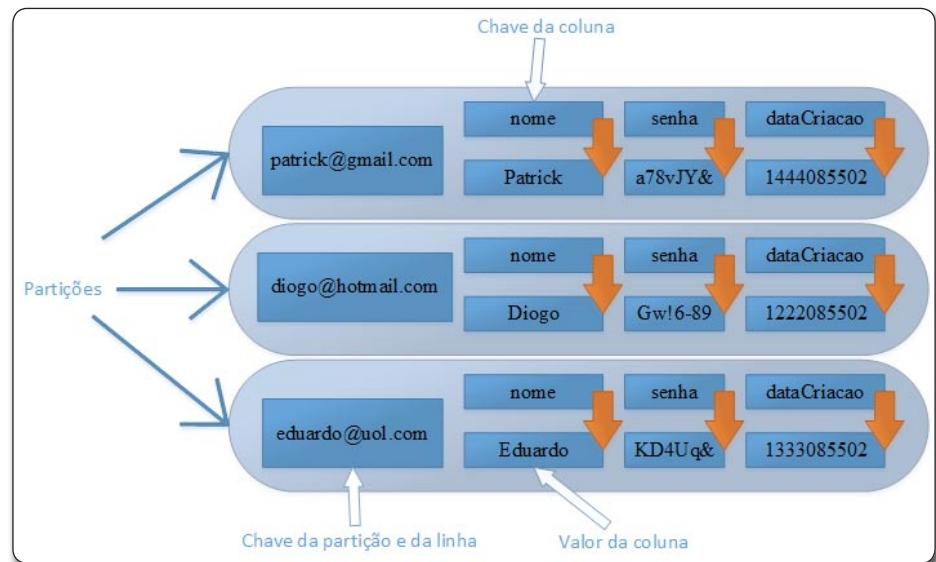
Figura 1. Modelo de dados do Cassandra

Para demonstrar esses conceitos, imagine a tabela Usuário conforme a **Tabela 1**.

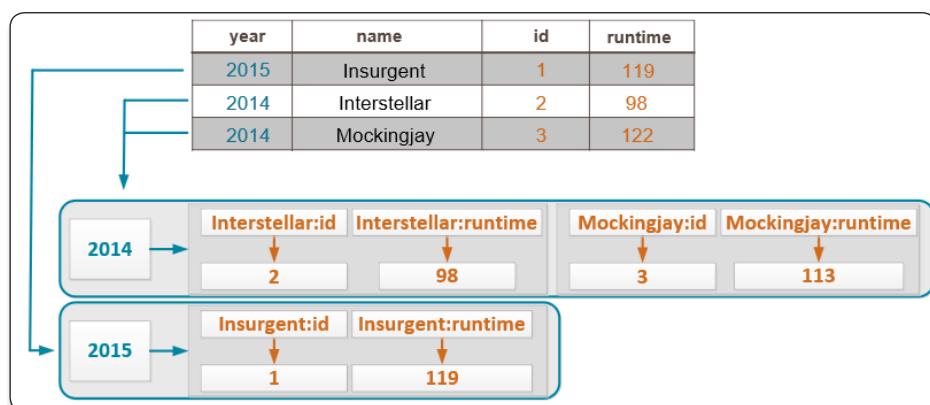
Sendo *email* a chave primária, ela também é a chave da partição, pois como mencionado anteriormente, na ausência de uma chave de ordenação, a chave primária e a chave da partição serão iguais. Logo, cada linha que representa um usuário nessa tabela está alocaada em uma partição (partição de única linha).

A **Figura 2** demonstra, de forma gráfica, como fica a estrutura de partções de uma única linha, tendo como base a **Tabela 1**. Com o objetivo de recuperar os dados com mais agilidade o Cassandra distribui cada partição de uma tabela em um nó do cluster de dados. Assim, supondo que tenhamos um cluster composto por três máquinas, cada máquina vai conter uma partição, ou seja, uma máquina armazenará a partição cuja chave é *patrick@gmail.com*, outra com *diogo@hotmail.com* e a terceira com *eduardo@uol.com*. Portanto, quando for realizada uma pesquisa pelo usuário cujo e-mail é *patrick@gmail.com*, o Cassandra irá consultar apenas uma máquina, a qual possui aquele dado específico, consultando pela chave da partição/linha. Por outro lado, quando existem partções com várias linhas (vide **Figura 3**), a abordagem é um pouco diferente.

Uma tabela com partções com múltiplas linhas tem a chave primária composta pela coluna que representa a chave da partição e pela coluna que representa a chave de ordenação. Essa figura, em conjunto com a **Tabela 2**, ilustram exatamente esse conceito, onde a chave da partição é a coluna *year* e a chave de ordenação/identificador da linha é a coluna *name*. Dessa forma, se uma pesquisa filtrar por *year = 2014*, consultará um único nó do cluster, o qual contém a partição cujo identificador é 2014, retornando duas linhas. No entanto, é preciso ter cuidado com o projeto de partição de múltiplas linhas, pois cada partição no Cassandra deve estar inteiramente armazenada em um único disco rígido, uma vez que não é possível dividir uma mesma partição em vários nós do cluster, a fim de evitar queda na performance das consultas.



**Figura 2.** Tabela com partções de única linha, conforme estrutura da **Tabela 1**



**Figura 3.** Tabela de partções de múltiplas linhas

email (Chave)	nome	senha	dataCriacao
patrick@gmail.com	Patrick	a78vJY&	1444085502
diogo@hotmail.com	Diogo	Gw!6-89	1222085502
eduardo@uol.com	Eduardo	KD4Uq&	1333085502

**Tabela 1.** Exemplo de tabela Usuário

year (Chave partição)	name (Chave ordenação)	id	runtime
2014	Interestellar	2	98
2014	Mockingjay	3	113

**Tabela 2.** Demonstração das chaves de partição e ordenação da tabela da **Figura 3**

#### Nota

Para um entendimento mais aprofundado acerca da estrutura interna do Cassandra, a forma de utilização dos tipos de dados, entre outras informações que estão fora do escopo deste artigo, recomenda-se a leitura da documentação oficial da DATASTAX, bem

como a leitura do artigo de Otávio Santana no site da DevMedia. Os endereços para acessá-los estão disponíveis na seção **Links** e estão intitulados "DataStax - Documentação do Cassandra" e "DevMedia - Artigo sobre Cassandra em Java", respectivamente.

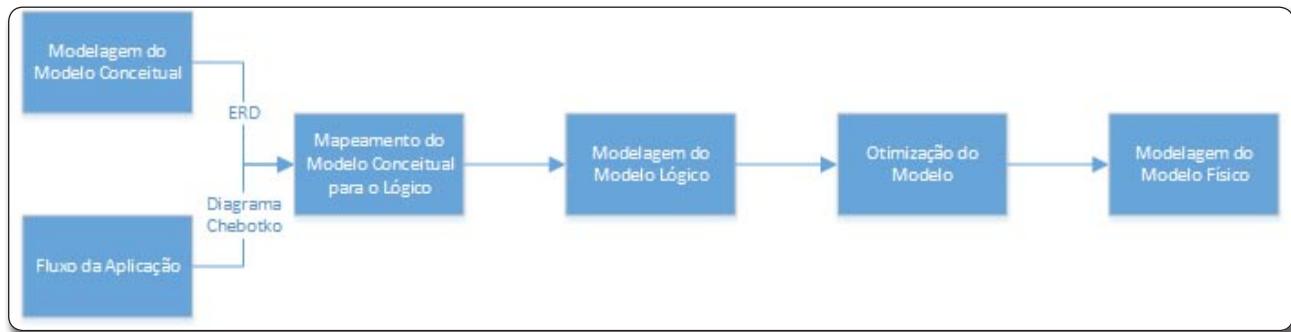


Figura 4. Processo de modelagem de dados para o banco de dados Cassandra

## Fluxo de Modelagem de Dados

Após uma breve introdução ao modelo de dados do Cassandra, a partir deste tópico será apresentado o processo de modelagem seguindo uma metodologia escolhida pelo autor. Entretanto, para iniciar a modelagem é preciso conhecer os requisitos de negócio que devem ser traduzidos em funcionalidades da aplicação. Para tanto, observe o cenário a seguir.

“Uma empresa deseja desenvolver uma aplicação para exibição de vídeos de forma gratuita, por meio da internet. A aplicação será disponibilizada globalmente na Internet fazendo uso de centros de processamento distribuídos em três países: China, Estados Unidos e Brasil. A expectativa é que a aplicação que irá funcionar nessa infraestrutura tenha milhões de usuários por todo o mundo. Cada usuário deve ser capaz de se cadastrar informando alguns dados (e-mail, senha, nome), logar na aplicação usando e-mail e senha, pesquisar vídeos, enviar vídeos, fazer comentários em vídeos, bem como visualizar vídeos de forma anônima ou logado. O sistema deve armazenar a data de cadastro do usuário no ato de seu cadastro para relatórios posteriores. Os usuários podem adicionar quantos vídeos quiserem e devido à quantidade de dados esperada, a empresa está preocupada em como lidar com esse volume em tamanha escala, como também com o desempenho da aplicação. Entretanto, ela afirmou que a eventual perda de alguns dados não é crítica para o negócio, tendo em vista que é uma plataforma de entretenimento.”

Este breve enunciado expõe, de forma sucinta, os requisitos necessários para a aplicação solicitada pelo cliente. Ao analisá-los, nota-se claramente que eles podem ser atendidos fazendo uso do Cassandra, pois temos:

- Necessidade de bancos de dados em múltiplos centros de processamento;
- Grande volume de dados e transações (milhões de usuários, número ilimitado de vídeos);
- Perda de informação é tolerável;
- Preocupação com escalabilidade e alto desempenho.

Subjetivamente, pode-se entender também que não é necessária a recuperação dos dados mais recentes em todas as ocasiões como, por exemplo, na listagem de comentários de um vídeo e na listagem de vídeos, pois nessas ocasiões o que importa é mostrar

algum resultado para o usuário, ainda que não seja o melhor possível. Dito isso, iniciemos o processo de modelagem de dados conforme os passos definidos no diagrama da **Figura 4**.

Nesta imagem estão demonstrados os passos ou tarefas que devem ser seguidos com o intuito de criar um modelo de banco de dados para uma aplicação que deseja fazer uso do Cassandra, conforme uma metodologia escolhida pelo autor. Como são muitos passos no processo, por questões didáticas, optou-se por dividi-los em três grandes passos, a saber: Modelagem Conceitual/Fluxo da Aplicação, Modelagem Lógica e Modelagem Física.

### Passo 1: Modelagem conceitual e fluxo da aplicação

O princípio de toda modelagem de dados é conceituar ideias ou contextos de negócio, transformando-os em uma linguagem ou modelo (modo de descrever algo, como um diagrama, por exemplo) que possa ser entendido da forma mais clara possível, sem espaço para ambiguidades. Por isso essa atividade é importante, pois em geral, várias pessoas de áreas diferentes precisam entender, ainda que de forma superficial, como os dados estão estruturados na aplicação.

Em paralelo à modelagem conceitual, quando se modela os dados para bancos de dados não relacionais, como é o caso do Cassandra, existe outra tarefa igualmente importante a ser executada, a análise do fluxo da aplicação. Essa tarefa se refere a identificar as consultas que devem ser realizadas pela aplicação, de acordo com os requisitos de negócio, sintetizando-as em um diagrama.

Como o primeiro passo do processo é executar as atividades supracitadas, esta seção as aborda de forma teórica, com o intuito de auxiliar na compreensão das mesmas.

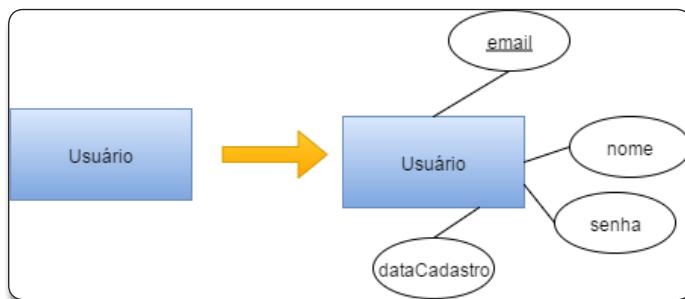
#### Modelo Conceitual

Para fazer a modelagem conceitual será utilizada a técnica de modelagem entidade-relacionamento (ERM – Entity-Relationship Model), a qual visa construir um modelo que descreve dados, informações, domínios de negócio e até mesmo requisitos da aplicação de forma abstrata. Os principais componentes desse modelo são as entidades e os relacionamentos que existem entre elas.

O intuito desse modelo é dar uma visão geral de como as entidades de uma aplicação se relacionam e que dados elas possuem, se abstendo de especificar detalhes técnicos de implementação. Ao

conhecer o escopo dos dados da aplicação através desse modelo é possível transformá-lo em outros mais detalhados, que serão a base para a implementação do repositório de dados. Por isso, para este artigo é interessante que se tenha algum conhecimento em modelagem entidade-relacionamento, a fim de melhor compreender os diagramas que aqui serão apresentados.

O primeiro conceito que se pode abstrair dos requisitos do software é a ideia de usuário da aplicação, ou simplesmente usuário. Esse conceito se obtém da percepção de que um dos elementos-chave para o funcionamento da aplicação é a presença do usuário, o qual é uma pessoa que faz uso do sistema que se quer desenvolver. Então, a primeira entidade a ser criada é a entidade Usuário (vide **Figura 5**).



**Figura 5.** Modelagem da entidade Usuário com seus respectivos atributos

No enunciado com os requisitos estão explícitos os dados pertinentes ao usuário que devem ser considerados. Esses dados são *email*, *nome*, *senha* e *dataCadastro*. Como no modelo entidade-relacionamento toda entidade deve ter um atributo que seja um identificador, foi escolhido o *email*, o qual está sublinhado na imagem, destacando-o como tal.

A segunda entidade que deve ser criada é a que representa o vídeo. Logo, denominaremos essa entidade como *Vídeo*. Esta entidade possui uma série de atributos, como data de publicação, título, duração e sequencial (identificador numérico que faz parte de uma sequência, o qual incrementa 1 ao último valor inserido). Conforme os requisitos de negócio, a modelagem deve ser de feita tal forma que cada vídeo no modelo de dados só pode ser

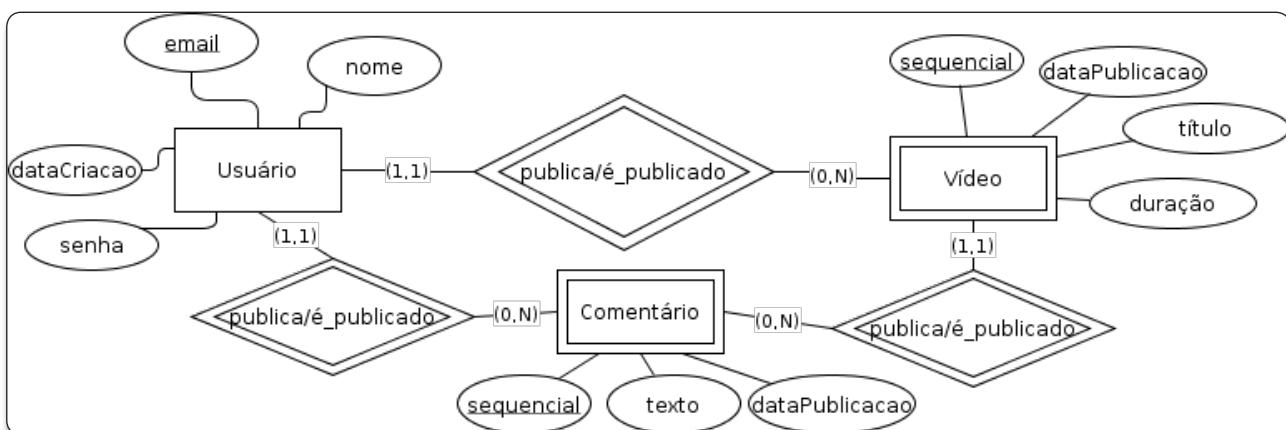
publicado por um único usuário e um usuário pode publicar vários vídeos. Dito isso, seguindo os conceitos de ERM, a relação entre as entidades *Usuário* e *Vídeo* deve ser modelada através de um relacionamento identificador (vide **BOX 1**) entre vídeo e usuário, no qual o vídeo é identificado pela chave primária do usuário (*email*) e um sequencial, pois um vídeo não existe sem um usuário. Portanto, cada vídeo será identificado unicamente por duas chaves: a chave *email* da entidade forte e o atributo *sequencial* da própria entidade.

O enunciado dos requisitos também informa que deve existir uma funcionalidade pela qual o usuário pode fazer comentários em qualquer vídeo. Por consequência, existe uma relação entre um vídeo e seu respectivo comentário. Logo, o modelo de dados deve possuir uma terceira entidade denominada Comentário. Os comentários, conforme a **Figura 6**, possuem atributos como texto (conteúdo digitado pelo usuário), data de publicação e sequencial (identificador numérico que faz parte de uma sequência). Entretanto, também é preciso levar em consideração que cada usuário logado pode publicar um ou mais comentários sobre determinado vídeo e, obviamente, cada vídeo pode receber vários comentários de diversos usuários diferentes. Ademais, como não existe comentário sem usuário e sem vídeo, Comentário é uma entidade fraca, com relacionamento identificador com *Usuário* e com *Vídeo*. Desta forma, cada comentário é unicamente identificado pelo usuário que o publicou e pelo vídeo ao qual aquele comentário se refere e mais um *sequencial*, para possibilitar que haja mais de um comentário por usuário e por vídeo.

#### BOX 1. Relacionamento identificador

Um relacionamento identificador ocorre quando uma entidade não faz sentido se separada de outra. Nesses casos temos uma “entidade fraca”, por exemplo: Empresa possui Filial. Filial é uma entidade fraca de empresa, possuindo, portanto, um relacionamento identificador. Assim, a chave primária de Filial provavelmente será composta pela chave primária de Empresa (a qual também será uma chave estrangeira) e um número que a identifica, como no código a seguir:

- Empresa(idEmpresa, Nome, ...);
- Filial(idEmpresa, idFilial, Endereço, ...).



**Figura 6.** Modelo de dados conceitual da aplicação em diagrama ERM

# Apache Cassandra: Modelando os dados de sua aplicação NoSQL

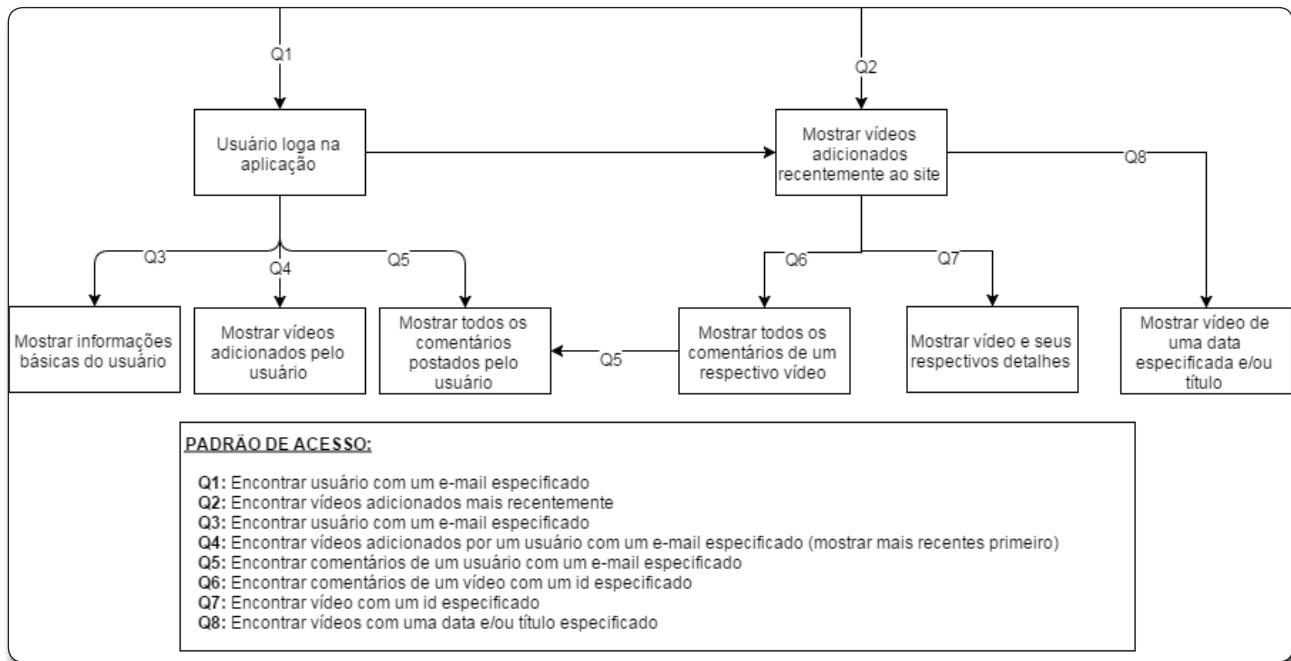


Figura 7. Fluxo da aplicação conforme os padrões de acesso

## Fluxo da aplicação

Com o modelo conceitual preparado é preciso definir o fluxo de funcionamento da aplicação, de forma a identificar os pontos de otimização do modelo a fim de atender às requisições feitas à base de dados. No fluxo da aplicação se especifica quais as tarefas/operações a aplicação executa (interações que os usuários realizam), como também as respectivas dependências (interações anteriores obrigatórias para se realizar a atual) e em que ordem. Além disso, são definidos os padrões de acesso (consultas que o sistema realiza para conseguir executar determinada tarefa) no fluxo da aplicação, os quais ajudam a determinar como os dados são acessados e a ordem das consultas realizadas. Logo, cada tarefa ou interação do usuário identificada no fluxo deve possuir ao menos uma consulta a banco de dados associada.

Um bom exemplo da relação entre uma tarefa e um padrão de acesso (consulta) é a ação/tarefa de logar na aplicação e a consulta à tabela *Usuário* no banco de dados, informando o e-mail do usuário que deseja logar, a fim de validar a existência do mesmo no sistema. Toda vez que um usuário tentar logar na aplicação, sempre haverá uma consulta ao banco de dados pelo e-mail fornecido, gerando um vínculo entre a atividade executada pelo usuário (tarefa) e a respectiva consulta. Assim, se constrói um diagrama com correlações entre tarefas e consultas.

Portanto, para identificar o fluxo da aplicação é necessário observar os requisitos desejados (no cenário apresentado no início do tópico “Fluxo de Modelagem de Dados”) a fim de deduzir algumas tarefas que a aplicação proposta necessita, e depois disso, identificar as consultas que podem ser associadas a cada uma delas. Dito isso, para este artigo serão consideradas as seguintes tarefas:

- Usuário logar na aplicação (site);
- Mostrar informações básicas do usuário;

- Mostrar os vídeos adicionados recentemente no site;
- Visualizar vídeo no site;
- Listar comentários de um usuário;
- Listar comentários de um vídeo;
- Pesquisar vídeo por data ou título;
- Listar vídeos de um usuário.

É importante salientar que as tarefas aqui mencionadas são aquelas que exigem consultas (padrões de acesso) ao banco de dados. Obviamente, existem outras inúmeras possibilidades de tarefas a serem executadas como, por exemplo, enviar um vídeo para a aplicação. No entanto, tarefas que são de escrita serão desconsideradas para o mapeamento do fluxo, pois para projetar o banco segundo a metodologia abordada, devemos considerar apenas as consultas, a fim de otimizar o planejamento do banco para realizar a leitura de dados da forma mais rápida possível.

Dito isso, ao modelar o fluxo da aplicação, o diagrama resultante é apresentado na Figura 7.

Observando essa imagem, nota-se que há uma correlação entre as tarefas (retângulos) e os padrões de acesso ou consultas (linhas). Percebe-se também que há uma ordem definida pela numeração de cada padrão de acesso (Q1 a Q8) e pelo fluxo das setas. A letra “Q”, disposta nas setas e no início de cada linha de padrão de acesso, se refere a uma query, ou consulta, ao banco de dados. Outro ponto importante a se notar no diagrama é que existem dois retângulos no mesmo nível, no topo do diagrama, os quais não possuem nenhuma seta proveniente de outro retângulo. Eles estão modelados dessa forma porque não guardam relação de dependência com uma tarefa anterior, visto que o usuário tanto pode logar na aplicação como pode visualizar os vídeos adicionados recentemente de forma anônima.

## Passo 2: Modelagem Lógica

A criação do modelo lógico segue uma abordagem top-down, ou seja, se modela tendo em mente os conceitos mais abstratos até os mais concretos e detalhados. Dessa forma, em primeiro lugar se divide a aplicação em partes, cada uma representando um tipo de conceito, para posteriormente detalhar cada uma, analisando um problema por vez. Esse tipo de abordagem pode ser definido de forma algorítmica ou programática, ou seja, é possível fazer uso de uma linguagem de programação e devido a isso, existem ferramentas que automatizam esse processo. Para o Cassandra uma das ferramentas é a KDM, a qual, inclusive, pode ser acessada de forma gratuita, exclusivamente pelo site. Entretanto, o escopo deste artigo não inclui a utilização de ferramentas de automação, pois o foco é na aprendizagem do processo.

Para realizar o mapeamento do modelo conceitual para o lógico, faz-se uso do modelo conceitual e do fluxo da aplicação, definidos anteriormente. Ademais, também existe uma série de regras de mapeamento e padrões que podem ser utilizados para auxiliar na criação do esquema lógico e assegurar que o mesmo esteja correto e funcione conforme esperado.

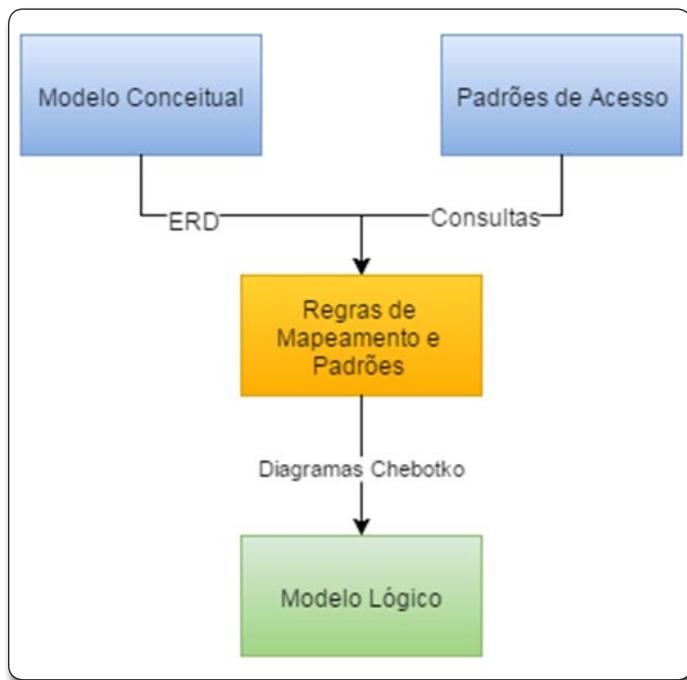


Figura 8. Processo de criação do modelo lógico

Como pode ser observado na **Figura 8**, o modelo conceitual é representado pelo diagrama entidade-relacionamento, também chamado de ERD, demonstrado na figura como um artefato de saída do retângulo “Modelo Conceitual” e os padrões de acesso são as consultas identificadas no fluxo da aplicação, estando representada como “Consultas”, numa linha de saída do retângulo “Padrões de Acesso”. Portanto, as consultas e o diagrama ERD servem de insumo para a aplicação de regras e padrões de

mapeamento (retângulo amarelo), a qual é uma tarefa intermediária para a tarefa de criação do modelo lógico. Como derivação desse processo de mapeamento tem-se o modelo lógico, o qual é diagramado utilizando-se a notação de Chebotko, que representa, de forma visual, a correlação de tabelas (em vez de tarefas, como no diagrama do fluxo da aplicação) e consultas (padrões de acesso).

A notação de Chebotko é utilizada tanto para documentar o modelo lógico quanto o modelo físico. A diferença é que enquanto no modelo lógico se tem apenas as tabelas com suas respectivas colunas e propriedades, no modelo físico é apresentado, além disso, os tipos de dados da coluna e as otimizações realizadas para se instanciar uma tabela no banco de dados com a linguagem CQL. É importante mencionar que o processo demonstrado na **Figura 8** ficará mais claro ao longo do artigo e o objetivo, nesse momento, é apenas fazer uma breve descrição geral.

Para criar um modelo consistente é preciso seguir alguns princípios de modelagem de dados, analisados a seguir:

- **Conheça os dados:** Ao olhar para os componentes de um modelo conceitual (entidades, relacionamentos, atributos, chaves/identificadores e cardinalidades) é preciso compreender os tipos de dados que são capturados pelo mesmo. Com os tipos de dados em mãos, pode-se definir o que deve ser armazenado no banco de dados, lembrando sempre de preservar suas propriedades, ou seja, se devem ser únicos, multivaleados, de tamanho específico (no caso de text), entre outras, de tal forma que os dados sejam organizados a fim de obter um bom desempenho nas consultas. Além do mais, as chaves/identificadores afetam diretamente a maneira como o modelo lógico deve ser projetado, uma vez que as chaves de entidades e relacionamentos se tornam chave primária (simples ou composta) em uma tabela do modelo lógico. Devido a isso, a definição da chave primária é extremamente importante, principalmente por causa das possíveis consultas, as quais só podem ser realizadas por itens da chave primária e também devido à forma de ordenação dos dados na tabela que se deseja, uma vez que caso haja uma chave de ordenação, ela faz parte da chave primária;

- **Conheça as consultas:** As consultas também afetam diretamente o modelo como um todo, pois as tabelas são projetadas com base nelas, e devido a isso, o modelo deve passar por ajustes se as consultas mudarem. Dito isso, existem algumas estratégias de se modelar para atender as consultas, como utilizar uma partição por consulta (ideal) e utilizar mais de uma partição por consulta (aceitável). Entretanto, não se deve fazer consultas em todas as partições da tabela ou fazer consultas em mais de uma tabela para obter o resultado final, visto que esse tipo de operação ocasiona um grande custo de processamento, pois será necessário consultar em mais de um nó do cluster e, consequentemente, o tempo de resposta será maior. O Cassandra consegue rodar várias consultas em paralelo e uma vez que cada partição está distribuída em um nó diferente do cluster, fazer uso de uma partição por consulta é o modo mais eficiente de se recuperar os dados. Nesse momento

# Apache Cassandra: Modelando os dados de sua aplicação NoSQL

é importante lembrar que as tabelas podem ser projetadas para cada partição conter uma ou mais linhas, adequando as partições de acordo com a quantidade de registros a guardar. Assim, se existe uma consulta da qual se espera um único resultado como, por exemplo, buscar o usuário por e-mail (no ato do login), então deve-se modelar uma linha por partição; do contrário, modelase uma partição com várias linhas, pois será esperado mais de um resultado, como em uma listagem de vídeos publicados pelo usuário;

**• Encadeie os dados:** O encadeamento de dados é considerado a principal técnica de modelagem para o Cassandra. Encadear os dados significa organizar múltiplas entidades correlacionadas em uma mesma partição, colocando atributos de mais de uma entidade numa mesma tabela, de tal forma a suportar a estratégia de uma partição por consulta. Deste modo, com uma única consulta é possível obter resultados mais completos, que podem servir para mais de uma tarefa do fluxo da aplicação, diminuindo assim a quantidade de tabelas e consultas necessárias no banco de dados. Por exemplo, uma mesma consulta poderia servir para as tarefas “Usuário loga na aplicação” e “Mostrar vídeos adicionados pelo usuário”, bastando encadear as entidades Usuário e Vídeo numa mesma tabela. Basicamente, existem três formas de encadeamento de dados no Cassandra: chaves de ordenação (gerando partições com várias linhas), colunas com coleções de dados (set, list, etc.) ou colunas de tipos de dados definidos pelo usuário (UDTs – User Defined Types).

A adoção de chaves de ordenação, identificadas pela letra C e uma seta para cima ou para baixo, conforme a Figura 9, é o mecanismo mais utilizado para o encadeamento de dados, pois possui uma série de características interessantes. Nesta mesma imagem, a chave da partição, a qual possui a letra K do lado direito, especifica uma entidade/tabela na qual outras entidades serão encadeadas. Os valores das chaves de ordenação, por sua vez, identificam unicamente as entidades encadeadas. Ademais, se houver mais de uma chave de ordenação, significa que há vários níveis de encadeamento. Logo, na Figura 9 é possível observar uma tabela *actors\_by\_video* cujo objetivo é listar os atores de acordo com o respectivo vídeo (tabela *Vídeos* da figura) em que atua. Note ainda que essa tabela possui múltiplos níveis de encadeamento, pois apresenta duas chaves de ordenação: *actor\_name* e *character\_name*.

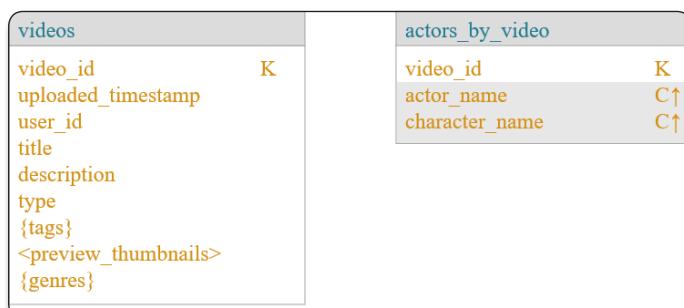


Figura 9. Encadeamento de múltiplos níveis

O segundo mecanismo de encadeamento de dados mais utilizado é a criação de um UDT (*User Defined Type*), ou tipo de dado definido pelo usuário, o qual é um tipo de dados que possui uma estrutura customizada. Ao contrário dos tipos de dados primitivos e complexos, o UDT é um tipo de dado especial que pode ser utilizado para representar uma entidade com o intuito de colocá-la numa tabela que possui colunas de outra entidade, sinalizando um relacionamento do tipo um para um (1-1). Por exemplo, na Figura 10 há uma tabela com uma coluna chamada *user\_id*, a qual pertence à entidade *Usuário*, e uma coluna *videos*, cujo tipo de dado é *video\_type*. Este tipo é um UDT que representa uma entidade hipotética de mesmo nome, para referenciar os tipos de vídeo. Logo, numa única consulta pelo *user\_id* é possível atender à tarefa de “Listar todos os vídeos publicados por um usuário”, uma vez que será retornado o usuário consultado com sua respectiva coleção de *video\_types*, os quais possuem os dados relacionados a cada vídeo.

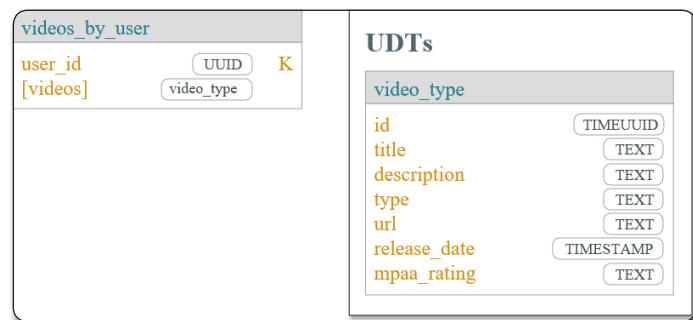


Figura 10. Encadeamento com tipo de dado definido pelo usuário (UDT).

**• Duplicar dados.** A duplicação de dados é o último dos princípios de modelagem para o Cassandra, mas não menos importante. Neste ponto é válido salientar que a utilização dos dois princípios anteriores pode resultar em duplicação de dados, uma vez que tabelas diferentes respondem a consultas diferentes e, portanto, se possuírem colunas iguais, haverá duplicação de um mesmo dado em mais de uma tabela. Com a duplicação dos dados as consultas são pré-executadas e materializadas (o resultado esperado da consulta já é gravado numa tabela especificada), não sendo necessário, portanto, realizar a filtragem de dados ou qualquer cálculo, pois tudo já terá sido feito anteriormente, diminuindo assim o tempo de resposta e o custo de processamento. Saiba também que existem várias formas de se duplicar os dados, como em tabelas, partições e/ou linhas, ou seja, dados iguais podem estar em tabelas, partições e até mesmo em linhas de uma mesma tabela. No exemplo da Figura 11, o título do vídeo está duplicado por tabela.

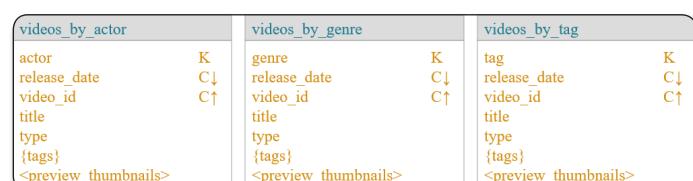


Figura 11. Duplicação de dados por tabela

## Regras de mapeamento

As regras de mapeamento são orientações a serem seguidas a fim de mapear as entidades do modelo conceitual para o modelo lógico. Essas regras, categorizadas em cinco etapas, protegem o modelo contra eventuais erros ao projetar cada tabela para atender a uma única consulta, retornando o resultado na ordem desejada. Vamos a elas:

1. Mapeamento de Entidade e Relacionamento;
2. Mapeamento de Atributos de Busca Igualitária (filtro de igualdade  $=$ );
3. Mapeamento de Atributos de Busca Desigual (filtros de desigualdade  $>=$ ,  $<=$ ,  $>$ ,  $<$ );
4. Mapeamento de Atributos de Ordenação;
5. Mapeamento de Atributos Identificadores.

### Nota

Saiba que as regras de mapeamento não são passos para a construção de um Modelo Entidade Relacionamento (MER), pois apesar de existir um MER como modelo conceitual, as regras de mapeamento apenas fazem uso do mesmo para criar um modelo lógico sem relações entre entidades, no qual cada tabela atende a uma consulta identificada no fluxo da aplicação.

## Regra 1 – Mapeamento de Entidade e Relacionamento

As entidades e relacionamentos devem ser mapeados para tabelas, nas quais cada linha representa uma instância ou unidade da entidade a qual ela representa e seus respectivos relacionamentos. Ao fazer isso, a intenção da regra é criar um modelo de dados com partições de múltiplas linhas, nas quais se armazenam dados relativos a uma ou mais entidades, diminuindo a fragmentação das linhas em vários nós diferentes, pois cada partição fica armazenada em um único nó, evitando assim uma queda na performance. Nessa regra, os atributos das entidades provenientes do modelo conceitual devem se tornar colunas nas tabelas do modelo lógico. Além disso, os dados do modelo conceitual devem ser preservados no modelo lógico, o que não impede que o mesmo dado possa ser duplicado por várias tabelas, partições e linhas. A **Figura 12** mostra um exemplo de modelagem da tabela *usuario*, seguindo as diretrizes do mapeamento.

Continuando a abordagem da regra de mapeamento 1, os relacionamentos devem se transformar em tabelas, de tal forma que os atributos das entidades relacionadas sejam representados por colunas nas mesmas. Neste ponto é válido destacar que a cardinalidade do relacionamento afeta diretamente o projeto da chave primária e eventuais escolhas erradas na composição da mesma podem levar a tabelas que não conseguem realizar pesquisas por determinados atributos, pois no Cassandra só é possível realizar pesquisas através

dessa chave. Além disso, saiba que todo relacionamento entre duas entidades tem o potencial de gerar duas tabelas, cada uma representando uma direção do relacionamento, uma vez que todo relacionamento é, por natureza, bidirecional. Logo, como demonstrado na **Figura 13**, o relacionamento 1-N entre as entidades *Usuario* e *Video* pode (ou não, a depender da necessidade do negócio) gerar as tabelas *videos\_do\_usuario*, a qual tem o intuito de listar os vídeos de determinado usuário, obtendo os usuários com seus respectivos vídeos, e *usuarios\_do\_video*, que, por sua vez, tem a intenção oposta, de listar os usuários de um determinado vídeo. Ao observar a figura, enquanto *videos\_do\_usuario* suporta consultas pelo identificador do usuário (a coluna *usuario\_id* é chave primária - K) e opcionalmente pelo identificador do vídeo (a coluna *video\_id* é chave de ordenação - C), *usuarios\_do\_video* suporta apenas consultas pelo identificador do vídeo, uma vez que no Cassandra só é possível realizar consultas utilizando chaves primárias (obrigatório) e chaves de ordenação (opcional). Logo, não é possível consultar pelo *usuario\_id* em *usuarios\_do\_video*.

### Nota

Nada impede que se escolha colocar a coluna *usuario\_id* como chave de ordenação na tabela *usuarios\_do\_video*, a fim de possibilitar a consulta por ela. Entretanto, para o caso de uso do negócio que seria obter o usuário de um determinado vídeo, a utilização do identificador do vídeo como chave primária é suficiente. Ademais, não faria sentido pesquisar informando além do *video\_id*, o identificador do usuário, pois já saberíamos, de antemão, de qual usuário se trata. Portanto, a escolha de qual coluna vai ser que tipo de chave é meramente objetiva, levando em conta apenas os tipos de consultas que a tabela deve suportar.

## Regra 2 – Mapeamento de Atributos de Busca Igualitária

Quando se pensa em consultas no Cassandra, deve-se analisar quais delas precisam de buscas igualitárias (utilizando o sinal de igual,  $=$ ) e quais precisam de buscas desiguais (isto é, que



Figura 12. Modelagem da tabela *Usuario* conforme a regra 1

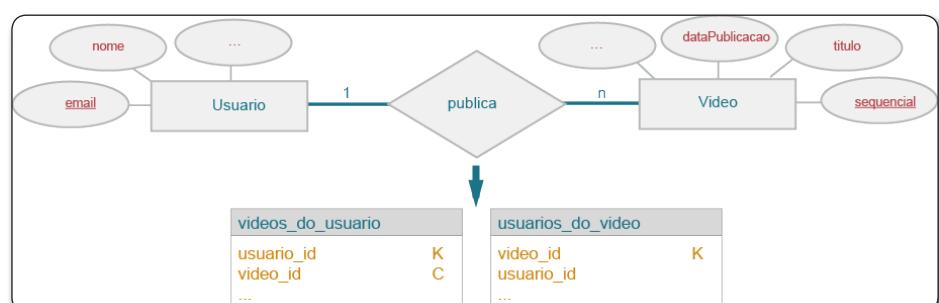


Figura 13. Mapeamento do relacionamento entre as entidades *Usuário* e *Vídeo*

# Apache Cassandra: Modelando os dados de sua aplicação NoSQL

fazem uso de sinais como maior que, menor que, menor ou igual, maior ou igual). Para atender à consulta, os atributos das entidades que precisam do critério de igualdade devem estar presentes na chave primária da tabela, do contrário a pesquisa não pode ser realizada. Por conseguinte, as consultas suportadas pela tabela devem, obrigatoriamente, incluir todas as chaves de partição na mesma, pois sem elas não é possível encontrar a partição em que o dado se encontra, impossibilitando a busca do mesmo. A Figura 14 demonstra os componentes da chave primária que podem ser utilizados para a realização de buscas igualitárias.

Ao aplicar essa regra para a entidade Vídeo, pegam-se os atributos da consulta que precisam ser filtrados pelo sinal de = e transforma-os em chaves primárias

em uma tabela. A Figura 15 demonstra algumas possibilidades de modelagem, caso fosse necessário realizar filtragens pelo título e pela data de publicação de um vídeo, por exemplo.

Outro exemplo de aplicação da regra 2 é apresentado na Figura 16, agora para tabelas que representam relacionamentos entre entidades do modelo conceitual. Neste caso, a relação entre as entidades Usuário e Vídeo poderia ser traduzida em duas tabelas (*videos\_do\_usuario\_1* e *videos\_do\_usuario\_2*), as quais visam atender à consulta “Listar vídeos de um determinado usuário informando o nome e a data de criação da conta do mesmo no sistema”. Logo, os atributos *dataCriacao* e *nome* da entidade Usuário devem ser filtrados por igualdade (*nome* = X e *dataCriacao* = Y), transformando-

os em chaves de partição (especificado como K na legenda), no caso de *videos\_do\_usuario\_1*, e em chave de partição e ordenação (especificado como C), no caso de *videos\_do\_usuario\_2*. Note que essas tabelas apresentam maneiras diferentes de responder à mesma consulta, sendo que na primeira os resultados serão ordenados pelo sequencial do vídeo de forma ascendente, enquanto na segunda serão ordenados primeiro pelo nome e em seguida pelo sequencial, ambos de forma ascendente.

## Regra 3 – Mapeamento de Atributos de Busca Desigual

As buscas desiguais, como mencionado anteriormente, se referem a filtros nas consultas que fazem uso de sinais de desigualdade. Todos os atributos que possuem a necessidade de serem filtrados dessa maneira se tornam chaves de ordenação (*clustering columns*) na chave primária da tabela. Por conseguinte, as colunas que se tornam chaves de ordenação devem ser definidas na chave primária, após as colunas que participam de buscas igualitárias, como expõe a Figura 17. É importante mencionar que as chaves de partição não podem ser utilizadas em filtros de desigualdade e não podem ser utilizadas mais de uma chave de ordenação em buscas desiguais.

Aplicando a regra 3 à entidade Usuário, levando em consideração o atributo de busca igualitária o nome e o atributo de busca desigual *dataCriacao*, tem-se o diagrama apresentado na Figura 18. Note que foi colocado o nome como chave da partição e o e-mail passou a ser chave de ordenação, a fim de também permitir a busca pelo mesmo.

A regra 3 também pode ser aplicada a relacionamentos, como no caso da Figura 19, que demonstra a aplicação dessa regra ao relacionamento entre as entidades Usuário e Vídeo. Neste caso, temos uma consulta que informa o e-mail do usuário (busca de igualdade) e uma data de publicação do vídeo maior que (busca desigual) uma data especificada, ou seja, algo como “e-mail = algumemail@xxx.com AND *dataPublicacao* > XXXXX”.

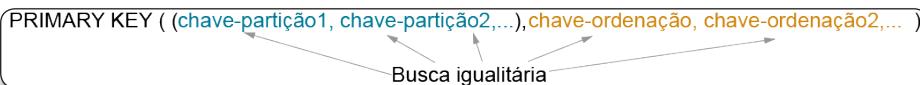


Figura 14. Componentes da chave primária que podem ser filtrados por busca igualitária



Figura 15. Exemplo de aplicação da regra 2 para a entidade Vídeo

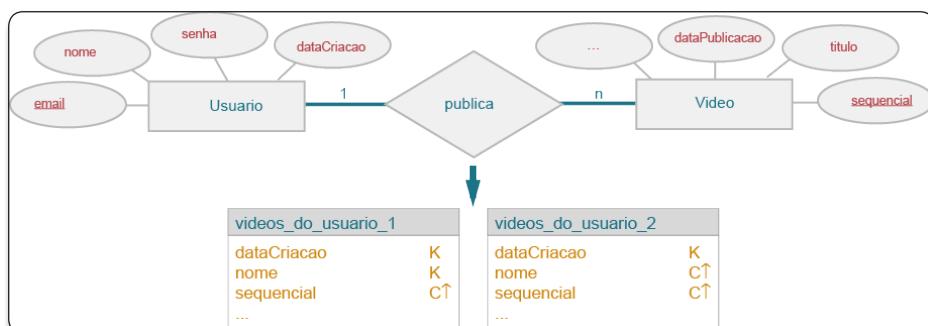


Figura 16. Mapeando relacionamento entre Usuario e Video conforme a regra 2



Figura 17. Colunas da chave primária e tipos de buscas

#### Regra 4 – Mapeamento de Atributos de Ordenação

Nesta regra, define-se que os atributos de ordenação das entidades especificadas no modelo lógico passam a ser chaves de ordenação (*clustering columns*) numa tabela. Feito isso, as linhas da tabela serão ordenadas de acordo com os valores das chaves de ordenação, de forma ascendente ou descendente, conforme pode ser observado na **Figura 20**.

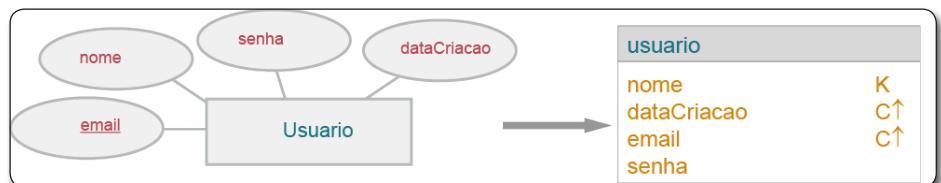
Ao aplicar a regra 4, supondo que se deseja fazer uma consulta pelo nome e pela data de criação do usuário maior que uma determinada data, ordenada de forma ascendente pela data de criação, a modelagem fica de acordo com a **Figura 21**.

Agora, ao aplicar a regra 4 para relacionamentos, analisando o exemplo da relação entre usuário e vídeo, a fim de atender uma consulta que busca uma lista de vídeos publicados pelo usuário após uma determinada data, ordenando o resultado da data mais antiga para mais recente, teremos como resultado a tabela exposta na **Figura 22**.

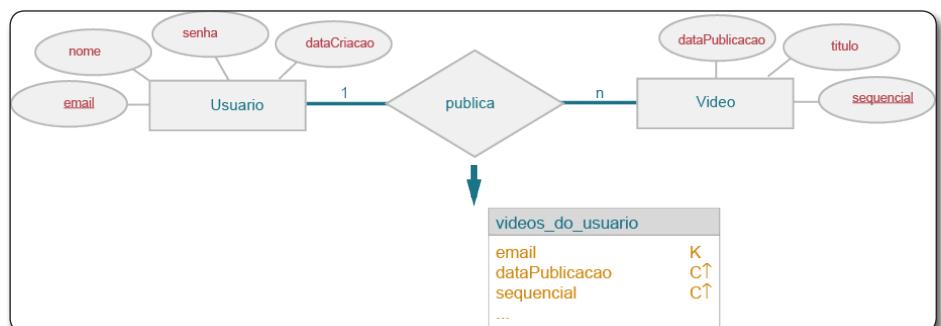
#### Regra 5 – Mapeamento de Atributos Chave ou Identificadores

A chave primária da tabela, derivada da aplicação da regra 5, deve conter as colunas identificadoras da entidade no modelo conceitual, sem que seja necessária uma preocupação com a posição (acima ou abaixo) e a ordenação destas. Com o intuito de permitir consultas específicas, no entanto, essa chave pode conter algumas colunas adicionais. Dito isso, para uma melhor compreensão da aplicação da regra 5, a **Figura 23** demonstra duas modelagens possíveis para a entidade *Usuário*.

A primeira tabela modelada, denominada *usuarios*, passou a ter o *email* como único representante da chave primária, sendo também chave de partição. Essa modelagem é perfeita para uma consulta em que se deseja apenas obter os dados do usuário informando o e-mail. Por outro lado, se fosse necessário pesquisar pelo nome do usuário, não seria possível nesta tabela, pois o atributo nome não compõe a chave primária. A fim de solucionar esse problema, pode-se definir também a tabela *usuarios\_por\_nome*, demonstrada na



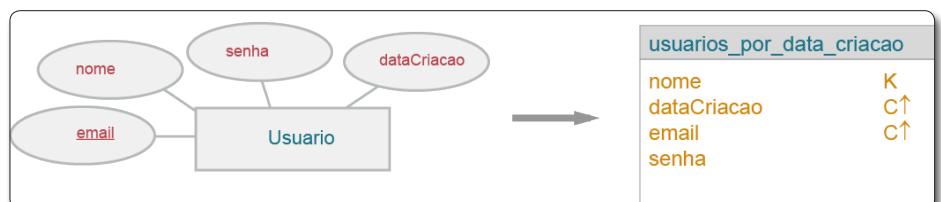
**Figura 18.** Mapeamento da entidade Usuário com a regra 3



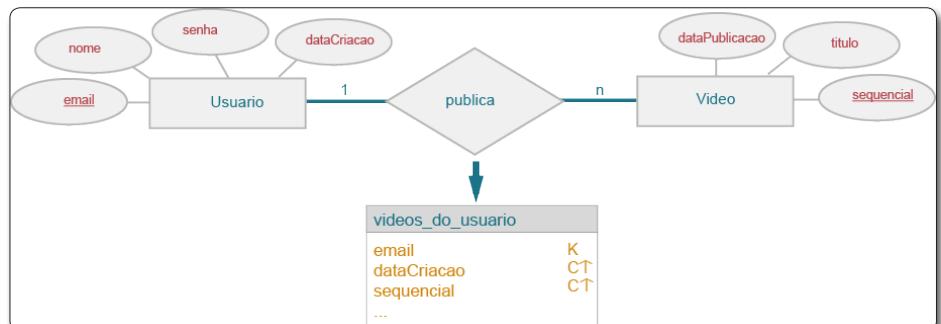
**Figura 19.** Mapeamento do relacionamento entre Usuário e Video conforme a regra 3

```
PRIMARY KEY ((chave-particao1,chave-particao2,...),coluna-ordenação1,coluna-ordenação2,...)
ORDER BY coluna-ordenação1 ASC, coluna-ordenação2 ASC, ...
```

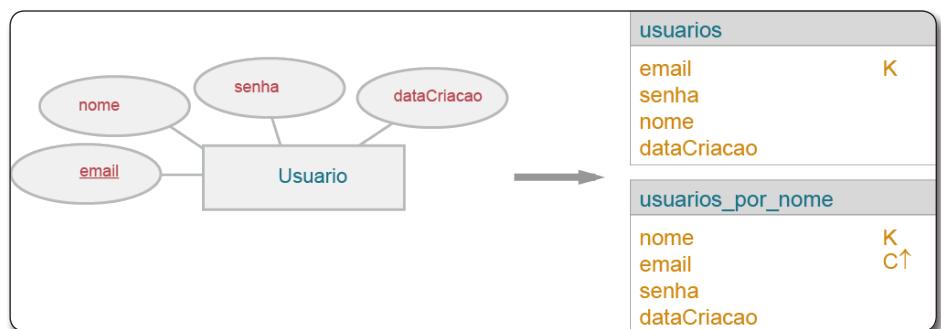
**Figura 20.** Chaves/colunas de ordenação na chave primária



**Figura 21.** Modelagem da tabela usuarios\_por\_data\_criacao



**Figura 22.** Modelagem da tabela videos\_do\_usuario a partir de uma consulta especificada



**Figura 23.** Modelagem da entidade usuário, conforme a regra 5

mesma figura, na qual o nome é a chave da partição e o e-mail a chave de ordenação, ambos compondo a chave primária da tabela.

### Aplicando as regras de mapeamento ao modelo do exemplo

A primeira relação que será modelada é entre as entidades Usuário e Vídeo. Por ser do tipo 1-N, esse relacionamento sinaliza que cada Usuário pode publicar um ou mais Vídeos. Logo, uma das possíveis consultas que se pode deduzir para essa relação é a “listar vídeos do usuário de forma ordenada (mais antigo para o mais novo ou vice-versa)”. Assim, de acordo com a regra 1 as entidades e seus relacionamentos devem se tornar uma tabela.

Para uma modelagem otimizada, recomenda-se que cada relacionamento entre duas entidades seja modelado como uma tabela. Assim, devemos criar uma apenas para a relação entre as entidades Usuário e Vídeo. Essa restrição é necessária para viabilizar uma economia de espaço na tabela por conta do limite de dados por partição (quanto mais encadeamento de dados, mais dados na partição) e direcionar o modelo para consultas específicas, o que por consequência otimizará o tempo necessário para retornar o resultado desejado. Seguindo, então, a regra 1, tem-se *videos\_do\_usuario*, a qual uma consulta pelo usuário deve retornar todos os vídeos publicados pelo mesmo.

A segunda regra se refere ao mapeamento de atributos de buscas igualitárias, ou seja, aqueles atributos que devem fazer parte do filtro de igualdade da consulta de forma a retornar o resultado desejado. Pensando nisso, nota-se claramente que em *videos\_do\_usuario* é preciso filtrar por igualdade os dados do identificador do usuário. A regra informa ainda que os atributos que necessitam participar dessa filtragem devem compor a chave primária. Dessa forma, o segundo passo é definir a coluna *email* como chave primária, identificando-a com a letra K.

O terceiro passo do mapeamento é a aplicação da terceira regra, a qual especifica que atributos de buscas desiguais devem se tornar chaves de ordenação e, portanto, devem fazer parte da chave primária. Como em nosso caso é preciso realizar uma filtragem por desigualdade pelo atributo *dataPublicacao* da entidade Vídeo, permitindo ordenar o resultado de forma ascendente (mais antigos primeiro) ou descendente (mais recentes primeiro) – a fim de atender à consulta mencionada no primeiro parágrafo deste tópico – define-se a coluna *data\_publicacao* com a letra C (alusão à *clustering column*) e com a seta ↑, resultando em C↑. A seta para cima significa ordem ascendente. Se quiser modelar com a ordem descendente, utiliza-se a seta apontando para baixo. É válido ressaltar que toda chave de ordenação precisa ter especificada a sua forma de ordenação.

A quarta regra é um pouco parecida com a terceira, no que diz respeito à chave de ordenação. A ideia dessa regra é definir na tabela os atributos que precisam ser ordenados para a consulta, transformando-os em chaves de ordenação. Visto que em nosso exemplo há um atributo que é desejável que seja ordenado para o retorno da consulta, o identificador do Vídeo, especifica-se a coluna *sequencial\_video* como uma chave de ordenação.

A quinta regra se refere à composição da chave primária, a qual deve conter o identificador de cada entidade no relacionamento. Conforme a aplicação das regras anteriores, até o momento a chave primária a ser gerada já possui as seguintes colunas: *email*, *sequencial\_video* e *data\_publicacao*. Logo, os requisitos para a regra 5 já foram cumpridos com a coluna *email*, pois é o atributo identificador de uma entidade, e *sequencial\_video*, como o identificador da outra.

A **Figura 24** ilustra a tabela resultante da aplicação das regras de mapeamento. A partir dela, uma consulta pelo identificador do usuário vai retornar todos os vídeos já publicados pelo mesmo, de forma ascendente.

videos_do_usuario	
email	K
<i>data_publicacao</i>	C↑
<i>sequencial_video</i>	C↑
<i>dataCriacao</i>	

Figura 24. Tabela resultante da aplicação das regras de mapeamento

### Passo 3: Modelagem Física

O último passo se refere a dois blocos do diagrama do processo de modelagem: otimização do modelo e modelagem do modelo físico (vide **Figura 4**). Esses dois blocos fazem parte de uma grande atividade, que é a criação do modelo físico. O primeiro objetiva encontrar problemas no modelo ao realizar análises, validações e otimizações necessárias para um melhor funcionamento do esquema como um todo. O segundo, visa o retoque final, transformando o modelo lógico num modelo físico ao adicionar tipos de dados às colunas das tabelas definidas, bem como ao especificar as rotinas de criação das tabelas com a linguagem CQL.

O modelo lógico que foi gerado seguindo as técnicas demonstradas até aqui está correto, funciona de forma apropriada e com uma performance eficiente, afinal, foram empregadas todas as regras e boas práticas. Na realidade, no entanto, a eficiência do modelo lógico não pode ser constatada sem a execução de testes de desempenho com ferramentas adequadas, pois o banco de dados possui limitações, os recursos são finitos (nós, espaço em disco, memória, etc.). Dessa forma, é preciso realizar uma análise do modelo a fim de encontrar potenciais problemas, ao levar em conta todas as limitações do ambiente de hardware e software. Entre as coisas que devem ser avaliadas estão o tamanho da partição de uma tabela, a redundância e consistência dos dados, operações de junções realizadas pela aplicação, regras de integridade referencial, transações e agregações de dados. Essa análise é primordial para otimizações mais detalhadas, de forma que o modelo possua um desempenho ainda melhor.

Para finalizar o modelo lógico criado, transformando-o em modelo físico, é preciso identificar os tipos de dados de cada coluna na tabela especificada, de tal forma que seja possível utilizar a

linguagem CQL para criar as tabelas. A partir disso, como exemplo fica a modelagem física da tabela *videos\_do\_usuario*, apresentada na Figura 25.



Figura 25. Representação da tabela *videos\_do\_usuario* no modelo físico

#### Nota

Existem basicamente três formas de otimizar o modelo: melhorando as chaves (de partição, primária e de ordenação), melhorando as tabelas e melhorando a concorrência de acesso a dados.

Este artigo teve como principal objetivo prover uma visão geral da modelagem de dados para o banco de dados Apache Cassandra, através de uma abordagem introdutória. Dessa forma, ainda existe muito assunto a ser explorado e que pode ser o objeto de estudo de outros artigos no futuro. O importante, neste momento, é entender os conceitos básicos e começar a colocá-los em prática em seus projetos.

Por fim, saiba que o Cassandra pode fazer muito mais do que o demonstrado aqui, sendo, portanto, de suma importância que os leitores consultem a bibliografia disposta na seção **Links**. Dito isso, explorem esse banco de dados, conheçam sua arquitetura, frameworks relacionados, configurações avançadas e também casos de uso, pois é importante saber quando e como fazer uso desta diferenciada opção.

#### Autor



**José Guilherme Macedo Vieira**

jguilhermemv@gmail.com



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

#### Links:

**Site oficial do Apache Cassandra.**

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

**DataStax - Site oficial**

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

**DataStax - Documentação do Cassandra**

[http://docs.datastax.com/en/cql/3.1/cql/cql\\_reference/cqlReferenceTOC.html](http://docs.datastax.com/en/cql/3.1/cql/cql_reference/cqlReferenceTOC.html)

**Site oficial da ferramenta KDM, para automação da modelagem de dados do Cassandra.**

<http://kdm.dataview.org/>

**Tese sobre a metodologia utilizada neste artigo.**

<http://www.cs.wayne.edu/andrey/papers/TR-BIGDATA-05-2015-CKL.pdf>

#### Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)

Ajude-nos a manter a qualidade da revista!



# Spring Boot: Como desenvolver microserviços seguros

Veja neste artigo algumas das opções disponíveis para isso e como garantir a segurança de microserviços através de autenticação

**A**té poucos anos atrás, grande parte dos desenvolvedores Java desenvolviam aplicações corporativas organizadas como sistemas monolíticos, contendo um ou mais módulos, implantados como um único pacote (WAR ou EAR) em um servidor de aplicações (ex.: GlassFish, JBoss AS). Esse conceito ainda é amplamente adotado no mercado, mas uma alternativa promissora começa a ganhar espaço: a arquitetura baseada em microserviços.

Com a demanda pelo desenvolvimento de sistemas cada vez mais complexos e o consequente aumento no número de dependências e linhas de código, a rotina de um desenvolvedor que precisa trabalhar com aplicações monolíticas tornou-se cada vez mais difícil. Além do grande número de módulos, componentes, pacotes e classes, que vão agregando funcionalidades que visam contemplar a cadeia completa do modelo de negócio em questão, o desenvolvedor precisa reimplantar a sua aplicação em um servidor de aplicações a cada modificação. Para lidar com isso algumas estratégias foram desenvolvidas como, por exemplo, o plugin do JRebel, que realiza o *hot-deploy* (implantação instantânea, sem necessidade de reiniciar o serviço) da aplicação no servidor. No entanto, qualquer modificação mais complexa, como a criação de novos componentes de negócio, demanda um comando de reinicialização completo, o que pode levar alguns minutos.

Fora a queda de produtividade devido à demanda de implantação de um pacote no servidor a cada modificação, o desenvolvedor de aplicações monolíticas ainda precisa lidar com outros problemas clássicos de

## Fique por dentro

Este artigo apresenta uma maneira simplificada de adicionar autenticação em microserviços baseados no Spring Boot. Os microserviços são aplicações autônomas que normalmente utilizam um servidor web embarcado e disponibilizam uma interface REST para interação com sistemas externos. Com esse novo estilo arquitetural, torna-se crucial a adição de componentes que realizem a segurança e só permitam o acesso de clientes (aplicações ou usuários) que tenham autorização para executar determinada ação.

Lembre-se que independentemente do tipo de aplicação, é de suma importância a proteção contra requisições indevidas e isso se torna ainda mais crítico quando se trata de microserviços baseados em REST, visto que toda forma de comunicação ocorre sobre o protocolo HTTP, podendo ser acessado por qualquer usuário na web. Com base nisso, este artigo busca explorar as opções que possibilitam construir microserviços mais seguros.

aplicações altamente acopladas, como a impossibilidade de manter módulos funcionando de maneira independente, a dificuldade da escalabilidade de um serviço específico (que demande mais processamento), o alto risco de inserção de efeitos colaterais após uma modificação e a dificuldade de integração com sistemas externos, visto que uma interface única de acesso normalmente é enorme e bastante suscetível a mudanças. Como se torna difícil manter um padrão, ou acabam-se criando interfaces especializadas para determinados clientes, o que gera ainda mais acoplamento entre os sistemas, ou o cliente precisará estar constantemente atualizando a sua interface de comunicação com o servidor.

Então, como pensar em uma arquitetura viável para este cenário? Que tal pensarmos em aplicações com o escopo reduzido e focadas em um serviço específico, que possam ser distribuídas pela nuvem e não dependam da implantação em um servidor externo? Esse é o conceito-chave por trás dos microserviços.

Neste cenário, ao invés de um grande pacote EAR implantado em um servidor de aplicações, vamos ter um conjunto de aplicações empacotadas cada uma como um simples JAR, que não precisam ser implantados em nenhum servidor. Basta executar a aplicação via `java -jar` que ela estará pronta para ser acessada através de qualquer cliente HTTP.

De maneira mais prática, é como se desmembrássemos um sistema monolítico em vários sistemas menores, que executam de maneira independente. Considerando como exemplo um sistema de loja on-line, poderíamos elencar vários microserviços, tais como: controle de logística, visualização de produtos, mala direta, emissão de notas fiscais, promoções, entre tantos outros; ou seja, são inúmeras as possibilidades. Com esse desmembramento, torna-se possível atualizar versões de quaisquer serviços sem ter de parar toda a infraestrutura, organizar o armazenamento de cada microserviço adotando uma tecnologia mais adequada para aquele propósito (ex.: NoSQL, In-memoryDB) ou até mesmo, se for o caso, desenvolver cada um em uma linguagem de programação diferente. Enfim, essa nova solução nos dá a liberdade total para desenvolver cada parte do negócio e ainda força o desenvolvedor a pensar em sua aplicação com uma visão mais focada.

Dentro desse ambiente tão livre, surge um novo requisito, que até então não havia nas aplicações monolíticas: a necessidade de uma camada de segurança nos microserviços. Considerando o exemplo do sistema de loja on-line, não seria adequado que um usuário sem a devida autorização consiga alterar o preço de um produto. Portanto, apesar de sua característica autônoma, cada microserviço precisa se adequar a um escopo de segurança, de forma que cada requisição realizada seja tratada adequadamente e só após a confirmação das credenciais válidas, que o acesso deve ser liberado.

Assim, o objetivo deste artigo é fornecer um passo a passo de como construir um microserviço seguro, evitando que acessos indevidos sejam realizados, fornecendo uma estrutura de autenticação flexível, que possa ser usada de maneira autônoma por cada serviço.

## Explorando o Conceito de Microserviços

O autor Sam Newman, no livro “Building MicroServices” (2015), em uma definição simples, afirma que “Microserviços são serviços pequenos, autônomos, que trabalham em conjunto”.

Por trás dessa definição, que de início parece simples, há uma mudança de paradigma bastante extensa. Até então, boa parte dos desenvolvedores ou arquitetos de software procura, no início do projeto, modelar uma solução que atenda, da melhor maneira, os requisitos que foram levantados junto ao cliente, de modo que seja entregue um produto “o mais perfeito possível”. Mesmo assim, é comum o surgimento de novas demandas quando a

solução começa a ser utilizada. Logo, caso o sistema não tenha sido desenvolvido em cima de uma arquitetura flexível, que suporte facilmente modificações e adaptações, o esforço para realizar as alterações solicitadas pode acabar gerando bastante retrabalho. Neste cenário, a cada mudança são gerados ajustes em cima de ajustes, visando atender às necessidades do usuário e então o sistema começa a se transformar em uma “colcha de retalhos”, dificultando ainda mais a sua evolução, o que acaba piorando com o passar do tempo.

A popularização da Computação em Nuvem foi um dos fatores primordiais para a difusão dessa nova arquitetura, tendo em vista que manter servidores físicos para cada microserviço seria uma tarefa extremamente onerosa. Juntamente com o conceito de infraestrutura na nuvem, surge uma nova mentalidade para o desenvolvimento de software, com novos objetivos, princípios e práticas. Um exemplo de movimento que promove essa nova mentalidade é o 12factor, que foi um dos pioneiros em relação a definir um conjunto de boas práticas para o desenvolvimento de software na infraestrutura baseada em microserviços. Logo, fica claro que o conceito de microserviços é algo bem mais amplo do que apenas decompor o seu projeto em um conjunto de serviços menores. É preciso desenvolvê-los e organizá-los de maneira concisa, preocupando-se principalmente com o protocolo de comunicação entre os serviços que formam a infraestrutura do sistema como um todo.

Para ilustrar o desenvolvimento de um microserviço, será utilizado neste artigo o Spring Boot, um framework que disponibiliza os principais recursos do tradicional Spring Framework, com o diferencial de possuir um mecanismo de configuração simplificado (baseado em um único arquivo de propriedades) e de poder ser distribuído como um pacote autônomo, dispensando a obrigatoriedade da utilização de um servidor de aplicações.



## Conhecendo o Spring Boot

O Spring Framework é uma das tecnologias mais adotadas para a construção de aplicações em Java. O framework disponibiliza vários módulos que buscam contemplar diversos recursos de uma aplicação corporativa, tais como persistência, segurança, injeção de dependências, programação orientada a aspectos, entre outros. Desta maneira, o Spring Framework visa facilitar a vida do desenvolvedor, fazendo com que o seu foco seja primariamente as regras de negócio da aplicação que está sendo desenvolvida, poupando tempo do desenvolvimento de recursos básicos, que em sua maioria funcionam de maneira padrão para várias aplicações.

No entanto, apesar de ser bastante conhecido no mercado, muitos desenvolvedores acabam se afastando do Spring Framework devido à complexidade inicial para configurar e compreender sua estrutura. Como tentativa de simplificar essa configuração, surge o Spring Boot, que nada mais é do que um *plugin* (com suporte do Maven e Gradle), que possibilita aos desenvolvedores construir aplicações utilizando quaisquer módulos disponibilizados pelo Spring Framework, sem ter de realizar nenhuma configuração complexa inicialmente.

Além disso, o Spring Boot possibilita a criação de aplicações autônomas, ou seja, aplicações que não dependem da configuração de um serviço externo (ex.: servidor de aplicações) para funcionar. Para isso, há a opção de embarcar um *container* no próprio pacote da aplicação (um simples JAR), que pode ser executado pelo comando “java –jar”.

Em resumo, o Spring Boot contempla, de maneira simples e eficiente, os requisitos para a construção de microserviços, aliando todo o aparato de recursos do Spring Framework, que vem sendo utilizado por décadas para a implementação de aplicações em Java, com a condição de configuração mínima e o funcionamento de maneira autônoma.



## Construindo a primeira aplicação no Spring Boot

O primeiro passo para construir uma aplicação utilizando Spring Boot é importar as dependências básicas. A **Listagem 1** mostra as dependências que precisam ser adicionadas usando o Maven. São elas:

- **spring-boot-starter-web**: adiciona o suporte aos componentes web, tais como filtros, servlets, etc., que serão necessários mais adiante para a configuração do protocolo de comunicação com os clientes externos;
- **spring-boot-starter-tomcat**: possibilita que o projeto rode de maneira autônoma, levando no mesmo pacote um Tomcat embalado. Desta maneira, será possível utilizar o comando `java –jar seujar.jar` para executar a aplicação e logo após acessá-la por um navegador web, sem a necessidade de configurar nenhum serviço adicional no servidor.

### Listagem 1. Adicionando dependências do Spring Boot.

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
  </dependency>
</dependencies>
```

Caso você utilize a API de persistência JPA, será necessária a importação da dependência **spring-boot-starter-data-jpa**, além das configurações de acesso ao banco, que ficarão no arquivo *application.properties* (ou *application.yml*). Você pode usar qualquer um desses dois formatos que o Spring Boot identifica automaticamente, desde que o arquivo esteja na pasta raiz ou na pasta *resources*.

Após isso, basta criar a classe principal que será responsável por executar a aplicação, como mostra a **Listagem 2**.

### Listagem 2. Classe principal na qual a aplicação será executada.

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Main.class)
    }
}
```

A anotação **@SpringBootApplication** funciona como um simplificador, englobando internamente um conjunto de outras anotações necessárias para a configuração inicial de uma aplicação com o Spring Boot. São elas:

- **@Configuration**: indica que a classe pode declarar métodos com a anotação **@Bean**, que serão processados em tempo de execução e injetados pelo container interno do Spring;

- **@EnableAutoConfiguration:** realiza a configuração automática dos componentes que a aplicação precisa para funcionar;
- **@ComponentScan:** provê suporte ao escaneamento, ou seja, detecção automática de classes a partir do *classpath*, sem demandar nenhuma configuração manual em arquivos XML.

Assim, basta colocar a anotação **@SpringBootApplication** que todas essas já serão aplicadas à sua classe.

Só o fato de ser adicionada a dependência para o Spring Security já faz com que a aplicação seja protegida por uma camada que irá exigir autenticação, com base em credenciais que devem ser definidas no arquivo de configuração padrão do Spring Boot (*application.properties/yml*). Mais à frente será apresentado como isso pode ser feito de maneira simplificada. Antes disso, é preciso entender que tipo de proteção será aplicada aqui, em relação à diferenciação entre os conceitos de autenticação e autorização.

#### Nota

É preciso um certo cuidado com a hierarquia de pacotes da sua aplicação. A classe principal, que contém a anotação **@SpringBootApplication**, deve estar em um nível acima das demais classes da aplicação, visto que ela precisará ler as demais para realizar a configuração automática. Isso significa que, se a classe principal estiver no pacote `br.com.devmedia.app`, as demais classes do projeto deverão seguir a mesma hierarquia como, por exemplo, `br.com.devmedia.app.entidades`, `br.com.devmedia.app.beans`, etc., do contrário, não serão reconhecidas.

## Autenticação VS Autorização

Autenticação e autorização são conceitos utilizados para validar pessoas ou coisas que interagem com sistemas. No contexto de segurança, *autenticação* é o processo pelo qual podemos confirmar que o requisitante é quem ele realmente diz ser. Usuários humanos normalmente são autenticados através de credenciais como nome de usuário e senha. Assim, podemos determinar que somente aquele usuário tenha acesso a uma operação específica, como também garantir que aquele usuário, uma vez autenticado, será responsável por qualquer modificação, em nível de dados, realizada por ele no sistema. Logicamente, mecanismos de autenticação mais modernos podem ser adotados, como o uso de certificados digitais ou até mesmo a autenticação via impressão digital, que já é uma realidade para os dispositivos móveis. No entanto, o mecanismo mais popular na Web hoje para autenticação ainda é a validação de credenciais simples, como *login* e senha.

*Autorização* é um mecanismo no qual são mapeadas quais operações determinado usuário pode realizar. Uma vez que o usuário esteja devidamente autenticado, pode-se atrelar ele a um determinado perfil de acesso, que irá indicar quais operações dentro do sistema ele estará apto a acessar.

Em sistemas monolíticos é comum que a própria aplicação gerencie a autenticação e autorização. No entanto, quando pensamos em microserviços, podemos chegar a cenários mais complexos, nos quais é possível informar as mesmas credenciais para se au-

tenticar em aplicações diferentes. Logo, dependendo do escopo da aplicação, podem inclusive existir serviços especializados apenas nesta tarefa de autenticação.

## Estratégias de segurança para autenticar serviços

Com a utilização da rede como mecanismo geral de comunicação entre os microserviços, torna-se crucial a preocupação com segurança. Neste sentido, além da preocupação com a validação do acesso de um usuário e suas respectivas permissões (o que já é mais comum a aplicações de modo geral), também surge a necessidade de validar o acesso proveniente de outros sistemas (autenticação serviço a serviço), o que acaba sendo necessário quando lidamos com microserviços, tendo em vista que a forma de acesso a esses serviços ocorre através de APIs utilizando o protocolo HTTP, o que, por padrão, não traz restrições para o acesso proveniente de outras aplicações ou clientes humanos. Para trazer um panorama geral de como é possível lidar com isso, serão expostas algumas estratégias para realizar a segurança entre aplicações, com foco na estratégia de autenticação BASIC.

Primeiramente, vale ressaltar que é totalmente impensável disponibilizar um microserviço em produção sem nenhum recurso de segurança. Se não é posta nenhuma camada de proteção, todas as operações disponibilizadas pelos serviços ficarão suscetíveis a acessos indevidos, o que seria extremamente crítico, visto que a ideia é que esses serviços possam manipular informações que dizem respeito ao cerne de um modelo de negócios de uma instituição.

No modelo de autenticação tradicional (entre usuário e sistema), utilizamos o conceito de *principal* para descrever qualquer coisa que possa se autenticar e estar autorizada a realizar operações, mas normalmente isso envolvia pessoas usando computadores. O que dizer então de programas ou serviços se autenticando um em relação ao outro?



## Restringindo um perímetro de acesso na rede

A opção com o menor impacto em nível de aplicação seria configurar restrições na rede para bloquear conexões de fontes não autorizadas. Por exemplo, se dois microserviços se comunicam em uma rede local, esse canal de comunicação pode ser bloqueado para acesso externos, permitindo a interação somente no perímetro da rede local. Como uma aplicação não age como um usuário, que pode utilizar várias máquinas ou redes para se conectar, fica mais fácil restringir o *host* de origem. No entanto, uma vez que um invasor tenha conseguido penetrar naquela rede, ele terá permissão total para realizar qualquer operação dentro da aplicação, podendo inclusive interceptar e modificar pacotes durante operações (ataque *man in the middle*), o que pode levar a graves inconsistências nos serviços.

Logo, mesmo que os serviços rodeem em HTTPS (o que é essencial), recomenda-se que haja mais alguma estratégia de autenticação nas bordas das aplicações, e não somente na rede, visando evitar confiar em algo que a aplicação não tem controle.

### Nota

A palavra-chave "Basic", utilizada no cabeçalho Authorization, deve ser escrita com a primeira letra maiúscula e as demais minúsculas. É preciso tomar esse cuidado visto que, caso seja escrito "basic" ou "BASIC", a comunicação não funcionará corretamente.

### Nota

Observe que neste exemplo estamos usando a estrutura de arquivo de configuração no padrão YML, mais recomendável devido à maior facilidade de leitura, através da criação de hierarquias. Outra opção seria utilizar o formato .properties tradicional. Deste modo, os atributos de configuração seriam configurados da seguinte forma: "security.user.name=admin", "security.basic.enabled=true" e assim por diante.

## Autenticação HTTP(S) Basic

A autenticação HTTP(S) Basic permite que um cliente envie credenciais (usuário e senha) num cabeçalho HTTP padrão. Isso será recebido pelo servidor, que irá confirmar se o cliente em questão possui acesso à operação solicitada. A vantagem desta abordagem é que este é um protocolo padrão, de fácil compreensão e com amplo suporte. O cuidado que deve ser tomado é utilizar HTTPS em toda comunicação, tendo em vista que se for utilizado HTTP, as credenciais serão trafegadas sem criptografia, o que consequentemente facilitará a captura desses dados no meio de uma operação e a sua utilização indevida em requisições posteriores.

Quando é utilizado o protocolo HTTPS, o cliente tem uma forte garantia de que o servidor que está respondendo é mesmo aquele que o cliente deseja se comunicar. O que dá uma proteção maior contra ataques que venham a mascarar endereços através do redirecionamento de DNS, visto que o endereço forjado não teria um certificado confiável que valide que aquele endereço IP corresponde ao DNS acessado.

No entanto, uma desvantagem neste caso é que cada servidor precisará gerenciar seu próprio certificado. Isso pode ser bastante problemático caso cada serviço rode em uma máquina separada, tendo em vista que cada uma terá de ter seu próprio, e logicamente, para que o certificado seja validado por uma entidade certificadora, há custos envolvidos, dependendo do tipo de certificado. Outra desvantagem é que, com HTTPS, não é possível realizar cache através de proxies reversos como Squid ou Varnish. Isso significa que será preciso implementar soluções de cache no servidor ou até mesmo no cliente, caso a aplicação tenha de lidar com grande número de acessos.

Apesar das desvantagens, no cenário de microserviços, utilizar HTTPS não é mais uma opção e sim praticamente uma obrigação, devido à grande demanda de requisições HTTP. Considerando a autenticação BASIC, a seguir será ilustrado um passo a passo de como realizar a implementação utilizando os recursos do Spring Boot aliados ao Spring Security, um módulo do Spring Framework responsável só por segurança.

O funcionamento da autenticação BASIC é bem simples: cada aplicação (microserviço) terá um nome de usuário e uma senha padrão. Assim, só serão aceitas requisições de outros serviços que tenham em seu cabeçalho de autorização (*Authorization*) as credenciais corretas para o acesso a aquele serviço.

Dito isso, o primeiro passo é configurar, na aplicação que irá prover o serviço, a restrição para que o acesso somente seja realizado mediante o fornecimento do *login* e da senha. Para fazer isso em um projeto utilizando o Spring Boot, basta importar no *pom.xml* a dependência do Spring Security (vide [Listagem 3](#)) e adicionar as credenciais de acesso ao arquivo de propriedades da aplicação, conforme a [Listagem 4](#).

Com isso, ao rodar o seu projeto e tentar acessar pelo navegador, você irá se deparar com uma caixa de autenticação solicitando que sejam preenchidos o *login* e a senha para obter acesso. Uma vez que você preencheu esses dados (conforme o que foi configurado



no arquivo *application.yml*), o acesso será obtido normalmente. O problema aqui é que estamos pensando na autenticação entre aplicações, e a aplicação não irá digitar essa senha através da interface do navegador.

**Listagem 3.** Adicionando a dependência do Spring Security compatível com o Spring Boot

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

**Listagem 4.** Adicionando as credenciais no arquivo "application.yml"

```
security:
  user:
    name: admin
    password: 78fa095c-3f4c-48f1-ad50-e23c31d5df35
  basic:
    enabled: true
```

Para que a aplicação possa se autenticar, será necessário enviar as credenciais dentro do cabeçalho "*Authorization*" do HTTP. Caso a aplicação cliente rode em cima da plataforma Java EE, pode-se utilizar a API Jersey para realizar o consumo do serviço. Neste caso, a adição do cabeçalho de autenticação à requisição HTTP pode ser feita através da criação de uma classe que implemente a interface **ClientRequestFilter** (vide **Listagem 5**) e registrando-a junto à chamada para o serviço (por meio do método **register()**), conforme mostra o exemplo da **Listagem 6**.

Note que o cabeçalho *Authorization* é preenchido no formato *Basic usuário:senha*, encriptado usando Base64. Apesar dos dados estarem encriptados seguindo esse padrão, isso não dispensa o uso de HTTPS, visto que o Base64 é um algoritmo de via dupla, ou seja, é possível, sem muito esforço, obter o valor original do código cifrado.

Uma vez que o filtro de requisição do cliente foi criado, é preciso adicioná-lo à requisição que você deseja fazer ao serviço, através do método **register()**, como mostra a **Listagem 6**.

O exemplo dessa listagem ilustra o acesso proveniente de uma aplicação Java. Outra possibilidade de comunicação ao serviço é o acesso direto a partir do navegador, via JavaScript.

Neste caso, é preciso adicionar os cabeçalhos diretamente no objeto **XMLHttpRequest**, através do método **setRequestHeader()**, como mostra a **Listagem 7**.

O diferencial em relação a uma requisição padrão que utiliza **XMLHttpRequest** é a presença do cabeçalho *Authorization*, definido através do método **setRequestHeader()**.

Para evitar o tráfego de mensagens planas contendo nomes de usuário e senha, é necessário codificar as credenciais de acesso passadas neste cabeçalho em Base64. Para fazer isso no cliente JavaScript, pode-se utilizar o método **window.btoa()**, passando como entrada uma **String** contendo as credenciais no formato *usuário:senha*.

**Listagem 5.** Filtro de autenticação em um cliente usando a API Jersey para o consumo de web services.

```
@Provider
public class ClientAuthenticator implements ClientRequestFilter {

  private final String user;
  private final String password;

  public ClientAuthenticator(String user, String password) {
    this.user = user;
    this.password = password;
  }

  public void filter(ClientRequestContext requestContext) throws IOException {
    MultivaluedMap<String, Object> headers = requestContext.getHeaders();
    final String basicAuthentication = getBasicAuthentication();
    headers.add("Authorization", basicAuthentication);
  }

  private String getBasicAuthentication() {
    String token = this.user + ":" + this.password;
    try {
      return "Basic " + DatatypeConverter.printBase64Binary(
        token.getBytes("UTF-8"));
    } catch (UnsupportedEncodingException ex) {
      throw new IllegalStateException("Cannot encode with UTF-8", ex);
    }
  }
}
```

**Listagem 6.** Adicionando o filtro no cliente.

```
Client client = ClientBuilder.newClient();
String usuario = "admin";
String senha = "78fa095c-3f4c-48f1-ad50-e23c31d5df35";
Response response = client.target("servicePath")
  .register(new ClientAuthenticator(usuario, senha))
  .request().get();
```

**Listagem 7.** Adicionando credenciais no cabeçalho do XMLHttpRequest (JavaScript).

```
var xhr = new XMLHttpRequest();
var usuario = "admin";
var senha = "78fa095c-3f4c-48f1-ad50-e23c31d5df35"
var credenciais = "Basic " + window.btoa((usuario+':'+senha));
xhr.setRequestHeader("Authorization", make_base_auth(username, password));
```

Caso prefira utilizar jQuery, a configuração torna-se um pouco mais simples. Basta passar os parâmetros *username* e *password* na chamada do método **\$.ajax()**, como mostra a **Listagem 8**.

Como estamos lidando com microserviços, a ideia é que o cliente (aplicação Java ou JavaScript) funcione em um projeto separado do que provê o serviço. Ou seja, cada um rodará em *hosts* ou pelo menos em portas diferentes. Neste cenário, para obter acesso a partir do cliente, é preciso ativar o parâmetro **crossDomain** (visível na **Listagem 8**). Assim, para que a chamada funcione corretamente,

teremos de realizar a configuração correta do padrão CORS para a comunicação entre serviços, o que exige um ajuste a mais no servidor. A seguir será descrito em detalhes o padrão CORS e como é possível implementá-lo.

**Listagem 8.** Registrando credenciais na chamada via jQuery (JavaScript).

```
$ajax({  
    url:"serviceURL",  
    type:"METHOD",  
    dataType:"json",  
    username: usuario,  
    password: senha,  
    crossDomain: true  
});
```

## Realizando requisições cross-origin

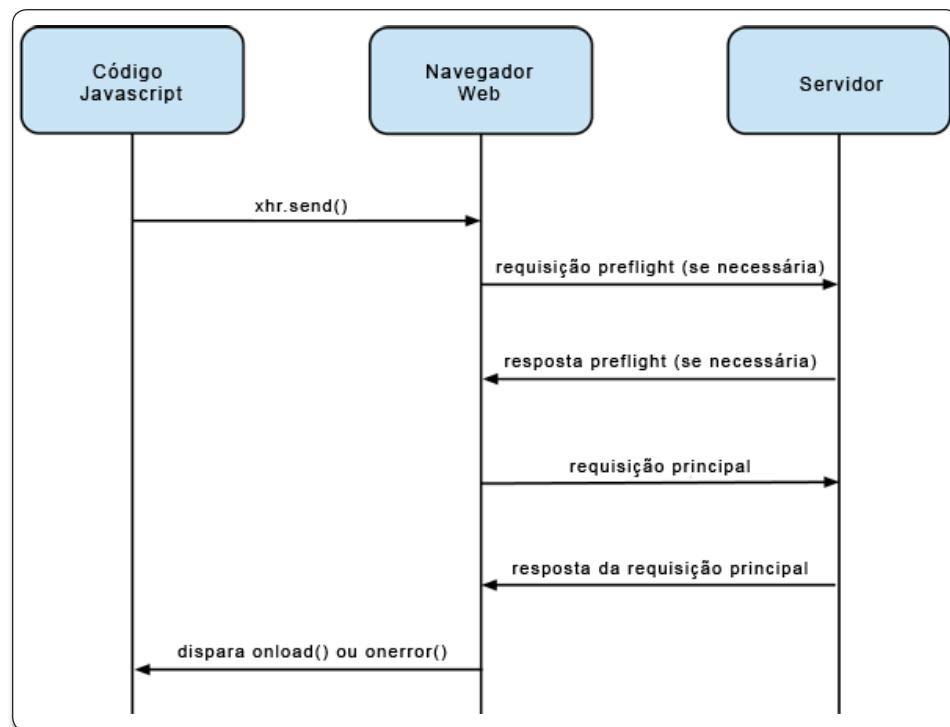
Uma requisição *cross-origin* ocorre quando é requisitado um recurso de um domínio diferente da sua origem. Por exemplo, é o caso quando um serviço que roda no endereço `http://dominioa.com` faz uma requisição a um serviço em um domínio `http://dominioB.com`. Esse conceito se aplica a qualquer tipo de requisição, seja para o consumo de dados ou para a recuperação de arquivos estáticos.

Por razões de segurança, os navegadores restringem, por padrão, requisições *cross-origin* originadas dentro de scripts, utilizados para o fornecimento/consumo de dados. Por exemplo, o objeto `XMLHttpRequest` segue uma política que considera que as requisições sempre serão

realizadas dentro do mesmo domínio (*same-origin policy*). Logo, uma aplicação que utiliza `XMLHttpRequest` só poderá realizar requisições HTTP no seu próprio domínio. Para superar essa limitação, o *Web Applications Working Group* da W3C passou a recomendar a utilização do novo mecanismo *Cross-Origin Resource Sharing* (CORS). O CORS possibilita que servidores Web tenham total controle do acesso proveniente de domínios diferentes. Assim, do lado do servidor, será necessário implementar algum mecanismo que intercepte as requisições e defina que tipo de acesso será permitido a partir de um domínio diferente. Já do lado do cliente, os navegadores mais modernos preenchem automaticamente as informações requeridas pelo padrão CORS quando realizam chamadas à própria API `XMLHttpRequest`, dispensando deste modo quaisquer adaptações em relação ao que já era utilizado em requisições com autenticação BASIC dentro de um mesmo domínio.

## Como o CORS funciona

O padrão CORS funciona através da adição de novos cabeçalhos HTTP que permitem que os servidores descrevam quais clientes podem ler informações a partir de um navegador web. No entanto, a adição dessas informações à requisição poderia trazer efeitos colaterais, alterando de alguma maneira o que foi pedido originalmente. Por isso, a especificação exige que os navegadores realizem um processo denominado “preflight”, que seria uma troca de informações antes da requisição principal, realizada através de uma requisição HTTP usando o método `OPTIONS`. Assim, só após a aprovação da conexão daquele cliente com o servidor (ocorrida durante a etapa de preflight) que a requisição principal é enviada. A **Figura 1** descreve o fluxo seguindo o padrão CORS, desde a chamada inicial via método `send()` da API `XMLHttpRequest` (xhr), até o retorno do servidor ao cliente.



**Figura 1.** Fluxo do padrão CORS

## Configurando o CORS no Servidor

Para a configuração do padrão CORS do lado do servidor é preciso criar algum mecanismo que intercepte as requisições e adicione os cabeçalhos necessários. Além disso, o servidor precisa identificar quando está lidando com a requisição *preflight* ou com a requisição principal.

Assim, é necessário criar um *HTTP Filter*, que iremos chamar de **CORSFilter**, como mostra a **Listagem 9**. Em seu código, o primeiro passo é verificar se o cliente (neste caso, o navegador web) informou corretamente o cabeçalho “Origin”, o que é feito automaticamente quando são utilizadas requisições HTTP via `XMLHttpRequest`.

Logo após, devem ser determinados valores para alguns cabeçalhos complementares, descritos em detalhes a seguir:

- **Access-Control-Allow-Credentials:** o valor **true** indica que será aceito o fornecimento de credenciais nas requisições como, por exemplo, o cabeçalho “*Authorization*”, utilizado pela autenticação BASIC;
- **Access-Control-Allow-Origin:** descreve quais origens (clientes) terão acesso ao servidor. O campo deve conter o endereço do(s) cliente(s) ou um coringa ‘\*’. No entanto, se for usado o coringa, não será permitida a utilização de credenciais. Portanto, a indicação é que sejam informados os hosts dos clientes permitidos. Caso queira liberar o acesso a qualquer servidor de origem, a estratégia que pode ser utilizada em alternativa ao coringa é ler o cabeçalho “Origin” da requisição e escrever ele nesse cabeçalho, como é feito no exemplo da **Listagem 9**;
- **Access-Control-Allow-Methods:** descreve quais métodos HTTP o servidor irá aceitar na requisição principal. No exemplo da **Listagem 9** adicionaremos os métodos POST, GET, PUT e DELETE. O método OPTIONS, que é usado nas requisições *preflight*, não precisa estar listado aqui;
- **Access-Control-Allow-Headers:** indica quais cabeçalhos HTTP serão aceitos pelo servidor. No exemplo da **Listagem 9** serão listados os cabeçalhos *Origin*, que será necessário para identificar a informação do servidor que faz a requisição; o cabeçalho *Content-Type*, que pode ser útil para a identificação

**Listagem 9.** Criando o filtro que irá possibilitar a comunicação seguindo o padrão CORS.

```
@Component
@Order(Ordered.HIGHEST_PRECEDENCE)
public class CORSFilter implements Filter {
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest response = (HttpServletRequest) res;
        HttpServletRequest request = (HttpServletRequest) req;
        if (request.getHeader("Origin") != null) {
            response.setHeader("Access-Control-Allow-Credentials", "true");
            response.setHeader("Access-Control-Allow-Origin", request.
                getHeader("Origin"));
            response.setHeader("Access-Control-Allow-Methods", "POST, GET, PUT,
                DELETE");
            response.setHeader("Access-Control-Allow-Headers", "Origin, Content-Type,
                Accept, Authorization");
        }
        if (request.getMethod().equals("OPTIONS")) {
            response.getWriter().print("OK");
            response.getWriter().flush();
            return;
        }
        chain.doFilter(req, res);
    }

    public void init(FilterConfig filterConfig) {}

    public void destroy() {}
}
```

do tipo de conteúdo trafegado; o cabeçalho *Accept*, no qual o cliente indica qual tipo de dado espera na resposta; e, por fim, o cabeçalho *Authorization*, utilizado para os mecanismos de Autenticação/Autorização.

Após as configurações dos cabeçalhos é preciso identificar e tratar de maneira diferente uma requisição preflight. Deste modo, deve ser verificado quando o método da requisição é OPTIONS e neste caso, retornar uma resposta OK (código 200) ao cliente, sem checar nenhuma credencial de acesso, evitando inclusive que o método *doFilter()* do filtro seja alcançado. Esse é um detalhe extremamente importante, tendo em vista que sem essa checagem o método OPTIONS cairia na validação do controle de acesso da aplicação, e como o método OPTIONS enviado no preflight não vai acompanhado de nenhuma credencial, sempre seria retornado um código 401 (não autorizado), inviabilizando a continuação do processo, ou seja, o envio da requisição principal.

Uma vez que o filtro está ativo, o servidor estará apto a receber requisições do cliente JavaScript de diferentes servidores. Assim, torna-se possível o acesso seguro a microserviços tanto a partir de clientes baseados em Java, como de clientes JavaScript, acessados diretamente pelo navegador.

Este artigo apresentou como adicionar uma camada de autenticação BASIC a microserviços utilizando Spring Boot e como realizar o acesso a partir de clientes Java, através da API Jersey e clientes JavaScript, utilizando a API XMLHttpRequest diretamente ou via jQuery. Ressalta-se que há outras maneiras para realizar a autenticação entre aplicações, como a utilização de SAML, OpenID Connect, Tokens de API, entre outros. No entanto, como a autenticação BASIC é o padrão para



autenticação utilizando HTTP, além de ser bastante simples de compreender e implementar, ela foi escolhida para ilustrar a adição de autenticação em microserviços.

Apesar de importante, a autenticação é apenas o primeiro passo para a criação de microserviços seguros. Além disso, é preciso adotar mecanismos de autorização confiáveis como, por exemplo, o OAuth2, assim como se preocupar com outros aspectos de segurança, que vão desde o tratamento de qualquer entrada fornecida pelo usuário, visando evitar ataques de *SQL Injection*, até mesmo o monitoramento contínuo da infraestrutura que dá acesso aos serviços, certificando-se de que o sistema operacional e a plataforma de desenvolvimento estejam devidamente atualizados.

## Autor



**Diego Ernesto Rosa Pessoa**

[diegopessoa12@gmail.com](mailto:diegopessoa12@gmail.com)



É professor do IFPB (Instituto Federal de Educação, Ciência e Tecnologia da Paraíba), doutorando em Ciência da Computação pela UFPE, mestre em Informática pela UFPB e graduado em Tecnologia em Sistemas para Internet pelo IFPB. Tem experiência de sete anos com desenvolvimento em Java, com foco em Web e Sistemas Distribuídos.

## Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)

Ajude-nos a manter a qualidade da revista!



Para obter uma visão geral da segurança para aplicações web, um bom lugar para começar é o *Open Web Application Security Project* (OWASP), que mantém e atualiza regularmente um documento contendo os 10 maiores riscos de segurança em aplicações Web. Esta é uma leitura essencial para qualquer desenvolvedor. Por fim, para ter acesso a uma discussão mais geral sobre criptografia, recomenda-se o livro “Engenharia de Criptografia”, de Niels Ferguson.

### Links:

#### **Guia de referência do Spring Boot**

[docs.spring.io/spring-boot/docs/current/reference/htmlsingle/](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/)

#### **Conjunto de boas práticas essenciais para o desenvolvimento de sistemas**

[12factor.net](https://12factor.net)

#### **Open Web Application Security Project**

[owasp.org](https://owasp.org)

#### **Detalhes sobre o padrão CORS**

[developer.mozilla.org/en-US/docs/Web/HTTP/Access\\_control\\_CORS](https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS)

#### **Codificação on-line de mensagens utilizando Base64**

[base64encode.org](https://base64encode.org)

#### **Decodificação on-line de mensagens utilizando Base64**

[base64decode.org](https://base64decode.org)

#### **Mecanismo de Autorização OAuth2**

[oauth.net/2/](https://oauth.net/2/)

# Primeiros passos no mundo da Internet das Coisas – Parte 1

Aprenda neste artigo o que é IoT e saiba como colocar seu código Java em todas as coisas e em todos os lugares

ESTE ARTIGO FAZ PARTE DE UM CURSO

**C**om tanta opção de aparelhos como interfaces para acessar aos mesmos sistemas, os laptops se tornaram equipamentos burocráticos para se conectarem à internet e aos poucos estão ganhando funções mais específicas como ferramentas de escritórios, já que esses outros dispositivos estão suprindo as necessidades de entretenimento, comunicação e coleta de informações, antes atendidas somente pelos laptops e PCs.

Gradativamente, as tecnologias vão surgindo e se mesclando em uma trilha de avanço contínuo, melhorando a experiência de interação dos usuários com os sistemas e ganhando velocidade de comunicação e processamento. E para que tudo isso se torne possível, os servidores corporativos têm acompanhado tal evolução; seja através da migração para as nuvens, ganhando flexibilidade para crescer em infraestrutura e processamentos distribuídos, seja através de arquiteturas orientadas a serviços, tornando as complexidades de negócio cada vez mais modularizadas.

Os desktops e laptops continuam e possivelmente continuarão presentes por muito tempo, sendo predominantes em escritórios de *call center*, ambientes de desenvolvimento e afins, enquanto os smartphones seguem

## Fique por dentro

Enfim a Internet chegou em todas as coisas! Hoje, qualquer objeto possui potencial de ser conectado à Internet e de gerar dados que sejam úteis para tomadas de decisão de uma pessoa ou até mesmo de outras coisas; mas do que isso, qualquer desenvolvedor é um criador com potencial de lançar novos dispositivos com capacidades sensoriais e que permitam diferentes experiências da mesma Internet que sempre esteve presente.

Neste artigo o leitor inicia um mergulho em uma Internet nova e já palpável: a Internet de vários dispositivos interconectados em um ambiente totalmente informatizado e sensorial. Nesta trilha rumo à exploração de novas formas de interação entre humanos e máquinas, veremos que o Java tem todo o potencial para continuar sendo a linguagem principal de controle desses dispositivos. Portanto, prepare seus sensores!

abrindo o caminho para uma interconexão sem precedentes na história da comunicação.

Agora está chegando a era em que várias coisas pessoais conectam não só pessoas a pessoas, mas principalmente, coisas a coisas, tudo isso através de uma internet descentralizada e composta por dispositivos capazes de intercomunicação de forma automática. Mesmo uma adega de vinho, uma geladeira, um relógio ou o sistema elétrico de uma casa; coisas impessoais. A estimativa para os próximos anos é de que aproximadamente 50 bilhões de dispositivos estarão conectados à grande rede. Isto representa um grande desafio de computação em nuvem, infraestrutura de armazenamento e processamento de eventos complexos em

uma arquitetura mais elaborada e descentralizada. Este será, certamente, o grande desafio dos servidores corporativos e até mesmo pessoais (leiam-se caseiros). Porém, a maior evolução com a Internet das Coisas provém do fato que hoje os sistemas embarcados têm maior capacidade de processamento. Em grande parte, esses sistemas suportam Java, conseguem rodar serviços mais elaborados, conversar com sensores e dispositivos como câmeras e microfones.

Neste cenário, o foco tenderá a ser concentrado nos dispositivos embarcados. Serão esses dispositivos que efetuarão o processamento final, em vez de coletar informações e enviá-las a um servidor que concentre a inteligência e a força de processamento, como é o modo mais tradicional de arquitetura de sistemas. Esta mudança de paradigma já representa, por si só, um grande desafio no que diz respeito à instalação do software (*deploy/delivery* contínuo) por causa da quantidade e variedade de dispositivos simultâneos a serem atualizados individualmente com novas versões. Nestas condições, o Java já começa a empreitada levando a vantagem de permitir atualização dinâmica via rede; aliás, vale lembrar que o Java já nasceu com este intuito: o de ser a linguagem de programação de diversas coisas, flexível e potente o suficiente para caber desde em um chip de cartão de crédito, até em um servidor com arquiteturas complexas de processamento.

Pensando neste mercado que está se ampliando, a Oracle se antecipou e preparou mecanismos para que o desenvolvedor Java que hoje se concentra em aplicações comerciais e está acostumado com o desenvolvimento orientado a servidores como JBoss, GlassFish, Tomcat, Jetty, entre outros, possa continuar concentrado nestas ferramentas sem precisar mudar de paradigma ou se preocupar com muitas mudanças no desenvolvimento para plataformas embarcadas. O Java Embedded Suite, assunto explorado na Java Magazine 118, foi criado com a finalidade de suprir estas novas demandas. Através dele, é possível embarcar servidores completos como GlassFish,

por exemplo, e utilizá-los como gateway de comunicação com outros dispositivos para enviar o estado de algum sensor ou receber comandos para executar alguma ação como ligar um sensor de Bluetooth, ler o valor de um sensor de umidade ou temperatura, tudo remotamente.

Em um dispositivo de arquitetura ARM, como um Raspberry PI, que é um computador bastante compacto e suficientemente barato, é possível configurar facilmente um servidor e ainda utilizar a sua saída de vídeo HDMI para trabalhar com interface gráfica em um ambiente Linux que hoje possui várias distribuições desenvolvidas para a sua arquitetura. Ele ainda conta com conectores Ethernet e USB, além de GPIO para conexão com sensores, o que permite conectar diversos dispositivos em ideias inovadoras. O Raspberry PI é considerado a tecnologia da vez nesta tendência de Internet das Coisas por apresentar-se já suficientemente completo, barato e compacto para rodar aplicações de processamento mais pesado; podendo ainda ser embarcado em uma solução de automação mais complexa.

Muito se tem falado sobre o termo IoT (*Internet of Things*), geralmente associado a automação de casas e criação de mecanismos inteligentes em que geladeiras gerenciam a si próprias e às suas necessidades de reabastecimento; banheiras que se enchem sozinhas quando o dono da casa está para chegar de um dia cansativo de trabalho; entre outros cenários. Mas o que faz com que tudo se conecte? Quando tudo isto se tornará realidade e começará a funcionar como parte do cotidiano? Como o Java se encaixa nesse novo mundo? Como o desenvolvedor deve se preparar para estes novos desafios? Neste artigo estas questões, entre outras, serão não somente respondidas, mas também exploradas, através da prototipação de um pequeno dispositivo capaz de medir a altura de uma pessoa e enviar essa informação a um Tablet, que poderá ficar na mesa de centro da sala de estar para apresentar ao visitante a sua altura, quando ele passar por baixo de um sensor ao atravessar a porta de entrada da casa.

## Nota

Sistemas embarcados são sistemas computacionais embutidos em um dispositivo completo, cujas funções dedicadas residem juntas a um sistema eletrônico ou mecânico maior. Tipicamente utilizados tanto em aplicações industriais como em aplicações de cliente final, os sistemas embarcados estão incorporando a linha de Internet das Coisas (IoT).

## O que é a Internet das Coisas?

Não existe uma definição que explique de maneira geral este termo, principalmente pelo fato de que a Internet das Coisas representa uma ampliação de praticamente toda a ciência da computação como uma evolução natural do modo como as pessoas, de usuários finais a engenheiros, arquitetos e desenvolvedores, enxergam a tecnologia e seus novos meios de proporcionar outras experiências de usabilidade. Essa evolução implica em diferenças na criação de novos produtos



desde a arquitetura de novos dispositivos até o desenvolvimento de softwares e serviços, arquiteturas e infraestruturas de processamento. Mas existe uma pergunta cuja resposta determina se algo faz parte da Internet das Coisas: um produto de um determinado fabricante pode se conectar aos produtos de outros fabricantes de forma automática? Um leitor de RFID (vide **BOX 1**) de um fabricante poderia, por exemplo, se conectar ao sistema de um portão eletrônico de outro fabricante para abri-lo ou fechá-lo através de um protocolo padrão?

#### BOX 1. Sistemas RFID (Radio Frequency Identification)

RFID ou identificação por radiofrequência é um termo genérico para as tecnologias que utilizam frequência de rádio para captura de dados. A forma mais tradicional desta tecnologia se dá através da leitura de etiquetas – mais conhecidas pelo nome em inglês (*Tags*) – que possuem códigos utilizados para identificar coisas, artigos de produtos, dispositivos ou pessoas.

Na Internet das Coisas, os leitores RFID, junto às etiquetas, apresentam-se como solução eficaz e barata para identificação de objetos ou ainda pessoas em soluções que exigem ações personalizadas. Um exemplo é o reconhecimento de moradores na automação de uma casa. Qual quarto deve ter sua luz acesa quando determinado morador chegar? Que tipo de música tocar no aparelho de som? Para qual temperatura o ar-condicionado deve ser regulado? Todos estes parâmetros podem ser personalizados em um sistema orientado ao reconhecimento através de um cartão ou etiqueta via leitor RFID.

Para exemplificar esta premissa de intercomunicação entre dispositivos para que estejam aptos à Internet das Coisas, podemos considerar o seguinte cenário: Um morador se aproxima do portão de entrada de sua casa e então uma etiqueta RFID instalada em seu carro é detectada pelo sistema residencial, que desbloqueia o portão e ainda é capaz de identificar quem é o morador que está chegando. O dispositivo wireless do portão envia uma mensagem via rede e dá o comando para que o quarto de solteiro tenha a luz ligada e o televisor é sintonizado em seu canal de esportes predileto, uma vez que o morador detectado é o filho adolescente de dezoito anos de idade. Ainda neste cenário, ao reconhecer o adolescente, o sistema regula o ar condicionado do quarto, que teve sua potência reduzida enquanto o adolescente saía para estudar, para ficar mais confortável, na temperatura pré-estabelecida pelo garoto. Como é possível verificar, tudo está funcionando de forma coordenada, pois esta é a grande premissa da Internet das Coisas: coordenação entre múltiplas coisas (sem a intervenção humana).

### Por que a Internet das Coisas é o futuro?

Estamos em uma era em que as crianças já nascem com acesso às mais avançadas tecnologias e estão desde muito cedo conectadas ao mundo, muitas vezes através dos próprios pais, que querem compartilhar o crescimento e os momentos dos filhos com toda a família e amigos. As pessoas querem e fazem questão de estarem conectadas umas às outras, noticiando os mais diversos tipos de atividades cotidianas: uma corrida em um parque, uma cena engraçada do bichinho de estimação, um momento importante no trabalho, como estão ganhando

peso, como estão perdendo peso, como estão evoluindo em uma atividade de estudo, na carreira, na escola. Diante disso, a frase “The internet is everywhere”, vinda do inglês e traduzida livremente para “A internet está em todos os lugares”, nunca se fez tão latente.

Em um cenário como este, com tantos dispositivos se conectando e conectando pessoas, com tanta informação sendo compartilhada, os dispositivos ganham uma extensão essencial para que essa onda de conexão entre pessoas se torne possível e para que seja possível coletar tudo sobre qualquer coisa: os sensores. Eles podem registrar mudanças de temperatura, claridade, pressão, som, movimento, entre outros. Se entrarmos no campo dos smartphones, então esta lista fica ainda mais interessante: sensores de batimentos cardíacos, velocidade, localização (GPS), vibração, etc. Esses sensores são os olhos e orelhas dos dispositivos para o que acontece no mundo e permitem que máquinas ganhem sensibilidades que humanos possuem ou, em algumas ocasiões, sensibilidades que nem mesmo humanos possuem, como a detecção de presença de álcool.

Em uma rede em que todos compartilham quase tudo sobre si próprio, aplicativos e dispositivos que necessitem de entradas diretas, ou seja, que precisem de uma tecla digitada ou de um botão pressionado começam a se tornar alternativas menos utilizadas, já que a tendência é que os dígitos e cliques sejam substituídos por movimentos, momentos, cheiros, sabores, substâncias, ondas, luzes e vozes detectados por sensores, sendo estas alternativas mais naturais e sucintas, exigindo menor esforço e concentração do usuário para que a interação ocorra com a mesma ou melhor eficiência.

Antes de falarmos sobre o que os sensores fazem, vamos entender sua eletrônica. Os sensores são parte de uma categoria de dispositivos chamados de sistemas Micro-eletro-mecânicos (sim, esta palavra existe em Português e é abreviada como MEMS em inglês: *Micro-Electro-Mechanical Systems*) e são manufaturados de forma muito semelhante à produção dos microprocessadores: através de um processo de litografia, possibilitando a concepção de micro sistemas fáceis de serem embarcados nos circuitos integrados de um microcontrolador como um Arduino ou em um microprocessador como um Raspberry PI através dos conectores GPIO ou via wireless (por meio de frequência de rádio) para detectar informações do ambiente.

### Na prática, como os sensores funcionam?

O funcionamento dos sensores pode ser compreendido através da análise da seguinte situação: um desenvolvedor de sistemas um pouco sedentário resolve entrar em forma andando de patins. Assim, ele resolve instalar sensores de velocidade e distância percorrida em seus patins, além de um dispositivo capaz de enviar essas informações via Bluetooth para outros aparelhos. O nosso novo esportista também adquire uma balança que, além de medir o peso, é capaz de medir a altura e enviar, também via Bluetooth, essas informações, assim como o IMC (Índice de Massa Corpórea), a outros dispositivos. O nosso desenvolvedor

# Primeiros passos no mundo da Internet das Coisas – Parte 1

então tem a ideia de unificar todas essas informações da balança e dos patins em um aplicativo de celular que receberá automaticamente o peso aferido. O aparelho, por sua vez, envia esses dados a um mini servidor Raspberry PI que executa uma aplicação Java Embedded em um web container Tomcat que ele instalou na rede local de sua casa. Com estas informações já sincronizadas com o servidor, ele sai para patinar e ao término emparelha o seu celular com os patins para coletar a velocidade média, tempo e distância percorrida. Ao chegar em casa, o aplicativo envia essas novas informações ao servidor Raspberry PI, que utilizará essas novas entradas na geração de dados estatísticos sobre a evolução dos treinos, perda de peso, entre outras informações importantes para o nosso novo atleta.

## Nota

Sistema Micro-eletro-mecânico (Micro-Electro-Mechanical System, em inglês) é o nome dado para a tecnologia que integra elementos mecânicos, sensores, atuadores e eletrônicos em um pequeno chip que possui instruções de funcionamento inalterável em um firmware. São praticamente micromáquinas programadas para cumprir determinada atividade.

## Nota

Na engenharia computacional, litografia é o processo pelo qual se criam os microprocessadores. Neste processo, um líquido foto resistente (ou seja, cuja resistência varia conforme a intensidade da luz) é aplicado sobre um disco de silício em rotação, fazendo com que o líquido seja espalhado uniformemente na placa. Em seguida, através da projeção de luz Ultravioleta sobre o disco, o líquido reage com a intensidade da luz e forma o desenho dos circuitos em tamanho microscópico. Todos os pontos atingidos pela luz tornam-se solúveis e, portanto, removíveis. Em seguida, íons de cobre são utilizados para recobrir a área removida e o processo se repete formando as várias camadas que criam o circuito final do chip. Esta etapa de injeção do líquido foto resistente e da sua “escrita” na placa através da luz Ultravioleta é o que chamamos de litografia.

Após todo esse exercício, o desenvolvedor então decide ir até a geladeira para comer um lanche, geladeira essa que possui um display como atuador conectado ao servidor Raspberry PI através da rede local. Ao abrir a porta, o dispositivo é acionado e exibe uma sugestão de quantas calorias são recomendadas que ele consuma para repor a energia gasta durante a patinação e sugere até qual alimento consumir com base em seus hábitos alimentares. Para isso, manualmente, o nosso usuário também configurou o servidor com uma relação de quais produtos estão na geladeira junto às suas informações nutricionais, permitindo, assim, que o sistema consiga verificar a melhor alimentação para o momento, de acordo com a evolução física e nutricional.

Portanto, na prática existem poucas coisas que não possam ser feitas com alguns sensores e um pouco de criatividade. Nessa aplicação fictícia, os dados não saíram da rede local do usuário e o celular funcionou como uma interface amigável para que o atleta pudesse coletar as informações de todos os sensores e também foi capaz de manter essas informações armazenadas até poder descarregá-las em um servidor central ainda bastante pequeno e simples, mas suficiente para o processamento na geração de estatísticas que posteriormente foram utilizadas como dados de saída para o display embutido na geladeira. O celular poderia ainda ter sido configurado para coletar e descarregar essas informações de modo automático, sem intervenção do usuário, criando assim um fluxo completo em que nenhuma intervenção humana seria necessária para que sensores em dispositivos diferentes pudessem trabalhar de forma orquestrada em função da geração dessas sugestões de consumo mostradas no display da geladeira. Esta é a Internet das Coisas em sua forma mais latente, funcionando através do uso prático de dispositivos simples, que já existem ou que podem facilmente ser adaptados, como no caso do sensor nos patins.

Note que o desenvolvedor poderia optar também por estender a funcionalidade do sistema e integrar-se a redes sociais para gabar-se de sua evolução e da sua boa forma, ou então poderia convidar outros amigos a seguir sua rotina nutricional ou a seguir uma bateria de corridas sob patins. Na prática, a criatividade e a utilidade são os únicos limites para a utilização de sensores.

## Pensando em prototipação

Um protótipo é a melhor forma de iniciar o projeto de um produto na Internet das Coisas e a sua utilização como parte desse processo de criação possui muitos benefícios, como: durante uma prototipagem o designer irá, inevitavelmente, passar por problemas que irão demandar mudanças em várias iterações até chegar a um resultado aceitável. Neste processo, a realização de testes em um experimento individual é uma abordagem trivial se comparado com a modificação de centenas ou milhares de produtos que venham a ser colocados em produção.

A construção de um produto para a Internet das Coisas envolve três atividades essenciais e paralelas: a construção da



**coisa** física, a parte eletrônica que dá a inteligência à coisa e o serviço web com o qual ela irá se comunicar. Esta última é relativamente fácil e barata de ser alterada, porém, não é tão simples modificar o objeto e seus componentes controladores, sensores e atuadores, a não ser que seja feito um recall em todos os dispositivos finais.

#### Nota

Recall é a palavra em inglês adotada com o significado de convocação por parte de um fabricante ou distribuidor para que determinado produto seja levado de volta para substituição ou reparo de possíveis ou reais defeitos.

Portanto, o protótipo deve ser otimizado para facilitar e acelerar o desenvolvimento, além de ser flexível para a alteração de seu comportamento. Tal experimento, como na maioria dos projetos de dispositivos para a Internet das Coisas, inicia-se com um microcontrolador ou microprocessador ligado a componentes através de fios interconectados por meio de uma placa de prototipagem normalmente conhecida como Protoboard, termo em inglês popularizado no Brasil, ou Breadboard, termo em inglês popularizado em outros países (vide **BOX 2**).

#### BOX 2. Matriz de contatos – Protoboard

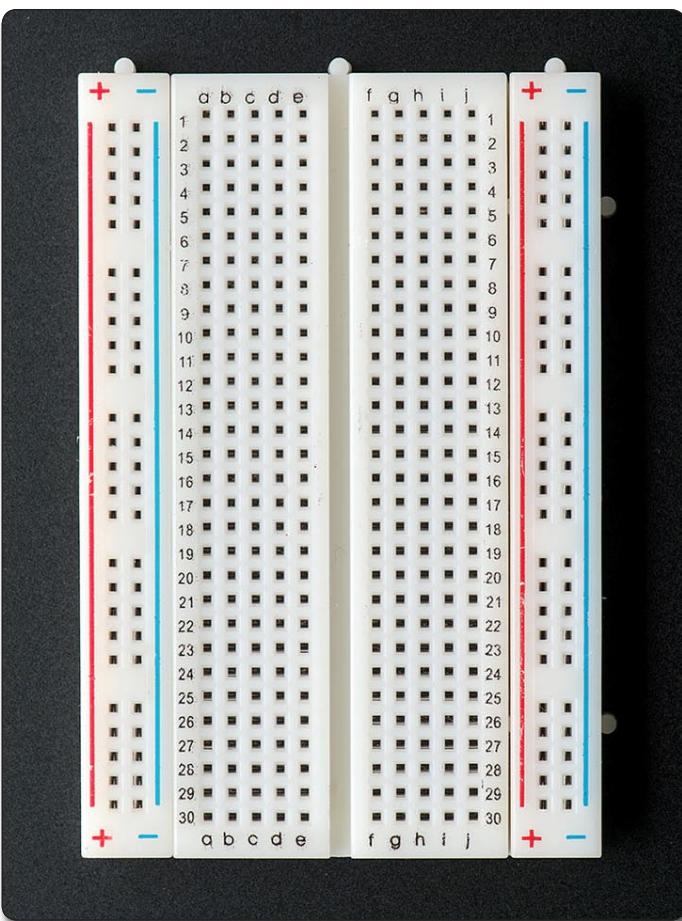
A matriz de contatos ou Protoboard é utilizada para fazer montagens provisórias em testes de projetos eletrônicos. Trata-se de uma composição simples feita a base de plástico e vários orifícios ligados uns aos outros, permitindo a interconexão serial e paralela através de circuitos criados pela conexão de fios e componentes eletrônicos, sem a necessidade de solda ou qualquer ligação rígida. A vantagem de utilizar a Protoboard é o fato de ela ser reutilizável em vários projetos, o que possibilita a montagem e desmontagem de vários modelos de circuitos eletrônicos.

Protobards podem ser encontradas em diversos tamanhos e classificadas pela quantidade de orifícios. Quanto mais furos a placa tiver, maiores e mais complexos podem ser os projetos. Independentemente do tamanho da placa e da quantidade de furos, no entanto, o funcionamento é o mesmo: existem pelo menos duas carreiras paralelas na vertical com um circuito negativo e outro positivo, sendo estes ligados a linhas horizontais, que podem ser alimentadas por fios vindos do circuito positivo. Ao conectar uma ponta de um fio na parte negativa e a outra ponta em uma das linhas perpendiculares, e outro fio com uma das pontas em uma parte positiva e a outra ponta em mais uma das linhas perpendiculares, é possível criar circuitos que unam componentes como transistores, resistores, sensores, atuadores (como LED e display), chips, entre outros componentes eletrônicos, formando um modelo de circuito que tenha alguma finalidade.

Normalmente as Protobards são conectadas a outras placas de prototipagem como um Arduino, que é uma solução também muito barata e que possui sistema GPIO, viabilizando a ligação e o controle de componentes elétricos via programação em software.

Este tipo de prototipagem é relativamente barato de ser desenvolvido e normalmente culmina em algo que pode ser demonstrado como um produto viável, mas ainda sem o acabamento de algo vendável ao consumidor final. Além disso, um protótipo com essas características possivelmente irá custar mais caro do que alguém estaria disposto a pagar em uma loja.

Na **Figura 1** é possível visualizar uma Protoboard de 400 furos, sendo 300 em paralelo e duas carreiras com 25 orifícios de cada polaridade (positivo e negativo) em ambos os lados.



**Figura 1.** Protoboard de 400 furos

Nesta Protoboard podemos perceber que existem 30 fileiras horizontais ilustradas por números de 1 (um) a 30 (trinta). Se um furo for energizado, todos os outros do mesmo segmento passarão a ter corrente elétrica. Já os orifícios das extremidades, ilustrados pelos sinais de + (mais), vermelho e - (menos), azul, são conectados em série, como sugerido pelas linhas coloridas. Da linha com furos de polaridade positiva saem fios conectando energia aos orifícios representados por números, permitindo assim a energização do segmento horizontal. Ademais, vale ressaltar que a linha vermelha determina o polo positivo e indica que o fluxo de eletricidade inicia-se ali, enquanto a azul indica o término do circuito.

Como todo circuito elétrico precisa de um ponto de partida como fornecimento de energia e de um ponto de término no qual a corrente termine, na Protoboard o fluxo elétrico pode ser iniciado por uma fonte qualquer (normalmente de cinco volts), que permanece fornecendo energia ininterruptamente, ou por um pino GPIO (vide **BOX 3**), que pode ser microcontrolado

dependendo da necessidade da aplicação, e pode ser finalizado com a conexão do segmento indicado com o sinal de negativo (-) à saída de corrente da fonte ou ao pino *Ground* (GND) da mesma placa GPIO.

#### BOX 3. General Purpose Input/Output – GPIO

General Purpose Input/Output ou I/O de propósito geral são pinos ou portas que podem ser programadas para receberem dados ou enviá-los através de pulsos elétricos em pinos configuráveis para entrada ou saída em diálogo principalmente com sensores. No Arduino ilustrado pela **Figura 2**, é possível localizar os pinos GPIO acima do símbolo da marca Arduino, indicados pela linha branca. Também nesta figura, podemos ver os pinos numerados de 0 a 13 e ainda o último pino GND (Ground), que deve ser utilizado para encerrar a corrente no sistema eletrônico.

Em um projeto de Internet das Coisas, o GPIO é imprescindível para dar ao sistema capacidades sensoriais e de exibição de dados. Na construção de novos dispositivos, a quantidade de portas para entrada e saída de dados indica qual o tipo de microcontrolador ou processador deve ser utilizado e, consequentemente, qual modelo pode ser adquirido para a prototipagem.

Como exemplo, imagine uma calculadora baseada em valores binários, ativados ou desativados por interruptores, e que mostre os resultados de cálculos em LEDs acesos ou apagados para representar um número. Em uma operação em que a calculadora some dois números em formato binário de até um byte cada, a quantidade de pinos a serem utilizados como entrada seria de no mínimo 16, para garantir a entrada dos dois números a serem somados. Além disso, mais nove portas GPIO seriam utilizadas para exibir o resultado nos LEDs em modo binário, já que a soma dos dois binários poderia retornar um número com estouro de base, obrigando o resultado a ter um binário a mais em sua composição.

Na **Figura 2** observa-se um protótipo desenhado na ferramenta computacional Fritzing (ver **BOX 4**) para ilustrar um circuito simples controlador de um LED. Nesta imagem os orifícios coloridos de verde simbolizam energia correndo pelo circuito. Note que a saída 13 da placa simbolizando um Arduino está conectando o primeiro pino de energia positiva da Protoboard com o polo também positivo do dispositivo através do fio vermelho. Neste mesmo segmento, um resistor é conectado em outro orifício, com sua outra extremidade ligada ao furo da outra divisão da placa, que passa a se energizar em todo o segmento. Ainda no mesmo circuito, observa-se a extremidade positiva do LED (identificada pela ponta ligeiramente entortada exatamente para indicar esta polaridade) sendo alimentada pela mesma linha paralela energizada pelo resistor. A finalização do circuito na Protoboard se dá pela união da extremidade negativa do LED ao furo que levará a corrente a corrente em série representada pela linha de cor azul na placa. Por fim, com a saída do fio azul conectado ao pino GND (*Ground*) do Arduino, encerra-se o circuito completo.

A partir disso, durante a execução do Arduino, se o pino 13 for configurado para enviar energia, então o LED se acenderá, caso contrário, ele permanecerá apagado, uma vez que não existirá passagem de energia para alimentá-lo. Este costuma ser o exemplo mais simples para ilustrar um projeto eletrônico com Arduino e Protoboard.

Uma vez que o pino 13 no Arduino já possui um resistor embutido, o que impede que o equipamento seja queimado em um

uso incorreto, na prática, o resistor utilizado na **Figura 2** torna-se opcional, podendo ser substituído por um fio comum apenas para continuar ligando os pontos e manter a corrente elétrica.

#### BOX 4. Fritzing

Fritzing é uma iniciativa de hardware open-source que torna a eletrônica acessível a todos. Trata-se de um projeto que, além de fornecer uma ferramenta de edição de circuitos, possui uma comunidade já bastante grande e ativa na troca de experiências e ideias no que diz respeito à criação e processamento com Arduino, promovendo um ecossistema criativo que permite aos usuários a documentação de seus protótipos, o compartilhamento com outros usuários, o ensinamento de eletrônica através de classes virtuais, além de permitir o layout e manufatura de placas de circuitos impressos (PCB ou Printed Circuit Board, em inglês) em um ciclo completo, da prototipação à produção.

A ferramenta Fritzing é ótima para que o entusiasta que ainda não possua as ferramentas necessárias (Protoboard, Arduino, Raspberry Pi, LEDs, resistores, entre outros) possa iniciar imediatamente o design de seus protótipos, mapeando toda a ideia, desde a simulação em uma Protoboard, até o desenho do circuito impresso, que posteriormente poderá ser produzido em larga escala.

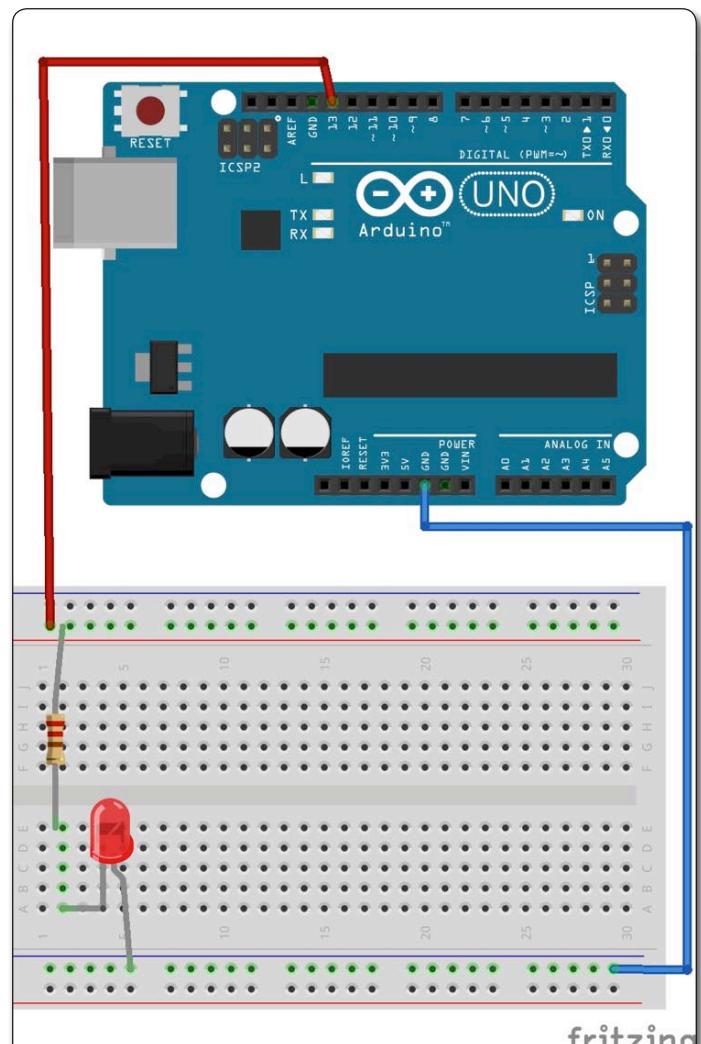


Figura 2. Design de um circuito acendedor de LED

A partir disso, o leitor mais destemido poderá realizar testes como mudar a conexão do pino 13 para o pino 12, substituir o resistor por um fio simples e ter a primeira lição sobre LEDs: sem um resistor, uma vez que LEDs geralmente exigem menos corrente elétrica que cinco volts, o LED queima.

### Sketches, para que te quero!

Em um projeto de circuitos eletrônicos para a construção do protótipo de um produto da Internet das Coisas é preciso que haja pelo menos um chip controlador com possibilidade de ser programado de forma rígida, de modo a exercer alguma ação sistemática (firmware). Tais controladores são geralmente programados em C ou em outra linguagem de programação nativa do controlador, como Assembly. O chip é essencial para que o sistema eletrônico deixe de ser estático e passe a ser dinâmico, permitindo a mudança de comportamento e tomada de decisão sem a necessidade de alteração nos componentes físicos.

O uso desse tipo de controlador através do Arduino se tornou viável em prototipações devido à sua flexibilidade de programação, seu baixo preço e facilidade de aquisição. A programação do chip no Arduino é feita na linguagem C através da criação de códigos chamados Sketches (ou, no singular, Sketch), que são basicamente instruções compiladas em um template bem definido dentro de uma API de programas para Arduino. Para a criação desses Sketches, existe uma IDE (vide **Figura 3**) desenvolvida em Java que facilita não somente a escrita desse tipo de programa, como também o seu envio para os dispositivos Arduino (ou compatíveis) de forma fácil e transparente.

Como firmwares, os Sketches são escritos para executar uma função específica e são limitados pela quantidade de memória disponível no microcontrolador. Portanto, para o desenvolvimento de programas mais elaborados, o desenvolvedor pode ter que lidar com muitas iterações de testes somente para garantir que a memória esteja sendo utilizada de forma otimizada. Além disso, pequenas modificações podem representar maiores problemas de otimização e manutenção pela complexidade inerente aos códigos escritos na linguagem C.

Este processo é bastante cansativo para o desenvolvedor que está acostumado às vantagens oferecidas pela linguagem Java, principalmente por ela ser Orientada a Objetos (e agora também Funcional) e por permitir que todo o desenvolvimento seja realizado em qualquer plataforma antes de ser implantado em um ambiente final, vantagens estas que não podem ser alcançadas através da escrita tradicional dos Sketches em C.

A fim de eliminar esta curva inicial de aprendizado para trabalhar com Sketches, o desenvolvedor Java pode contar com

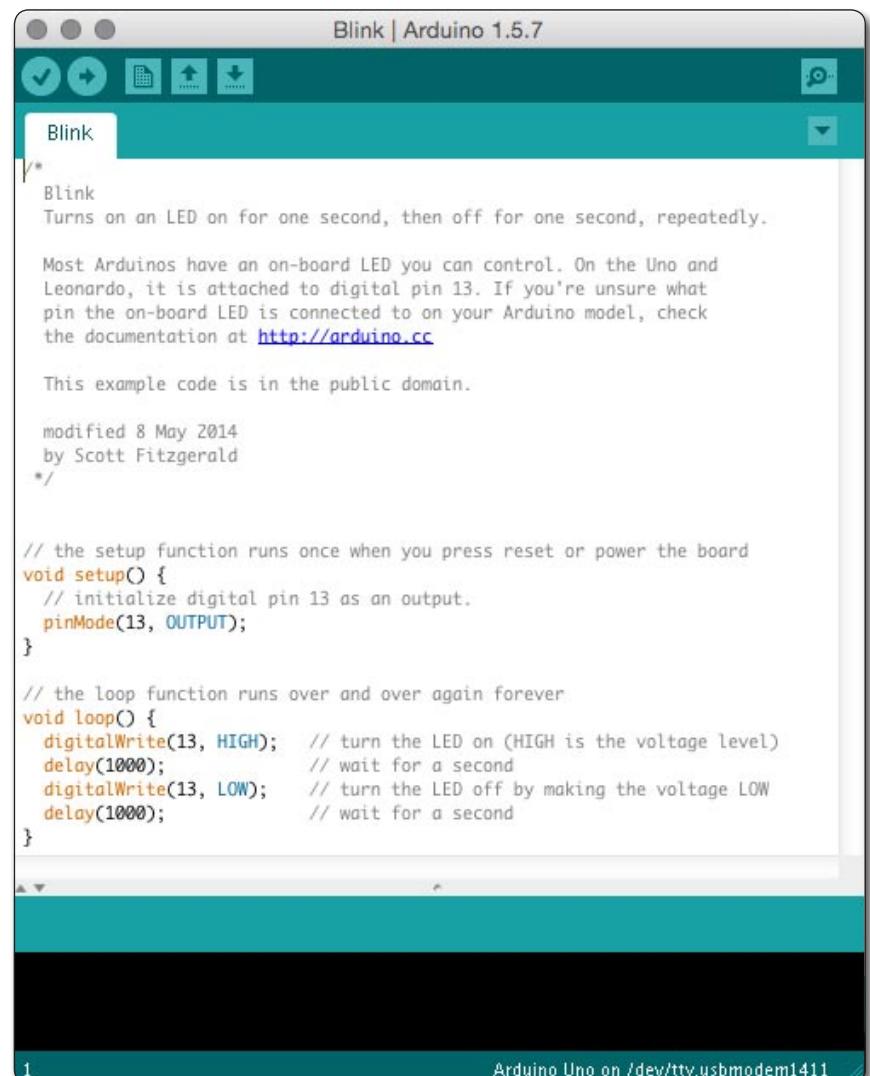


Figura 3. IDE Arduino

#### Nota

Em eletrônica e computação, firmware é o conjunto de instruções operacionais programadas diretamente no hardware de um equipamento eletrônico, armazenado permanentemente em seu circuito integrado (chip) como uma ROM, PROM, EPROM, EEPROM ou memória flash, no momento da fabricação do componente.

o projeto JArduino, disponível no GitHub. Este possibilita o controle de um Arduino via Java através da porta serial. Com isso, apenas um Sketch necessita ser instalado no controlador de uma vez por todas e todo o controle passa a ser feito via Java de forma dinâmica, com possibilidade de alteração do comportamento do microcontrolador sem necessidade de mudança de firmware. Esta abordagem se apresenta como ótima escolha para o desenvolvedor Java que quer iniciar imediatamente seus primeiros protótipos sem ter a preocupação de mergulhar nos detalhes de códigos Sketches mais complexos. Além disso, como o controle via Java necessita de um microprocessador

atuando em conjunto com o Arduino para que a comunicação ocorra, automaticamente o desenvolvedor ganha um meio mais poderoso de processamento através de uma JVM instalada no dispositivo micro processado. Nestas circunstâncias, o Raspberry PI se apresenta como uma ótima forma de integração com o Arduino para fornecer não somente capacidade e flexibilidade de processamento, mas também de comunicação com a internet e com outros dispositivos; tudo isto em um computador muito pequeno e fácil de ser embarcado.

## É preciso realmente conhecer eletrônica para criar “coisas”?

A melhor forma de responder a esta pergunta é através da seguinte analogia: suponha que exista uma oportunidade de negócios para a sua empresa e que o seu cliente tenha agendado uma reunião em seu escritório para ver a amostra de um sistema que você prometeu, contendo pesquisas, relatórios, persistência de dados, acesso via qualquer browser (*cross-browser*), envio de e-mails avisando sobre eventos importantes que tenham ocorrido, tudo em uma interface super bonita, fácil de usar e intuitiva.

Você certamente não terá tempo para lançar mão de todas as suas aptidões de arquiteto Java e de toda a sua capacidade para conceber um sistema bem distribuído, com processamento paralelo, orientado a serviços, com processamento de mensagens em background e com o que há de mais bem pensado em *design patterns* e orientação a objetos e, quem sabe, até utilizar o paradigma Funcional com Java 8. É claro que é essa a sua intenção depois de vender o projeto. Você quer fazer algo bem feito, que seja escalável e de fácil manutenção, mas para a apresentação que deverá acontecer em dois dias, o melhor que

você conseguirá é um protótipo persistindo dados na sessão do usuário em `java.util.List` ou, na melhor das hipóteses, `java.util.Map`. E isto não será um problema, pois o que o cliente quer ver em um primeiro momento é se a ideia funciona, se vai agilizar o seu negócio, se vai facilitar a sua vida e a de seus funcionários, gerando valor, seja financeiro ou de outra natureza. Ele não precisa ter certeza de que você é um Arquiteto Java naquele momento ou se você tem esse profissional em sua equipe. Essas aptidões você pode aprender depois que o projeto estiver em suas mãos, ou até mesmo esse profissional pode ser contratado.

Assim como um protótipo de sistema não requer a utilização de todas as técnicas de arquitetura e desenvolvimento, a prototipação em dispositivos físicos também não precisará das melhores práticas em termos de composição de circuitos complexos e cálculos de resistência ideal. Tais complexidades podem ser pensadas quando o projeto chegar na fase de construção para uma versão de produção que exija algo mais requintado. Sendo assim, para o desenvolvimento de um modelo funcional de uma primeira ideia é possível começar adquirindo kits completos, baratos, de montagem simplificada (através de componentes plugáveis uns aos outros), com sensores, atuadores, fios, Protoboard e Arduino. Com um kit desses em mãos, criar um protótipo de um robô com capacidade de se locomover, de utilizar uma câmera para identificar coisas, de proferir algumas palavras através de uma placa de som e que pisque um LED quando estiver “feliz” é relativamente fácil e de fato existem muitos kits já prontos para isso disponíveis no mercado.

É claro que a ideia deste artigo não é que o leitor aprenda a criar seus próprios brinquedos, mas sim que tenha um ponto de partida para a descoberta de um mundo novo, que está aí para ser explorado. Da mesma forma que dificilmente o aprendizado de JSPs se iniciará em um projeto sofisticado com PrimeFaces ou qualquer outra implementação de JSF, o caminho para se chegar a um produto acabado de um dispositivo IoT deve ter seu início em protótipos simples, mas com alguma utilidade, de forma a levar o desenvolvedor a conhecimentos mais complexos conforme ele se sentir mais familiarizado com as tecnologias disponíveis. Enfim, “baby steps” são essenciais no aprendizado de qualquer tecnologia, mesmo em eletrônica.

Baby-Steps é um termo utilizado para denominar a prática de dividir problemas complexos ou muito grandes em problemas menores, mais fáceis de serem resolvidos e compostos em uma solução maior que culmine na resolução do problema inicial. Podemos substituir facilmente este termo pela frase “dividir para conquistar”.

### Nota

Atuadores, assim como sensores, são componentes eletrônicos que se acoplam a controladores como Arduino ou Raspberry PI. A diferença está na função, pois os atuadores são componentes destinados à exibição de informações ou ao diálogo com o usuário, diferentemente dos sensores, que se destinam à obtenção de dados do mundo externo.



## Prototipação na prática

Agora, vamos modelar um dispositivo que será embutido na parede, no teto de uma casa, preferencialmente próximo à porta de entrada, para que quando chegue uma visita o aparelho calcule e exiba sua altura em um tablet localizado em cima da mesa central (quem sabe, no futuro, coloquemos uma balança no carpete da entrada e integremos as duas informações para determinar o IMC do visitante).

Como verificado, o experimento será simples: utilizaremos um sensor ultrassônico capaz de detectar objetos com precisão de dois a quatro metros de distância, dependendo do ruído. Este sensor será conectado a um Arduino e terá suas portas controladas através de um Raspberry PI. Na Figura 4 são apresentados todos os componentes utilizados no protótipo do medidor de altura, a saber:

- Quatro fios (Jumpers);
- Um sensor ultrassônico;
- Um suporte para o sensor;
- Uma pequena placa Protoboard de 170 furos;
- Um Arduino Uno R3;
- Um cabo de alimentação USB de 5V (cinco volts) para o Arduino;
- Um Raspberry PI 2 Modelo B;
- Um adaptador Wi-Fi que irá conectar o Raspberry PI à rede local;
- Um cartão SD que irá conter o sistema operacional Linux;
- Uma fonte de alimentação de 2A (dois amperes) para o Raspberry PI.

O experimento final terá a aplicação Java rodando no Raspberry PI. Todavia, durante o desenvolvimento é possível realizar todo o procedimento de testes diretamente em um laptop ou PC, sem a necessidade do Raspberry PI, pois a comunicação com o Arduino se dá via USB. Portanto, toda a implementação do código Java pode ser feita em uma IDE Java qualquer e, inclusive, incluir testes unitários que realizem a conexão diretamente com o dispositivo para verificar a funcionalidade, sendo este um dos maiores benefícios desta abordagem.

O JArduino será essencial em todas as etapas do experimento, desde a compilação do Sketch diretamente no dispositivo até a utilização da API disponibilizada como forma de comunicação com o microcontrolador na aplicação Java. Dito isso, o primeiro passo é baixar o projeto JArduino do repositório GIT, que pode ser encontrado na seção **Links** (é necessário que o projeto seja obtido deste link, pois ele possui customizações feitas especificamente para o protótipo aqui apresentado).

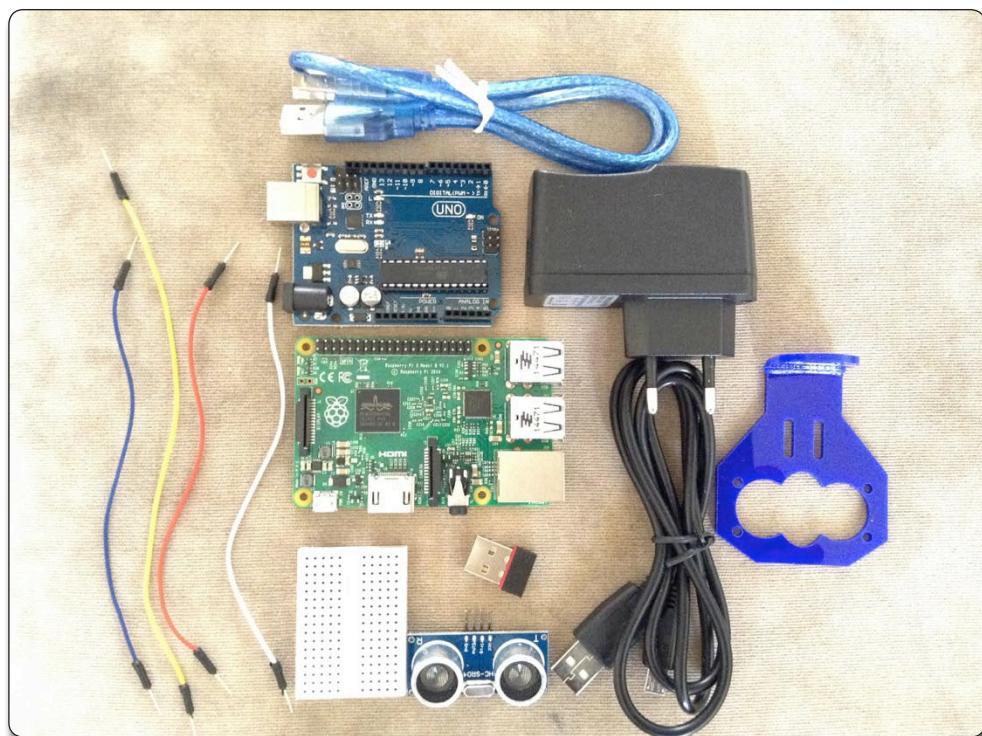


Figura 4. Componentes utilizados para criar o medidor de altura por ultrassom

Com o código do GitHub em mãos, no subprojeto *jarduino.core* é possível encontrar o diretório *src/main/arduino* e nele o projeto *JArduinoFirmware*, que deve ser aberto via IDE Arduino e posteriormente compilado e transferido ao dispositivo. Este procedimento é bem simples: com o projeto aberto na IDE, basta clicar no botão carregar – segundo ícone no canto superior esquerdo – e aguardar a compilação e transferência do programa ao chip.

Após o primeiro passo de transferência do código de firmware para o Arduino, todo o restante do processo se desenvolve na camada Java. Antes disso, no entanto, ainda no projeto baixado do GitHub, iremos encontrar vários módulos compostos em um projeto Maven que pode ser compilado para que seus artefatos finais (os arquivos JAR) sejam incorporados como bibliotecas em novos projetos. Estes componentes viabilizarão a comunicação serial com o microcontrolador e conterão a camada de comunicação com o Sketch (firmware) anteriormente carregado no dispositivo, facilitando todo o controle do Arduino via Java como se o controle fosse diretamente via firmware. A partir deste ponto é possível criar um projeto Java e incorporar a nova API recém-compilada como dependência da aplicação para comunicação com o controlador Arduino.

No exemplo deste artigo, o Maven foi utilizado para gerenciar as dependências e o build. Como é possível notar na **Listagem 1**, a declaração do projeto inclui três dependências principais da API JArduino: `org.sintef.jarduino:core`, que fornece a estrutura base de controle dos Sketches via Java (principalmente a classe abstrata `org.sintef.jarduino.JArduino`); `org.sintef.jarduino:serial`, que fornece a classe de acesso à comunicação serial com o Arduino

(através da classe `org.sintef.jarduino.comm.Serial4JArduino`); e, finalmente, a dependência `org.kevoree.extra.osgi:rxtx`, que proporciona à implementação `Serial4JArduino` a comunicação com o GPIO do dispositivo.

Esta declaração do `pom.xml` possui como projeto pai (*parent*) o `br.com.pontoclass:iot`, que irá configurar o experimento por completo, já que, como veremos mais adiante, o exemplo deste artigo também contará com um módulo web para a implementação da camada que ligará a aplicação de controle do sensor com a visualização do seu estado em um browser.

É importante notar que as dependências supracitadas foram declaradas no projeto com escopo `system`, fornecendo um diretório através do `systemPath`, que indica aonde o arquivo JAR da dependência se encontra diretamente dentro do projeto. Esta abordagem é necessária porque o projeto JArduino ainda não é armazenado em nenhum repositório público do Maven, pelo menos até o momento da escrita deste artigo. Por este motivo, para que essas dependências sejam geradas, é necessário que o desenvolvedor efetue o build – `mvn install` – do projeto JArduino baixado do repositório GitHub citado anteriormente e as configure diretamente no projeto como demonstrado na [Listagem 1](#).

Uma vez realizado o procedimento de criação do projeto e configuração das dependências, podemos partir para a codificação do Sketch Java que irá controlar o verdadeiro Sketch no firmware do Arduino. A partir deste momento é importante que o leitor observe a mudança de significado do termo Sketch, pois uma vez instalado o Sketch do projeto JArduino no dispositivo através da IDE Arduino, todo o resto da implementação em Java tratará de encapsular a complexidade de comunicação

com essa camada nativa e começaremos a chamar de Sketch esta implementação Java, que por sua vez possuirá os métodos de controle: `setup()` e `loop()`.

Antes de chegarmos à codificação da classe que implementará estes dois métodos, no entanto, precisamos criar o código Java capaz de encontrar o dispositivo através de sua porta, independentemente de qual Sistema Operacional executará esse código. Isto é extremamente importante para que seja mantida a portabilidade inherente ao Java; caso contrário, a aplicação poderá funcionar em determinados sistemas, como Linux ou OS X, e deixar de funcionar em outros sistemas, como o Windows. Na [Listagem 2](#), a classe `br.com.pontoclass.iot.hmeter.sketch.AbstractArduinoSketch` foi criada com a finalidade de facilitar este processo de conexão dos Sketches Java com o Arduino. Tal conexão acontece no bloco estático da classe, quando tentamos encontrar o possível nome da porta de comunicação serial para o Sistema Operacional detectado por meio da propriedade de sistema obtida na linha 12, através da chamada `System.getProperty("os.name").toLowerCase()`.

Observa-se que a classe `AbstractArduinoSketch` está estendendo (herdando) `org.sintef.jarduino.JArduino` e se mantendo abstrata, obrigando às suas implementações concretas que sobrescrevam os métodos `setup()` e `loop()`. Estes são os métodos disponíveis na API JArduino para que o desenvolvedor possa, respectivamente, configurar a aplicação e os pinos GPIO a serem utilizados e executar as ações inerentes ao comportamento esperado do dispositivo. Uma vez que a classe `AbstractArduinoSketch` é abstrata, ela precisará ser estendida por uma classe concreta para implementar estes métodos e, desta forma, criar o comportamento do Sketch. Para esta função, na [Listagem 3](#), a classe `br.com.pontoclass.iot.hmeter.sketch.Ultrassonic` é declarada.

Os métodos mais importantes a serem observados nessa classe são `setup()` e `loop()`. São eles que simulam o mecanismo dos Sketches nativos do Arduino. Nesses Sketches (escritos em C) também são disponibilizados dois métodos homônimos que possuem as mesmas funções dos aqui implementados em Java. Ou seja, a classe `Ultrassonic` é basicamente um Sketch Java com as mesmas funções que teria se fosse escrito como firmware diretamente no micro controlador, porém com todos os benefícios de ser uma classe Java.

Em nosso exemplo, o método `setup()` realiza a configuração do pino 5 como saída – *trigger* – e do pino 4 como entrada – *echo* – do sensor ultrassom (vide [BOX 5](#)). Na [Figura 5](#) podemos ver a conexão física entre o Arduino e o sensor através do fio amarelo no pino 5 do dispositivo conectando-se à entrada *trigger* do sensor, e através do fio azul no pino 4 do dispositivo conectado à saída *echo* do sensor. Para que o sistema sensorial funcione, o fio vermelho conectado à saída de energia do Arduino recebe 5V (cinco volts) para alimentá-lo, e a saída GND do sensor finaliza o circuito ao ser conectado ao pino de mesmo nome no Arduino.

Ainda no método `setup()`, podemos perceber que existe uma instância da classe `br.com.pontoclass.iot.hmeter.websocket.WebSocket` tendo o seu método `connect()` invocado (linha 30).



**Listagem 1.** Declaração do projeto hmeter com Maven.

```
01. <?xml version="1.0"?>
02. <project xmlns="http://maven.apache.org/POM/4.0.0"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
03.   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/xsd/maven-4.0.0.xsd">
04. <modelVersion>4.0.0</modelVersion>
05.
06. <parent>
07.   <groupId>br.com.pontoclass</groupId>
08.   <artifactId>iot</artifactId>
09.   <version>1.0.0-SNAPSHOT</version>
10. </parent>
11.
12. <groupId>br.com.pontoclass.iot</groupId>
13. <artifactId>hmeter</artifactId>
14. <name>Internet of Things - Height Meter Electronic Project</name>
15.
16. <properties>
17.   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
18. </properties>
19.
20. <dependencies>
21.   <dependency>
22.     <groupId>junit</groupId>
23.     <artifactId>junit</artifactId>
24.     <scope>test</scope>
25.   </dependency>
26.
27.   <dependency>
28.     <systemPath>${basedir}/lib/JArduino/org.sintef.jarduino.core-0.1.7.jar
   </systemPath>
29.   <scope>system</scope>
30.   <groupId>org.sintef.jarduino</groupId>
31.   <artifactId>core</artifactId>
32.   <version>0.1.7-SNAPSHOT</version>
33. </dependency>
34.
35.   <dependency>
36.     <systemPath>${basedir}/lib/JArduino/org.sintef.jarduino.serial-0.1.7.jar
   </systemPath>
37.   <scope>system</scope>
38.   <groupId>org.sintef.jarduino</groupId>
39.   <artifactId>serial</artifactId>
40.   <version>0.1.7-SNAPSHOT</version>
41. </dependency>
42.
43.   <dependency>
44.     <systemPath>${basedir}/lib/JArduino/org.kevoree.extra.osgi.rxtx-2.2.0.jar
   </systemPath>
45.   <scope>system</scope>
46.   <groupId>org.kevoree.extra.osgi</groupId>
47.   <artifactId>rxtx</artifactId>
48.   <version>2.2.0</version>
49. </dependency>
50.
51.   <dependency>
52.     <groupId>javax.websocket</groupId>
53.     <artifactId>javax.websocket-client-api</artifactId>
54.     <version>1.0</version>
55.   </dependency>
56.
57.   <dependency>
58.     <groupId>org.glassfish.tyrus</groupId>
59.     <artifactId>tyrus-container-grizzly-client</artifactId>
60.     <version>1.8.3</version>
61.   </dependency>
62. </dependencies>
63.
64. <build>
65.   <finalName>hmeter</finalName>
66.   <plugins>
67.     <plugin>
68.       <groupId>org.apache.maven.plugins</groupId>
69.       <artifactId>maven-compiler-plugin</artifactId>
70.       <version>3.3</version>
71.     </configuration>
72.     <source>1.8</source>
73.     <target>1.8</target>
74.   </configuration>
75. </plugin>
76.
77.     <plugin>
78.       <groupId>org.apache.maven.plugins</groupId>
79.       <artifactId>maven-shade-plugin</artifactId>
80.       <version>2.4.1</version>
81.     <executions>
82.       <execution>
83.         <phase>package</phase>
84.         <goals>
85.           <goal>shade</goal>
86.         </goals>
87.       <configuration>
88.         <transformers>
89.           <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransformer">
90.             <manifestEntries>
91.               <Main-Class>br.com.pontoclass.iot.hmeter.main.Bootstrap</Main-Class>
92.             <Build-Number>${project.version}</Build-Number>
93.           </manifestEntries>
94.         </transformer>
95.       </transformers>
96.       <minimizeJar>true</minimizeJar>
97.     <artifactSet>
98.       <includes>
99.         <include>br.com.pontoclass.iot:hmeter</include>
100.      </includes>
101.    </artifactSet>
102.  </configuration>
103. </execution>
104. </executions>
105. </plugin>
106. </plugins>
107. </build>
108.</project>
```

# Primeiros passos no mundo da Internet das Coisas – Parte 1

## BOX 5. Sensores Ultrassônicos

Sensores ultrassônicos são dispositivos capazes de detectar corpos (objetos, pessoas, substâncias) por meio do envio de sinais ultrassonoros propagados pela parte emissora (trigger) e da recepção da onda refratada pelo corpo e recebida pelo detector (echo). Através do cálculo do tempo passado entre a emissão e a recepção da onda em microssegundos, o sensor consegue calcular com boa precisão a distância entre ele e o objeto refrator (corpo).

Curiosamente, os sensores ultrassônicos, como o apresentado neste artigo, utilizam o mesmo princípio de localização de objetos realizada pelos morcegos, que também se utilizam da técnica de emissão sonora para obter uma base da localização das coisas que estão pelo seu caminho.

Esta classe e todo o restante do experimento serão explorados no próximo artigo, quando implementaremos toda a camada web com WebSockets para apresentar os dados extraídos da medição com o sensor ultrassônico e para permitir a interação via interface com o dispositivo, permitindo solicitar a sua calibragem e criando uma integração de ponta a ponta, do hardware à visualização web.

No momento da chamada ao método `connect()` é realizada a tentativa de conexão com o servidor WebSocket que conectará

## Listagem 2. Código da classe abstrata AbstractArduinoSketch.

```
01. package br.com.pontoclass.iot.hmeter.sketch;
02.
03. import gnu.io.CommPortIdentifier;
04. import java.util.Enumeration;
05. import org.sintef.jarduino.JArduino;
06.
07. public abstract class AbstractArduinoSketch extends JArduino {
08.     private static final String[] PORT_NAMES;
09.     private static String TRIED_PORT_NAME;
10.
11.     static {
12.         final String OS = System.getProperty("os.name").toLowerCase();
13.         if(OS.contains("win")) {
14.             PORT_NAMES = new String[] {"COM"};
15.         } else if(OS.contains("mac")) {
16.             PORT_NAMES = new String[] {"/dev/usbmodem"};
17.         } else if(OS.contains("nix") || OS.contains("nux") || OS.contains("aix")) {
18.             PORT_NAMES = new String[] {"/dev/usbdev", "/dev/tty", "/dev/serial"};
19.         } else if(OS.contains("sunos")) {
20.             PORT_NAMES = new String[] {"/dev/tty"};
21.         } else {
22.             PORT_NAMES = new String[] {"/dev/tty"};
23.         }
24.         @SuppressWarnings("unchecked") Enumeration<CommPortIdentifier>
```

```
25.         portEnum = CommPortIdentifier.getPortIdentifiers();
26.         System.out.println("Trying:");
27.         boolean set = false;
28.         outer: while (portEnum.hasMoreElements()) {
29.             CommPortIdentifier currPortId = (CommPortIdentifier)
30.                 portEnum.nextElement();
31.             System.out.println(String.format("\tPort [%s]", currPortId.getName()));
32.             for (String portName: PORT_NAMES) {
33.                 if (currPortId.getName().startsWith(portName)) {
34.                     TRIED_PORT_NAME = currPortId.getName();
35.                     set = true;
36.                     break outer;
37.                 }
38.             }
39.             if(!set) {
40.                 TRIED_PORT_NAME = "/dev/tty";
41.             }
42.         }
43.         public AbstractArduinoSketch() {
44.             super(TRIED_PORT_NAME);
45.         }
46.     }
```

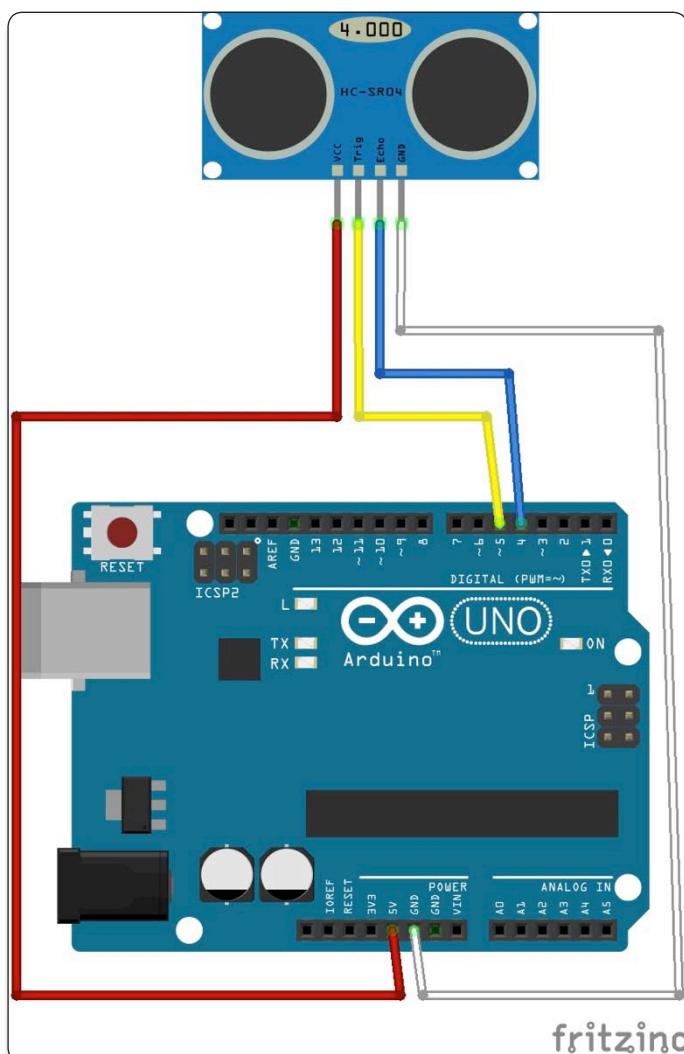
## Listagem 3. Declaração da classe Ultrassonic.

```
01. package br.com.pontoclass.iot.hmeter.sketch;
02.
03. import java.util.logging.Logger;
04. import java.util.stream.IntStream;
05. import org.sintef.jarduino.DigitalPin;
06. import org.sintef.jarduino.PinMode;
07. import br.com.pontoclass.iot.hmeter.strategy.Strategy;
08. import br.com.pontoclass.iot.hmeter.websocket.WebSocket;
09.
10. public class Ultrassonic extends AbstractArduinoSketch {
11.
12.     private static final int AVERAGE_PRECISION = 10;
13.     private static final Logger LOGGER = Logger.getLogger(
14.         Ultrassonic.class.getName());
15.     private Strategy strategy = Strategy.Available;
16.     private WebSocket socket = new WebSocket(
17.         this, "ws://localhost:8080/websocket/hmeter");
18.     private float lastResult = -1;
19.
20.     @Override protected void loop() {
21.         synchronized(strategy) {
22.             float result = strategy.execute(this);
23.             if(lastResult != result) {
24.                 LOGGER.info(String.format("Último resultado (diferente) %f obtido do medidor: [%f].", result));
25.             }
26.             lastResult = result;
27.         }
28.     }
29.     @Override protected void setup() {
30.         socket.connect();
31.         pinMode(DigitalPin.PIN_4, PinMode.OUTPUT);
32.         pinMode(DigitalPin.PIN_5, PinMode.INPUT);
33.     }
34.
35.     public float average() {
36.         return IntStream.range(1, AVERAGE_PRECISION)
37.             .mapToObj(i -> ultrassonic(DigitalPin.PIN_4,
38.                                         DigitalPin.PIN_5)/29f/2f/100f)
39.             .reduce((a, b) -> a+b)
40.             .orElse(0f)/AVERAGE_PRECISION;
41.     }
42.
43.     public void setStrategy(Strategy strategy) {
44.         synchronized(strategy) {
45.             this.strategy = strategy;
46.         }
47.     }
48.
49.     public WebSocket getWebSocket() {
50.         return this.socket;
51.     }
52. }
```



o Sketch **Ultrassonic** à camada de visualização responsável por exibir a medida de altura das pessoas que se posicionarem abaixo do dispositivo. Note que a variável **socket** é instanciada na linha 15, recebendo o próprio objeto **Ultrasonic** através do **this** (no próximo artigo veremos como o **WebSocket** irá utilizar esta instância) e uma **String** com a URL da aplicação web em que o **WebSocket** está escutando por novas conexões.

Nesta primeira parte do nosso estudo sobre o mundo da Internet das Coisas, avançamos bastante na compreensão da IoT e porquê ela é o futuro. Com o conhecimento adquirido até aqui já é possível exercitar a integração de novos sensores ao dispo-



**Figura 5.** Conexão entre Arduino e sensor ultrassônico

sitivo, dando a ele quaisquer outras funções de acordo com a capacidade desejada (detectar imagens ou sons, por exemplo), pois o princípio de controle do GPIO é sempre o mesmo. Para tanto, é importante informar-se muito bem quando da aquisição de novos sensores, para saber como eles trabalham, se necessitam de resistores ou outros tipos de componentes auxiliares na montagem de seus circuitos.

## 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/hanotecnologia/3254-o-que-sao-mems-.htm>

### Como funcionam os sensores ultrassônicos (ART691).

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

### GitHub – Jarduino.

<https://github.com/romerorsp/JArduino>

### Eletrônica Didática - Protoboard.

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

## Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)

Ajude-nos a manter a qualidade da revista!



# Conheça os papéis do desenvolvedor na cultura DevOps

Aprenda nesse artigo como desenvolver software aderente à cultura DevOps

**A** paternidade do termo DevOps é atribuída ao administrador de sistemas Patrick Debois que, incomodado ao experimentar os tradicionais conflitos entre desenvolvedores e administradores de sistemas, sentiu a necessidade de fazer algo para que este tipo de impasse fosse, enfim, solucionado.

Este movimento iniciou-se no ano de 2007 e, após alguns encontros de Debois com outros profissionais, como o desenvolvedor Andrew Shafer e o engenheiro de sistemas John Allspaw, culminou na organização de um evento para discutir especificamente esta questão dos desafios da integração entre as áreas de desenvolvimento e operações das empresas.

A ideia do evento era muito interessante, mas precisava de um título com boa sonoridade e que fosse curto o suficiente para facilitar sua disseminação através das redes sociais – principalmente o Twitter. Partindo desses princípios, Debois teve a ideia de combinar as três primeiras letras das duas áreas, chegando no nome ‘DevOps Days’. O evento, muito bem aceito pela comunidade, foi bastante comentado no Twitter e o termo, DevOps, ganhou rapidamente uma expressiva popularidade. Desde então é o nome que representa este novo *mindset* dentro do universo da TI.

A partir disso, ao longo deste artigo cobriremos os principais aspectos ligados ao desenvolvimento de software em um projeto DevOps, analisando algumas ferramentas, plataformas e técnicas importantes para que se desenvolva código com qualidade e de forma gerenciada.

Para o leitor que quiser conferir a história completa a respeito do surgimento do DevOps, é só conferir a referência que deixamos na seção **Links**.

## Fique por dentro

DevOps não é apenas um dos termos mais citados no universo da computação dos dias de hoje. Tornou-se, desde já, um item essencial no currículo e no cotidiano de qualquer profissional de computação que trabalhe com desenvolvimento de software e deseje se manter atualizado e, principalmente, competitivo no mercado de trabalho.

Entretanto, acompanhar o mundo de DevOps não é tarefa fácil. As ferramentas, práticas e metodologias a ela associadas evoluem muito rapidamente, trazendo a cada semana um leque ainda maior de opções para atacar os desafios de desenvolvimento de software.

Com base nisso, este artigo tem por objetivo introduzir os principais aspectos deste novo mindset e, a partir de plataformas como Maven, Git, Redmine e Jenkins, demonstrar na prática o que e como fazer para aplicar os princípios de DevOps no nosso dia a dia.

## Mas o que é DevOps, afinal?

DevOps é o termo usado para representar um modelo de pensamento em que as áreas de desenvolvimento e operações se harmonizam no intuito de otimizar a construção e entrega de software. Trata-se de um paradigma cujo objetivo principal é eliminar os obstáculos historicamente enfrentados pelas empresas na entrega de sistemas, através de um fluxo mais equilibrado, sinérgico, entre todas as áreas ativas neste processo.

Não é, entretanto, algo que se pode chamar de novo. Assim como outros movimentos que surgem e rapidamente se popularizam na computação – como o próprio fenômeno da Internet das Coisas ou o advento das metodologias ágeis –, a popularização de DevOps tem provocado em alguns profissionais mais experientes o sentimento de que, a bem da verdade, não há nada nesta proposta que já não viesse sendo praticado por eles há anos.



De fato, toda esta glamourização pode parecer um exagero para alguns, pois indica um certo ineditismo. No entanto, há o lado positivo de toda a história: este rótulo de ‘pop-star’ que DevOps ganhou no mercado acaba atraindo uma grande parcela de pessoas que não está habituada a trabalhar em conformidade com os conceitos e as práticas sugeridas pelo movimento. E isto, aliado a todo o arsenal de ferramentas que tornam possível a implementação da ideia, contribui indubitavelmente para uma transformação bastante positiva do nosso mercado.

### O aspecto cultural

Uma implementação bem-sucedida da cultura DevOps em qualquer empresa passa, necessariamente, por uma mudança cultural. Embora esta não seja a perspectiva tonante do artigo, é importante termos em mente que todo reforço realizado na infraestrutura de uma empresa terá sido em vão caso seus colaboradores não incorporem a verdadeira mensagem que este novo paradigma busca transmitir.

Existem hoje, não raros, casos em que ferramentas de trabalho são empregadas de forma equivocada simplesmente pelo fato de que a empresa, ainda presa às suas antigas raízes, insiste em manter seu foco em valores errados.

Um caso clássico, com o qual muitos leitores poderão, inclusive, se identificar, é a prática de revisão de código. Existem, atualmente, ferramentas que geram sessões de revisão a cada submissão de código no repositório. O objetivo é garantir que nada seja versionado sem conhecimento e, mais importante ainda, discussão e consenso em torno de sua relevância e validade.

No entanto, principalmente devido a uma má gestão de projetos, o que acaba acontecendo é que os colaboradores, sempre com prazos bastante apertados, negligenciam este processo e apenas analisam rapidamente o código-fonte. Assim, a prática em si perde exatamente a sua principal função, e o impacto na qualidade final do produto/serviço, a médio e longo prazos, tende a ser bastante negativo.

O maior desafio de DevOps nos dias atuais é, portanto, equilibrar a marcha em que tecnologia e pessoas evoluem. Praticá-la com qualidade passa, necessariamente, por revisar todas as práticas adotadas no ambiente em que nos acostumamos a trabalhar e, sem exceções de papéis ou departamentos, empenharmo-nos para torná-las melhores, mais enxutas, mais eficientes. Ademais, a atmosfera precisa ser uma só, e toda a empresa deve – bem como seus clientes, indiretamente – respirar o mesmo ar de mudança que este paradigma propõe.

### Automatização de processos

Além da cultura, existe outro aspecto essencial para que DevOps seja bem-sucedido: a *automatização*. Mas, afinal de contas, o que devemos automatizar? E, se temos que fazê-lo, quando é o melhor momento?

Em DevOps, tanto Desenvolvimento quanto Operações acabam se beneficiando largamente quando seus processos vitais passam a ser controlados por software. Em resumo, devemos tornar

automático tudo aquilo cuja natureza é repetitiva e que, portanto, estaria bastante sujeita a erros quando executadas manualmente, por seres humanos.

Logo, cada empresa deve observar o que, dentro de seus processos habituais de desenvolvimento e TI, tem essas características e, em seguida, redesenhe-los de forma a usufruir de ferramentas e plataformas que executem tudo com maior precisão e velocidade, melhor qualidade e, ainda, menor custo. Algumas atividades que normalmente acabam se tornando francas candidatas a um processo de automatização são:

- Compilação e empacotamento de código;
- Geração de build a partir de submissões de código-fonte;
- Análise e gestão de qualidade (débito técnico, métricas em geral);
- Execução de suítes de testes de regressão a cada submissão de código;
- Implantação agendada de ‘entregáveis’ (versões do produto/serviço).

Caso paremos para observar todo o movimento DevOps, constataremos que esta é a frente que mais evoluiu. A quantidade e a diversidade de tecnologias que surgem a cada semana são muito grandes, e acompanhar esta evolução acaba se tornando um grande desafio.

Entretanto, é muito importante que não se confunda a parte pelo todo, pois DevOps não é e nunca será sinônimo de automatização. Esta é, sem dúvida, essencial para aquela, mas não a define por si só.

### Postura DevOps?

Sem uma mudança cultural, como já discutimos há pouco, não há DevOps. É indiscutível que as questões técnicas são muito importantes (e serão elas os ingredientes a compor todo o artigo), mas antes de entrarmos definitivamente nesse movimento,



# Conheça os papéis do desenvolvedor na cultura DevOps

dedicaremos algumas palavras finais ao que denominaremos de '*postura DevOps*'.

É fundamental que todo o arsenal tecnológico disponível no mercado trabalhe a nosso favor, e isto só ocorrerá caso seus usuários usufruam dele com responsabilidade e, principalmente, consciência de seu papel dentro da equipe.

Este último ponto, o da consciência, ainda é uma barreira a ser vencida no mercado corporativo 'convencional', enquanto em *start-ups* normalmente não chega nem a ser identificada.

Transparência e naturalidade ao lidar com adversidades dentro de um projeto de software são, a partir de DevOps, premissas que não devem, jamais, ser negligenciadas. Nesta nova filosofia de trabalho, é importante que as pessoas entendam que, no cotidiano de desenvolvimento de software, o espírito a ser cultivado é o de uma constante busca por colaboração, progresso coletivo, em que culpa ou mérito transcendam o indivíduo. 'Quebrar' builds, equivocar-se em algum trecho de código ou expor fragilidades individuais devem ser coisas absolutamente naturais em qualquer *equipe DevOps*, pois o objetivo comum de todos tem de ser apenas uma entrega ágil e com qualidade assegurada ao cliente. Ideologicamente, ao atingir um ponto ótimo de maturidade no processo, o próprio cliente passaria a abraçar a causa e, assim, tornar-se um colaborador ativo e frequente do produto ou serviço em desenvolvimento.

## Objetivos do artigo

Este artigo terá foco puramente prático. Nosso objetivo primeiro será orientá-lo na configuração de um ambiente, passo a passo, com algumas das principais ferramentas, frameworks e plataformas associadas a DevOps e acessíveis a todos na rede.

Para isso, abordaremos os princípios básicos relacionados ao desenvolvimento de software, cobrindo áreas importantes como escrita de código-fonte e código de teste, versionamento e gerenciamento de atividades básicas em nível de gerenciamento do projeto (como compilação, empacotamento, testes, dentre outros).

Ao seu final, a meta será garantir que temos um projeto de software bem estruturado, devidamente versionado e gerenciável.

Nos próximos artigos, entraremos em diversos outros aspectos, como:

- A automatização do processo de build usando servidores de integração contínua;
- A automatização do processo de entrega de software;
- O provisionamento de containers e máquinas virtuais;
- Gestão integrada de requisitos e defeitos.

Assim, esperamos apresentar ao leitor um guia prático para a implantação de DevOps utilizando ferramentas adotadas largamente no mercado, de modo que até os projetos pessoais que venhamos a fazer passem a ser suportados por uma estrutura sólida e eficiente de gestão, do desenvolvimento à entrega.

## Exemplo prático

A partir de agora veremos como um projeto de software pode ser gerenciado no contexto de DevOps. É importante ressaltar, no entanto, que nem todos os aspectos serão cobertos, pois este é um tema muito amplo e exigiria uma série bastante longa de artigos para ser totalmente explorado. Focaremos, portanto, em algumas áreas essenciais que todo programador deve dominar.

## Organização também é DevOps

No DevOps, há uma máxima que devemos respeitar: 'organização também é DevOps'. Neste contexto, há algumas práticas que recomendamos aos leitores, tais como:

- Definir uma convenção para nomeação de diretórios de projetos;
- Definir uma estrutura de diretórios para identificar ferramentas e plataformas a serem empregadas no nosso trabalho.

Como exemplo prático, na forma de uma sugestão para o leitor, observe a **Figura 1**. Nela, é apresentada uma estrutura de diretórios que visa, de forma simples e memorizável, diagramar tudo aquilo que é necessário para o desenvolvimento tanto de projetos pessoais quanto outros para uma empresa à qual estejamos vinculados. Em ambos os casos, perceba que foram definidos diretórios tanto para configuração de ambiente (*DevEnv*) quanto para projetos (*Projects*) e desenvolvimento (*Workspaces*). Da forma como a figura sugere, fica simples identificar o caminho em que os recursos que precisamos para trabalhar estão localizados, poupando-nos um tempo significativo ao longo do dia.

## O escopo do produto

Com o advento da Internet das Coisas, muitos profissionais vêm tomando mais contato com a área da eletrônica. Neste contexto, uma das coisas que acabam sendo necessárias, quando se quer montar um circuito, é entender bem a lógica do cálculo de resistores, que são componentes essenciais e presentes em praticamente todos os esquemas eletrônicos que venhamos a estudar. Tomando carona neste tema, construiremos uma calculadora de resistências



responsável por determinar o valor nominal de um dado resistor a partir das cores de suas quatro faixas, fornecidas pelo usuário. Neste artigo, focaremos no desenvolvimento de uma interface *standalone*, em Swing, e no modelo de dados.

Resistores, como já dito, são normalmente identificados por um conjunto de quatro faixas coloridas, em que três delas são dedicadas a representar o valor de base e a última indica a porcentagem de variação associada ao material usado na sua fabricação. Cada cor está associada a um numeral, e a composição final da resistência, bem como do intervalo de variação ('margem de erro' associada ao material) é determinada de acordo com as seguintes regras:

- A leitura deve começar da esquerda para a direita;
- A primeira e a segunda faixa coloridas compõem um número de base;
- A terceira faixa compõe uma potência de 10;
- A quarta faixa refere-se ao intervalo de variação do material usado na fabricação do resistor.

O valor de cada cor está descrito na **Tabela 1**. Baseado nesta tabela, imaginemos um exemplo prático em que um resistor seja composto pelo seguinte padrão de cores:

- Faixa 1: Marrom;
- Faixa 2: Preto;
- Faixa 3: Vermelho;
- Faixa 4: Dourado.

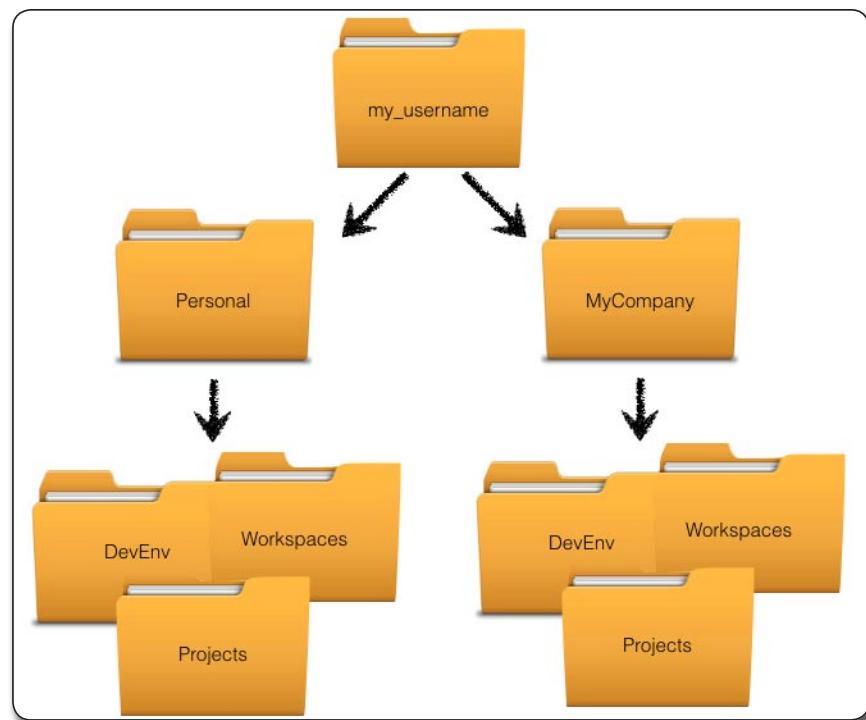
De acordo com os valores tabelados, o resultado seria um número de base 10 (marrom equivalendo a 1 e preto equivalendo a 0), multiplicado por 10 elevado à potência de 2 (faixa vermelha) e um intervalo de variação de 5%. Logo, o valor nominal do resistor é de 1000 Ohms, variando entre o mínimo de 950 e o máximo de 1050 Ohms.

Agora que já sabemos o escopo do projeto que desenvolveremos, precisamos configurar uma plataforma que nos permita gerenciar os seus principais aspectos de forma simples, configurável e, o mais importante, automatizado.

### **Configurando o Maven para o gerenciamento do projeto**

Existem algumas plataformas muito populares no mercado que podem ser usadas para gerenciar todas as fases de um projeto de software. Atualmente, a plataforma que se encontra no topo desta lista é o Apache Maven.

Para configurá-lo basta que baixemos o pacote de instalação diretamente do site do projeto (vide seção **Links**) e o descompactemos em algum diretório em seu computador. Prefira, para efeito de minimização de problemas, caminhos curtos e nomes de pastas sem espaços em branco. Algumas vezes, quando o nome do diretório possui espaços em brancos, pode haver confusão no momento da localização das bibliotecas.



**Figura 1.** Estrutura de diretórios para organização de ambiente

Cor da faixa	Valor
Preto	0
Marrom	1
Vermelho	2
Laranja	3
Amarelo	4
Verde	5
Azul	6
Violeta	7
Cinza	8
Branco	9
Dourado	+/- 5%
Prateado	+/- 10%

**Tabela 1.** Tabela de cores para cálculos de resistência

Para este projeto, basta que passemos pelos seguintes passos:  
1. Baixe o pacote de instalação do Maven em seu computador;  
2. Descompacte-o respeitando a estrutura sugerida anteriormente, ou seja, no caminho *my\_username/Personal/DevEnv*.

É importante que, antes mesmo de instalar o Maven em seu computador, o JDK já esteja configurado. Caso contrário, ferramentas como o Eclipse e o próprio Maven não funcionariam.

# Conheça os papéis do desenvolvedor na cultura DevOps

Para certificar-se que tudo está instalado e configurado corretamente, abra um terminal (Prompt) e execute o comando `mvn -version`. O resultado deverá ser algo como apresentado na **Listagem 1**.

Até aqui, toda a configuração da ferramenta foi feita localmente. Isto garantirá que todo o trabalho realizado localmente, no ambiente de desenvolvimento, estará correto. Quando adicionamos ao projeto um servidor de integração contínua (como o Jenkins), é importante garantirmos que ele esteja devidamente configurado, com a mesma versão do Maven utilizada em desenvolvimento, para que cada submissão de código possa resultar no mesmo processo de compilação, testes e empa-

cotamento realizado ao longo de todo o desenvolvimento.

## Escolhendo e configurando sua ferramenta de desenvolvimento

O ambiente de desenvolvimento também é outro tema em que boas discussões sempre surgem. A dica aqui é escolher a ferramenta que mais lhe traga conforto, já que o conceito de melhor e pior é bastante subjetivo e depende diretamente do quanto nos sentimos seguros e familiarizados com os recursos que nos são apresentados.

Entretanto, há algumas IDEs que já gozam de alto prestígio no mercado e, por isso, merecem ser destacadas no artigo. A principal delas ainda é o Eclipse, que

será utilizada ao longo do artigo exatamente por este motivo. Uma alternativa ao Eclipse, muito robusta e, em diversos aspectos, mais ‘smart’ e eficiente que seu concorrente, é o IntelliJ IDEA. Por fim, embora seja menos utilizado no mercado, é importante também dar destaque ao NetBeans.

A versão do Eclipse que utilizamos neste artigo é a *Mars*, a mais recente no momento em que este artigo foi escrito. A distribuição que baixamos e instalamos é a ‘Java EE for Developers’, que pode ser encontrada facilmente na página de downloads da Fundação Eclipse (vide **Links**).

O pacote de instalação, assim como no caso do Maven, também é um ZIP. Portanto, basta que o descompactemos e ele estará pronto para uso. Novamente, a título de sugestão, o caminho a ser escolhido para a instalação da IDE é `my_username/Personal/DevEnv`.

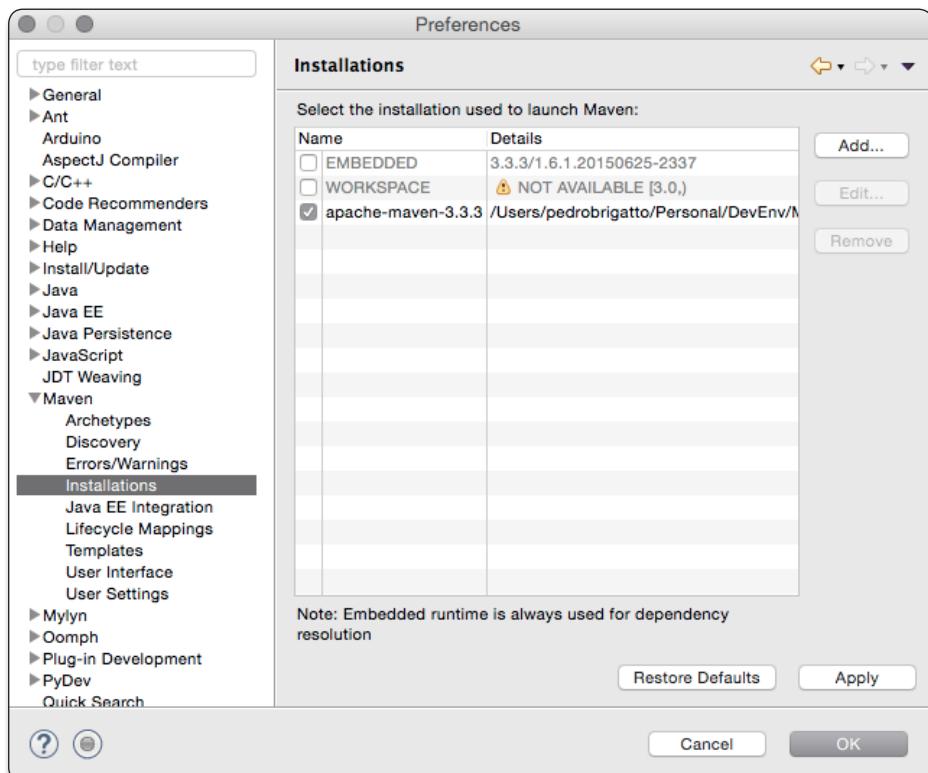
Assim que tiver baixado e descompactado o Eclipse em sua máquina, abra o programa a partir de seu executável. Realizaremos, neste momento, a configuração do Maven dentro desta IDE, garantindo que a versão a ser utilizada pelo Eclipse seja a que instalamos na seção anterior.

Para isso, abra as preferências do Eclipse e selecione a opção *Maven* na lista à esquerda. Em seguida, selecione a opção *Installations* e clique no botão *Add...*. Agora, encontre a distribuição do Maven que instalamos momentos atrás e confirme a operação clicando no botão *Apply*. Por fim, certifique-se que a instalação que você acabou de confirmar é a que ficará selecionada, conforme indicado na **Figura 2**.

Logo depois, precisamos apontar para o arquivo de configuração do Maven que acabamos de definir. A **Figura 3** exibe, na estrutura de menu à esquerda, a opção *Maven > User Settings*. Ao clicarmos nela, vemos que existem as configurações *Global (Global Settings)* e *User (User Settings)*. Em ambas, apontaremos para o arquivo de configuração da versão do Maven que acabamos de configurar; para isto, basta que apontemos para o arquivo `settings.xml` que se encontra no diretório `conf` do Maven (seguindo as convenções já

**Listagem 1.** Verificação da versão do Apache Maven instalado na máquina.

```
Pedro-Brigattos-MacBook:~ pedrobrigatto$ mvn -version
Apache Maven 3.2.3 (33f8c3e1027c3ddde99d3cdebad2656a31e8fdf4; 2014-08-11T17:58:10-03:00)
Maven home: /Users/pedrobrigatto/Personal/DevEnv/MavenBundles/apache-maven-3.2.3
Java version: 1.8.0_25, vendor: Oracle Corporation
Java home: /Library/Java/JavaVirtualMachines/jdk1.8.0_25.jdk/Contents/Home/jre
Default locale: pt_BR, platform encoding: UTF-8
OS name:"mac os x", version:"10.10.5", arch:"x86_64", family:"mac"
```



**Figura 2.** Definição do uso de uma instalação específica do Maven na IDE Eclipse

sugeridas no artigo, o caminho completo até o arquivo corresponderia a algo como `my_username/Personal/DevEnv/<maven_dir>/conf/settings.xml`.

O próximo passo será a configuração de outra ferramenta bastante importante no desenvolvimento de sistemas e, consequentemente, na entrega final em um contexto de DevOps. Trata-se da *EclEmma*, utilizada para análise de cobertura de código-fonte por testes.

Lembremos, mais uma vez, que a facilitação da integração entre DEV e OPS passa, necessariamente, por um serviço de boa qualidade entregue nas duas ‘pontas’. Neste primeiro momento, estamos cuidando de aspectos mais relacionados à frente DEV, e cuidar muito bem desta área resultará em um processo final muito mais harmonioso, em que OPS será naturalmente beneficiado.

Para configurarmos o *EclEmma* no Eclipse, vá até a barra de menu da IDE e selecione a opção *Help > Install New Software*. Na janela que se abre, clique no botão *Add* para configurarmos o plug-in do *EclEmma*. A partir daí, preencha o formulário com os valores indicados na **Figura 4**, confirmando a operação. Ao reiniciar o Eclipse, o plug-in já estará devidamente configurado. Na seção **Links** o leitor encontrará a URL para a página do projeto, caso queira saber mais informações.

Nos próximos artigos sobre DevOps, voltaremos a abordar a configuração do Eclipse para adicionar outros plug-ins relacionados à gestão do software em desenvolvimento. Lembre-se que um dos aspectos mais importantes desta cultura é a qualidade do produto em construção e, neste sentido, existem inúmeras ferramentas que certamente nos auxiliam na observação e no mapeamento do código-fonte no tocante a aspectos como Débito Técnico, vulnerabilidades, complexidade ciclomática, dentre outros.

#### Criando um repositório de código para o projeto

Para este projeto, utilizaremos o Git, um sistema de versionamento de código distribuído que nos permite controlar as versões de nosso software localmente e, ainda, submetê-lo também a um

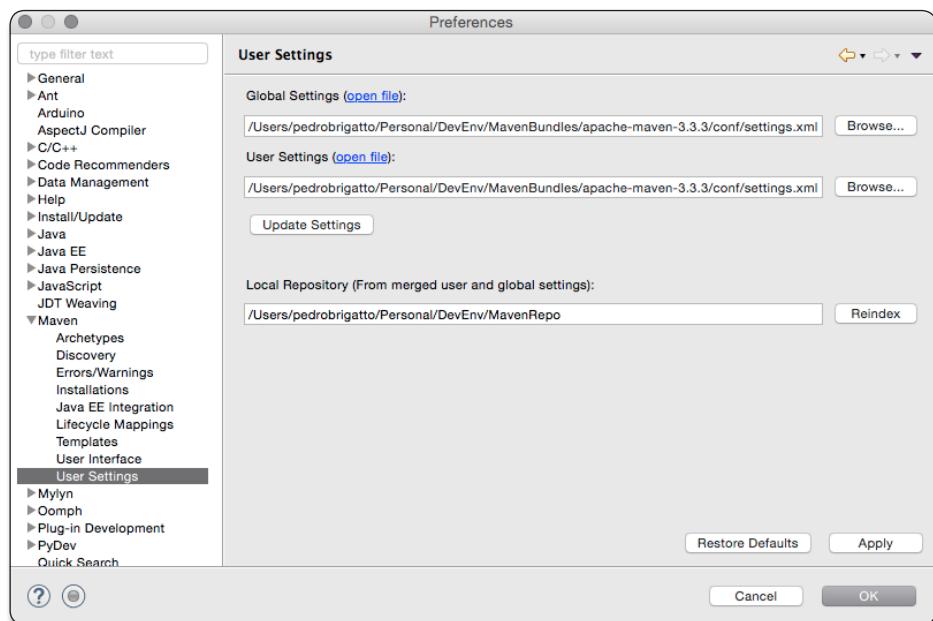


Figura 3. Configuração do Maven na IDE Eclipse

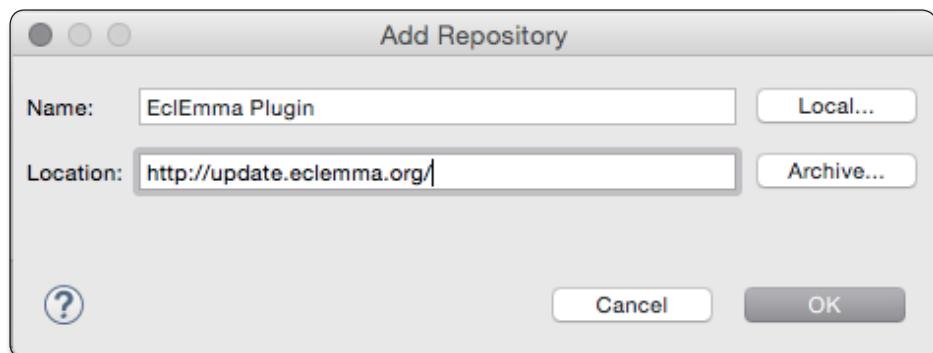
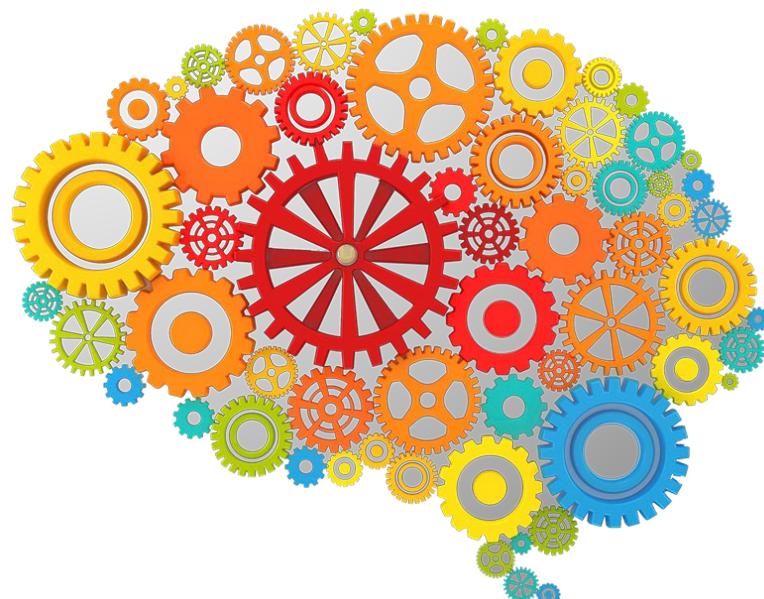


Figura 4. Configuração do plug-in do EclEmma na IDE Eclipse



repositório centralizado (normalmente remoto) a partir do qual todos os desenvolvedores de um time possam extrair suas cópias do projeto.

Atualmente, os principais serviços de versionamento de código que utilizam o Git são o GitHub e o Bitbucket, e a diferença básica entre eles é que, no caso do Bitbucket, é possível criar tanto repositórios privados quanto públicos sem custo algum. No caso do GitHub, repositórios privados só são possíveis a partir de planos pagos.

Ao longo deste artigo, faremos uso do GitHub. A URL para o código-fonte do projeto-base pode ser encontrada na seção [Links](#).

Para começar a usar o Git, entretanto, não é necessário que tenhamos conta em nenhum desses serviços mencionados. Basta que instalemos o Git em nossas máquinas e o usemos. No entanto, é natural que queiramos ter um serviço no qual possamos confiar e, desta forma, salvar nosso código-fonte também em uma fonte remota, livre de intempéries e riscos como uma queima de HD. E, para isto, o GitHub ou o Bitbucket são excelentes opções.

Todo o material para download, instalação e configuração do Git em sua máquina, bem como um guia para criar uma conta no GitHub ou no Bitbucket, podem ser encontrados na seção [Links](#). Recomendamos que, caso o leitor queira acompanhar o artigo fazendo as configurações e executando os comandos passados, seja feita uma pausa para ler o material mencionado. Quando estiver com tudo configurado, volte e continue a partir daqui.

Nosso projeto será salvo localmente a partir das convenções pré-definidas, em tópico anterior. Deste modo, via linha de comando, navegue até o diretório `my_username/Pessoal/Projects/` e crie, nele, uma pasta para o projeto. Neste caso, conforme podemos ver na [Listagem 2](#), o nome escolhido para a pasta é `/Users/pedro-brigatto/Pessoal/Projects/DevOpsArticle`.

Assim que executamos os comandos verificados nessa listagem, teremos o nosso repositório local criado. A partir disso o leitor poderia se perguntar: é só começar a codificar, então?

Teoricamente, sim. Entretanto, é importante revisitarmos nossa discussão acerca do aspecto cultural de DevOps e reforçá-la com a adição de algumas práticas relacionadas à versionamento de código altamente recomendadas. São elas:

- Atualize seu repositório antes de iniciar seu trabalho. Sempre que possível, inicie sua jornada diária atualizando o código-fonte de sua cópia do repositório. Desta forma, o leitor garantirá que está trabalhando em uma versão atual e reduzirá as chances de problemas na hora de integrar seu código;
- Submeta código com frequência. Quanto mais frequentemente submetemos código-fonte no repositório, menores são as chances de conflitos entre a versão que estamos editando e aquela em edição por outros desenvolvedores;
- Escolha mensagens significativas para identificar submissões de código;
- Jamais submeta código sem certificar-se de que ele esteja compilando e seja funcional (atenda o propósito a que se destina);
- Evite, sempre que possível, submissões de código sem o código de teste que valide toda a lógica contida nele.

Quando o leitor acessar o repositório `git` que configuramos neste artigo, perceberá que todo o código-fonte já se encontra lá. Entretanto, quando aprendemos, em uma seção anterior, a criar o repositório GitHub, vimos que o resultado foi apenas uma estrutura simples de diretório, e ainda não havia código-fonte algum por lá. O que houve, portanto, foi a posterior criação do projeto e sua consequente submissão ao repositório, passos que elucidaremos neste momento.

Recordando um pouco o que vimos há pouco, pela [Listagem 2](#), o repositório remoto de nosso projeto foi clonado em nossa máquina de desenvolvimento. O próximo passo que demos, para criar o projeto propriamente dito, foi abrir a nossa IDE e, quando perguntados sobre qual workspace usar, apontamos exatamente para a pasta em que o nosso projeto Git foi clonado. Desta forma, tudo o que criássemos via Eclipse seria automaticamente salvo nesta pasta.

O projeto que o leitor vê no repositório, portanto, foi inteiramente desenvolvido e disponibilizado seguindo este processo. Esta é apenas a primeira versão de código, utilizado para orientar o desenvolvimento deste texto. Nos próximos artigos, utilizaremos este mesmo projeto para introduzir mais atividades e processos, relacionados principalmente à área de Operações (OPS). Os comandos que utilizamos para submeter a primeira versão do projeto ao repositório remoto do GitHub pode ser visto na [Listagem 3](#).

Observe que o que fizemos foi:

- Adicionar todos os arquivos do diretório ao Git (através do comando `git add`);
- Gerar uma primeira versão de código ao submetermos o conteúdo do projeto ao controle do Git (através do comando



### Listagem 2. Verificação da localização e conteúdo do projeto criado usando Git.

```
Pedro-Brigattos-MacBook:~ pedrobrigatto$ pwd  
/Users/pedrobrigatto  
Pedro-Brigattos-MacBook:~ pedrobrigatto$ cd Personal/Projects/DevOpsArticle/  
Pedro-Brigattos-MacBook:DevOpsArticle pedrobrigatto$ git clone https://github.  
com/pedrobrigatto/devmedia_devops_series.git  
Cloning into 'devmedia_devops_series'...  
remote: Counting objects: 3, done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  
Unpacking objects: 100% (3/3), done.  
Checking connectivity... done.  
Pedro-Brigattos-MacBook:DevOpsArticle pedrobrigatto$ ls -la  
total 0  
drwxr-xr-x 3 pedrobrigatto staff 102 Nov 7 11:47 .  
drwxr-xr-x 13 pedrobrigatto staff 442 Nov 7 11:45 ..  
drwxr-xr-x 4 pedrobrigatto staff 136 Nov 7 11:47 devmedia_devops_series  
Pedro-Brigattos-MacBook:DevOpsArticle pedrobrigatto$ cd devmedia_  
devops_series/  
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ ls -la  
total 8  
drwxr-xr-x 4 pedrobrigatto staff 136 Nov 7 11:47 .  
drwxr-xr-x 3 pedrobrigatto staff 102 Nov 7 11:47 ..  
drwxr-xr-x 13 pedrobrigatto staff 442 Nov 7 11:47 .git  
-rw-r--r-- 1 pedrobrigatto staff 192 Nov 7 11:47 README.md  
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$
```

*git commit -m <comentário>*). Perceba que, até aqui, o código se encontra versionado apenas e tão somente na máquina local, de desenvolvimento;

- Submeter, por fim, o código versionado localmente também ao sistema de versionamento remoto (através do comando *git push*), onde estarão as versões oficiais do projeto.

O resultado é imediatamente refletido na página do GitHub, como podemos observar a partir da **Figura 5**.

Por enquanto, estamos trabalhando diretamente com o versionamento e qualquer versão do produto a ser disponibilizado para um usuário envolveria trabalho manual. Entretanto, DevOps é toda uma cadeia de processos, da qual também faz parte a automatização do processo de entrega. Chegaremos lá, em um próximo artigo. Por enquanto, no entanto, vamos nos ater ao processo de desenvolvimento e submissão de todo o conteúdo a um repositório com controle de versão. Para efeitos de ‘roadmap’, para que o leitor entenda onde chegaremos, teremos ainda:

- Configuração de um servidor para que a integração do código seja contínua e automática, a partir de cada submissão de código que fizermos ao repositório central (remoto);

### Listagem 3. Processo de submissão de código aos repositórios Git local e remoto.

```
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ pwd  
/Users/pedrobrigatto/Personal/Projects/DevOpsArticle/devmedia_devops_series  
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ ls -la  
total 24  
drwxr-xr-x 6 pedrobrigatto staff 204 Nov 7 12:18 .  
drwxr-xr-x 6 pedrobrigatto staff 204 Nov 7 12:18 ..  
-rw-r--r--@ 1 pedrobrigatto staff 6148 Nov 7 12:11 .DS_Store  
drwxr-xr-x 13 pedrobrigatto staff 442 Nov 7 11:47 .git  
-rw-r--r-- 1 pedrobrigatto staff 192 Nov 7 12:05 README.md  
drwxr-xr-x 8 pedrobrigatto staff 272 Nov 7 12:12 devops  
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ git add .  
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ git commit -m  
"Primeira versão do código salva no repositório"  
[master da3e7f3] Primeira versão do código salva no repositório  
27 files changed, 648 insertions(+)  
create mode 100644 devops/.project  
create mode 100644 devops/.settings/org.eclipse.m2e.core.preferences  
create mode 100644 devops/model/.classpath  
create mode 100644 devops/model/.gitignore  
create mode 100644 devops/model/.project  
create mode 100644 devops/model/.settings/org.eclipse.jdt.core.preferences  
create mode 100644 devops/model/.settings/org.eclipse.m2e.core.preferences  
create mode 100644 devops/model/.springBeans  
create mode 100644 devops/model/pom.xml  
create mode 100644 devops/model/src/main/java/com/devmedia/articles/development/CalculadoraDeResistencia.java  
create mode 100644 devops/model/src/main/java/com/devmedia/articles/development/FabricaDeResistores.java  
create mode 100644 devops/model/src/main/java/com/devmedia/articles/development/MapaDeCores.java  
create mode 100644 devops/model/src/main/java/com/devmedia/articles/de-
```

```
vops/model/Resistor.java  
create mode 100644 devops/model/src/main/java/com/devmedia/articles/development/exceptions/NumerodeFaixasIncorreto.java  
create mode 100644 devops/model/src/main/resources/devops_model-context.xml  
create mode 100644 devops/model/src/test/java/com/devmedia/articles/devops/model/FabricaDeResistoresTest.java  
create mode 100644 devops/pom.xml  
create mode 100644 devops/standalone-cli/.classpath  
create mode 100644 devops/standalone-cli/.gitignore  
create mode 100644 devops/standalone-cli/.project  
create mode 100644 devops/standalone-cli/.settings/org.eclipse.jdt.core.preferences  
create mode 100644 devops/standalone-cli/.settings/org.eclipse.m2e.core.preferences  
create mode 100644 devops/standalone-cli/.springBeans  
create mode 100644 devops/standalone-cli/pom.xml  
create mode 100644 devops/standalone-cli/src/main/java/com/devmedia/articles/development/Main.java  
create mode 100644 devops/standalone-cli/src/main/java/com/devmedia/articles/development/CalculadoraResistores.java  
create mode 100644 devops/standalone-cli/src/main/resources/devops_standalone-client-context.xml  
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$ git push origin  
master  
Counting objects: 59, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (39/39), done.  
Writing objects: 100% (59/59), 8.69 KiB | 0 bytes/s, done.  
Total 59 (delta 6), reused 0 (delta 0)  
To https://github.com/pedrobrigatto/devmedia_devops_series.git  
11e9b05..da3e7f3 master -> master  
Pedro-Brigattos-MacBook:devmedia_devops_series pedrobrigatto$
```

# Conheça os papéis do desenvolvedor na cultura DevOps

- Configuração do processo de entrega contínua de builds do projeto, disponibilizando o software em um local pré-determinado.

## Explorando o código-fonte e demais artefatos do projeto

É chegado o momento de explorar o conteúdo do sistema que construiremos neste arti-

Repositório que será usado nos artigos que cobrem DevOps na Java Magazine (edições a serem informadas quando os artigos forem aprovados e publicados pela revista) — Edit

2 commits 1 branch 0 releases 1 contributor

Branch: master devmedia\_devops\_series / +

pedrobrigatto Primeira versão do código salva no repositório Latest commit da3e7f3 4 minutes ago

devops Primeira versão do código salva no repositório 4 minutes ago

README.md Initial commit 4 days ago

README.md

## devmedia\_devops\_series

Repositório que será usado nos artigos que cobrem DevOps na Java Magazine (edições a serem informadas quando os artigos forem aprovados e publicados pela revista)

Figura 5. Repositório criado na conta do GitHub

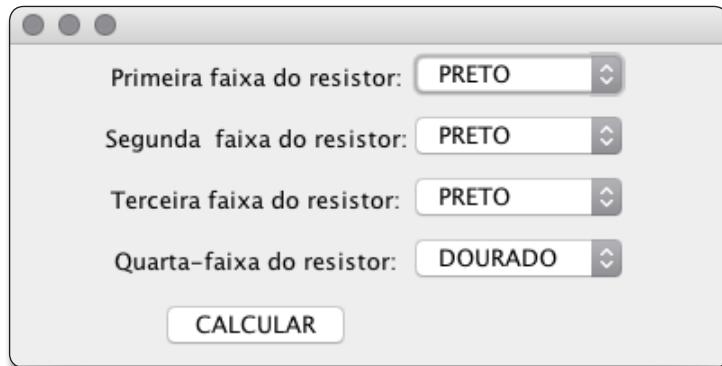


Figura 6. Tela da aplicação-guia desenvolvida para o artigo, para cálculo de resistências



Figura 7. Estrutura do projeto na IDE Eclipse

go. Iniciemos, então, observando a **Figura 6**. Esta é – até o presente momento – a única tela do sistema. Por ela, podemos dizer qual é a configuração de cores das faixas de um resistor qualquer e, ao pressionarmos o botão *CALCULAR*, descobrir o seu valor nominal e também a sua precisão.

A ideia do projeto é simplesmente ter um material para poder explorar as atividades típicas de DevOps: desenvolvimento, revisão de código, gestão de projeto, controle de versão, implantação, testes automatizados, dentre outros.

A estrutura do projeto está ilustrada na **Figura 7**. Trata-se de um projeto modularizado, gerenciado por meio do Maven, e que utiliza o Spring principalmente para o controle de injecção de dependências no código. Além disso, toda a configuração comum a todos os módulos será concentrada no POM do projeto, apresentado na **Listagem 4**. Ao observarmos esta listagem, vemos que a única característica comum aos dois módulos, neste momento, é a utilização da biblioteca básica do Spring para injecção de dependências.

Passemos, agora, por cada módulo para entender como o sistema funciona.

## A camada de modelo de dados

A análise da camada de modelo de dados começa na **Listagem 5**. Nela, vemos a declaração de um objeto do tipo bean do tipo `br.com.devmedia.articles.devops.model.api.FabricaDeResistores`, que será responsável por toda a lógica de montagem de resistores a partir de um conjunto de faixas fornecidas pelo usuário da solução.

Esta classe de objetos é, na prática, a implementação de referência para uma interface definida na camada de modelo, do tipo `br.com.devmedia.articles.devops.model.api.CalculadoraDeResistencia`, cujo código-fonte podemos observar na **Listagem 6**. Note que há apenas um método declarado nesta interface, que tem por função receber um conjunto variável de faixas e retornar um resistor com os valores nominais estabelecidos ou, em caso de alguma inconsistência nos parâmetros fornecidos, representá-la na forma de uma exceção, do tipo `br.com.devmedia.articles.devops.model.exceptions.NumeroDeFaixasIncorrecto`.

**Listagem 4.** pom.xml – Configuração do projeto Maven.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>br.com.devmedia.articles</groupId>
  <artifactId>devops</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>DevOps</name>
  <description>Projeto desenvolvido para explorar ferramentas e conceitos de DevOps</description>

  <properties>
    <spring.version>4.2.2.RELEASE</spring.version>
  </properties>

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

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

**Listagem 5.** devops\_model-context.xml – Configuração dos beans da camada de modelo de dados.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="fabricaResistores" class="br.com.devmedia.articles.devops.model.FabricaDeResistores" />
</beans>
```

Esta implementação de referência que mencionamos está apresentada na **Listagem 7**. Observe que a lógica consiste simplesmente em avaliar a sequência de cores fornecida de acordo com a seguinte interpretação:

- Primeiro valor: cor que resultará na escolha da dezena do valor de base do resistor;
- Segundo valor: cor que resultará na escolha da unidade do valor de base do resistor;
- Terceiro valor: define a potência de 10 que deverá ser utilizada no cálculo;
- Quarto valor: indica a precisão (intervalo de variação) inerente ao material de que o resistor é feito.

O valor de cada cor será fornecido, em tempo de execução, por um objeto do tipo **br.com.devmedia.articles.devops.model.MapaDeCores**, cuja implementação podemos ver na **Listagem 8**.

**Listagem 6.** CalculoDeResistencia – Interface para o cálculo de valores nominais de resistores.

```
package br.com.devmedia.articles.devops.model;

import br.com.devmedia.articles.devops.model.exceptions.NumeroDeFaixasIncorreto;

public interface CalculadoraDeResistencia {

    /**
     * Constrói representações de resistores baseado nas faixas fornecidas.
     *
     * @param faixas Faixas gravadas no corpo do resistor físico
     * @return Especificações nominais do resistor em questão
     */
    Resistor montarResistor (String ... faixas) throws NumeroDeFaixasIncorreto;
}
```

**Listagem 7.** FabricaDeResistores – Implementação de referência para o cálculo de valores nominais de resistores.

```
package br.com.devmedia.articles.devops.model.api;

import br.com.devmedia.articles.devops.model.MapaDeCores;
import br.com.devmedia.articles.devops.model.Resistor;
import br.com.devmedia.articles.devops.model.exceptions.CorDeBaseInvalida;
import br.com.devmedia.articles.devops.model.exceptions.CorDePrecisaoInvalida;
import br.com.devmedia.articles.devops.model.exceptions.NumeroDeFaixasIncorreto;

public class FabricaDeResistores implements CalculadoraDeResistencia {

    /**
     * Este método assume que as três primeiras faixas passadas compõem o valor nominal do
     * resistor, enquanto a quarta faixa compõe o intervalo que este resistor varia.
     *
     * @param valores Faixas coloridas no corpo do resistor
     * @return A resistência em si, composta por valor nominal e faixa de variação
     */
    public Resistor montarResistor (String ... valores)
        throws NumeroDeFaixasIncorreto, CorDePrecisaoInvalida, CorDeBaseInvalida {

        if (valores == null || valores.length != 4) {
            throw new NumeroDeFaixasIncorreto(
                "O número de faixas fornecidas não corresponde às 4 faixas esperadas");
        }

        StringBuilder builder = new StringBuilder();
        String token = String.valueOf(MapaDeCores.lerValorBase(valores[0]));

        builder.append(token.substring(0, token.indexOf(".")));

        token = String.valueOf(MapaDeCores.lerValorBase(valores[1]));
        builder.append(token.substring(0, token.indexOf(".")));

        Double base = Double.parseDouble(builder.toString());
        Double powerOfTen = (Double) MapaDeCores.lerValorBase(valores[2]);

        Double precisao = (Double) MapaDeCores.lerPrecisao(valores[3]);

        return new Resistor(new Double(base * Math.pow(10, powerOfTen)), precisao);
    }
}
```

# Conheça os papéis do desenvolvedor na cultura DevOps

Para compor o valor numérico correspondente a cada cor de faixa, esta será a classe acessada. Como podemos observar, há mapas separados para a identificação dos numerais referentes à composição da base, da potência de 10 e da porcentagem de variação que o resistor poderá apresentar quando em funcionamento em um circuito eletro-eletrônico.

A camada de modelo de dados tem sua análise finalizada com um elemento essencial e que, embora negligenciado por um grande número de desenvolvedores, tem vital importância para qualquer lógica de negócio: os testes unitários.

Para analisar este último elemento, voltemos ao aspecto cultural de DevOps. A 'frente' Dev de um ambiente DevOps só será bem-sucedida caso a elaboração de todas as unidades lógicas do sistema sejam criadas com absoluta responsabilidade. De nada adiantaria termos um arsenal espetacular de ferramentas que automatizam todas as atividades essenciais do processo de desenvolvimento e entrega de software se, por outro lado, o que estamos entregando apresente um nível de qualidade abaixo das expectativas (de todos os stakeholders, incluindo a própria equipe de criação).

Mas como é possível garantir qualidade naquilo que é produzido? Qual seria a ferramenta a ser usada, de modo que tenhamos a convicção de que os requisitos são cobertos em sua totalidade?

A resposta é uma só: testes unitários. Embora muitos desenvolvedores – experientes, inclusive – torçam o nariz para esta afirmação sempre que ela é proferida, testes unitários são a garantia mais clara de que uma classe funciona conforme o esperado. A grande razão pela qual se encontra, ainda hoje, uma certa resistência a testes unitários (acreditem, isso é real) é o fato desta atividade ter sido, ao longo dos anos, tão negligenciada e mal executada.

Testes unitários deveriam ser muito mais que blocos de código que visam validar saídas esperadas de acordo com entradas pré-determinadas. Testes unitários precisam, imediatamente, ser encarados como o **único** recurso que nos permitirá garantir, a partir de dados concretos, que o comportamento dos objetos de uma determinada classe funciona **em absoluto alinhamento** com os requisitos mapeados junto ao cliente.

Sendo, inclusive, um pouco extremista, diríamos que os testes unitários seriam a técnica mais convincente de, junto ao próprio cliente, validar os requisitos levantados. Se, ao final de toda uma bateria de testes executados sobre uma determinada unidade lógica do sistema, garante-se que todos os resultados obtidos são exatamente aqueles que eram esperados, qual seria a reação de um cliente senão a satisfação de observar que seus anseios foram totalmente captados pela equipe de desenvolvimento?

**Listagem 8.** MapaDeCores – Classe utilitária para fornecedor o valor correspondente a cada cor.

```
package br.com.devmedia.articles.devops.model;

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

import br.com.devmedia.articles.devops.model.exceptions.CorDeBaseInvalida;
import br.com.devmedia.articles.devops.model.exceptions.CorDePrecisaInvalida;

public final class MapaDeCores {

    public static final String PRETO = "PRETO";
    public static final String MARROM = "MARROM";
    public static final String VERMELHO = "VERMELHO";
    public static final String LARANJA = "LARANJA";
    public static final String AMARELO = "AMARELO";
    public static final String VERDE = "VERDE";
    public static final String AZUL = "AZUL";
    public static final String VIOLETA = "VIOLETA";
    public static final String CINZA = "CINZA";
    public static final String BRANCO = "BRANCO";
    public static final String DOURADO = "DOURADO";
    public static final String PRATEADO = "PRATEADO";

    private static Map<String, Double> codigosDeValor;
    private static Map<String, Double> codigosDelPrecisao;

    static {
        inicializarMapaDeCores();
    }

    private MapaDeCores () {}

    private static void inicializarMapaDeCores() {
        codigosDeValor = new HashMap<String, Double>();
        codigosDeValor.put(PRETO, 0.0);
        codigosDeValor.put(MARROM, 1.0);
        codigosDeValor.put(VERMELHO, 2.0);
        codigosDeValor.put(LARANJA, 3.0);
        codigosDeValor.put(AMARELO, 4.0);
        codigosDeValor.put(VERDE, 5.0);
        codigosDeValor.put(AZUL, 6.0);
        codigosDeValor.put(VIOLETA, 7.0);
        codigosDeValor.put(CINZA, 8.0);
        codigosDeValor.put(BRANCO, 9.0);

        codigosDelPrecisao = new HashMap<String, Double>();
        codigosDelPrecisao.put(DOURADO, 0.05);
        codigosDelPrecisao.put(PRATEADO, 0.1);
    }

    public static Number lerPrecisao(String chave) throws CorDePrecisaInvalida {
        Number valor = codigosDelPrecisao.get(chave);

        if (valor == null) {
            throw new CorDePrecisaInvalida(
                "A cor fornecida não é válida para identificação da precisão de material");
        }

        return valor;
    }

    public static Number lerValorBase(String chave) throws CorDeBaseInvalida {
        Number valor = codigosDeValor.get(chave);

        if (valor == null) {
            throw new CorDeBaseInvalida(
                "A cor fornecida não é válida para composição da resistência");
        }

        return valor;
    }
}
```

**Listagem 9.** FabricaDeResistoresTest.java – Testes unitários para validação dos requisitos de cálculo de resistências.

```
package br.com.devmedia.articles.devops.model;

import org.junit.Assert;
import org.junit.Test;

import br.com.devmedia.articles.devops.model.api.FabricaDeResistores;
import br.com.devmedia.articles.devops.model.exceptions.CorDeBaseInvalida;
import br.com.devmedia.articles.devops.model.exceptions.CorDePrecisaInvalida;
import br.com.devmedia.articles.devops.model.exceptions.NumeroDeFaixasIncorrecto;

public class FabricaDeResistoresTest {

    @Test
    public void testMontagemResistorCom4Faixas()
    throws NumeroDeFaixasIncorrecto, CorDePrecisaInvalida, CorDeBaseInvalida {
        FabricaDeResistores fabrica = new FabricaDeResistores();

        Resistor resistor = fabrica.montarResistor(
            MapaDeCores.MARRON, MapaDeCores.PRETO,
            MapaDeCores.PRETO, MapaDeCores.PRATEADO);
        Assert.assertTrue(resistor.getValorBase() == 10.0);
        Assert.assertTrue(resistor.lerMaiorValor() == 11.0);
        Assert.assertTrue(resistor.lerMenorValor() == 9.0);
    }

    @Test(expected = NumeroDeFaixasIncorrecto.class)
    public void testMontagemResistorComFaixasSobrando()
    throws NumeroDeFaixasIncorrecto, CorDePrecisaInvalida, CorDeBaseInvalida {
        FabricaDeResistores fabrica = new FabricaDeResistores();
        fabrica.montarResistor(MapaDeCores.MARRON, MapaDeCores.PRETO,
            MapaDeCores.PRETO);
    }

    @Test(expected = CorDeBaseInvalida.class)
    public void testMontagemResistorComFaixasFaltando()
    throws NumeroDeFaixasIncorrecto, CorDePrecisaInvalida, CorDeBaseInvalida {
        FabricaDeResistores fabrica = new FabricaDeResistores();
        fabrica.montarResistor(MapaDeCores.MARRON, MapaDeCores.PRETO,
            MapaDeCores.PRETO, MapaDeCores.PRATEADO);
    }

    @Test(expected = CorDePrecisaInvalida.class)
    public void testCalculoComCorDeBaseInvalida()
    throws NumeroDeFaixasIncorrecto, CorDePrecisaInvalida, CorDeBaseInvalida {
        FabricaDeResistores fabrica = new FabricaDeResistores();
        fabrica.montarResistor(
            MapaDeCores.PRATEADO, MapaDeCores.PRETO,
            MapaDeCores.PRETO, MapaDeCores.PRATEADO);
    }

    @Test(expected = CorDePrecisaInvalida.class)
    public void testCalculoComCorDePrecisaInvalida()
    throws NumeroDeFaixasIncorrecto, CorDePrecisaInvalida, CorDeBaseInvalida {
        FabricaDeResistores fabrica = new FabricaDeResistores();
        fabrica.montarResistor(
            MapaDeCores.MARRON, MapaDeCores.PRETO,
            MapaDeCores.PRETO, MapaDeCores.PRETO);
    }
}
```

A **Listagem 9** tem o objetivo de tornar mais visual e, portanto, claro, o que acabamos de apresentar. Esta é a classe de testes unitários que tem por função verificar o comportamento da classe **FabricaDeResistores**.

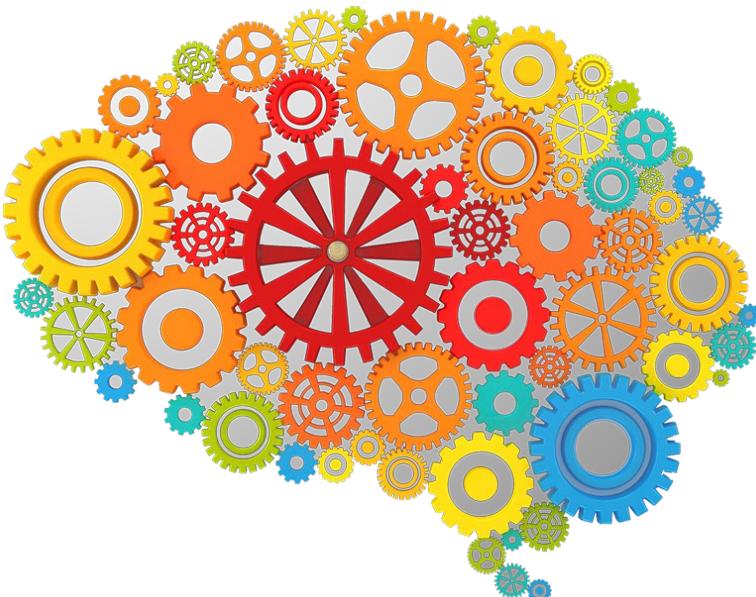
Nem tudo, naturalmente, precisa ser associado a testes unitários. Este é um grande equívoco cometido com relativa frequência por equipes de desenvolvimento que, forçadas a atingir métricas um tanto quanto questionáveis, associadas a débito técnico (quando na verdade não tem qualquer razão de sê-lo), acabam escrevendo código que, ao invés de efetivamente validar o conteúdo do código-fonte produzido, prestam-se apenas a garantir que a cobertura do código atinge uma porcentagem pré-estabelecida como aceitável.

Este é um ponto, inclusive, que precisa ser muito bem estabelecido para que projetos de software não acabem sendo prejudicados por pressões desnecessárias. No entanto, até por sua natureza polêmica, é um assunto sobre o qual não nos prolongaremos por questão de espaço.

O principal requisito existente para o cálculo de resistores é que o número de cores que deve ser fornecida para a fábrica de resistores não ultrapasse 4. Além disso, é importante também garantir que a quarta cor passada seja uma dentre duas possíveis: prateada ou dourada. Da mesma forma, as cores passadas para a composição do valor de base do resistor não poderão ser nenhuma daquelas utilizadas para a composição de seu intervalo de precisão. Durante a implementação do código-fonte do sistema, esses requisitos foram mapeados de acordo com a **Tabela 2**.

Condição/situação levantada	Tipo de objeto para representar
Número de faixas passado para o cálculo do resistor não é 4	NumeroDeFaixasIncorrecto
Cor da faixa referente ao intervalo de precisão do resistor não é válida	CorDePrecisaInvalida
Cor de uma das faixas referentes ao valor de base ou à potência de 10 não é válida	CorDeBaseInvalida

**Tabela 2.** Mapeamento de exceções de acordo com os requisitos do sistema



# Conheça os papéis do desenvolvedor na cultura DevOps

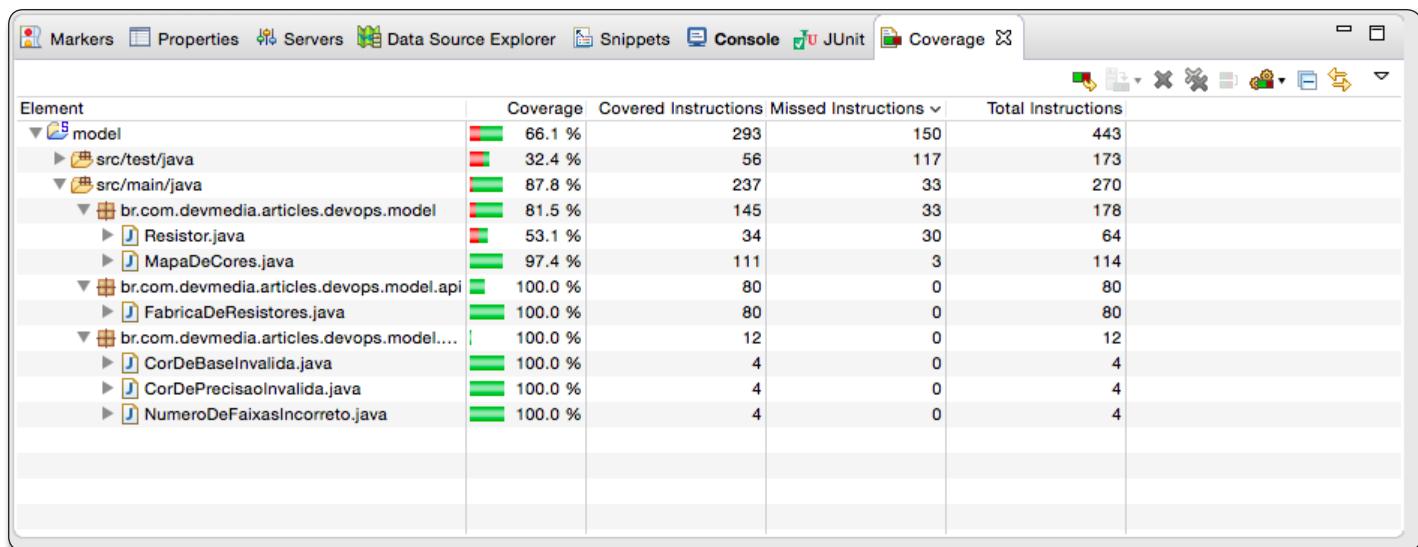


Figura 8. Cobertura de código através do plug-in EclEmma

Na classe de testes unitários, perceba que há ao menos um método para exercitar cada uma dessas situações, bem como a do cálculo de uma resistência com um conjunto válido de parâmetros. Da forma como elaboramos este código, qualquer pessoa que venha a executar este artefato de testes unitários constatará que, em cada cenário coberto, os resultados estão em plena conformidade com as expectativas. Para verificar isto na prática, é importante que o leitor execute esses testes, estudando a sua lógica e, eventualmente, trocando os valores de entrada para avaliar o comportamento geral dos testes propostos.

Outro recurso que configuramos no início do artigo será, agora, bastante útil. Trata-se do plug-in para análise de cobertura de código, o EclEmma. Lembra-se dele?

Para executá-lo, basta que selezionemos a classe de teste e, em seguida, clicando com o botão direito do mouse, açãojemos a

opção *Coverage as*. O resultado pode ser analisado a partir da Figura 8. Veja que, para a unidade lógica que esta classe de testes unitários se propõe a validar – *FabricaDeResistores* –, a cobertura obtida foi de 100%. Como efeito colateral, é possível também ver que outras classes utilizadas por esta fábrica de resistores foram também cobertas em uma porcentagem significativa.

## A camada de apresentação

A segunda e última camada é a de apresentação. Este primeiro módulo gráfico do projeto oferece uma tela simples, desenvolvida em Swing/AWT, exibida na Figura 6.

Antes de passarmos à análise deste código, porém, vamos avaliar o conteúdo da Listagem 10. Este é o arquivo descritor do módulo ‘standalone’ e, no momento, o único ponto que

### Listagem 10. pom.xml – Configuração do módulo cliente (standalone) do projeto.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>br.com.devmedia.articles</groupId>
    <artifactId>devops</artifactId>
    <version>0.0.1-SNAPSHOT</version>
  </parent>
  <artifactId>standalone-cli</artifactId>
  <name>Standalone CLI</name>
  <description>Cliente standalone implementado em Swing</description>

  <dependencies>
    <dependency>
      <groupId>br.com.devmedia.articles</groupId>
      <artifactId>model</artifactId>
      <version>${project.version}</version>
    </dependency>
  </dependencies>
</project>
```



merece destaque é a inclusão da camada de modelo de dados como dependência. É a partir, portanto, do serviço de cálculo de resistência disponível no modelo de dados que esta tela poderá exibir os dados do resistor calculado a partir dos parâmetros de entrada do usuário.

Na **Listagem 11** mostramos o código da classe **br.com.devmedia.articles.devops.view.CalculadoraResistores**, que representa a interface gráfica pela qual o usuário interagirá com o sistema. Esta é uma tela composta por quatro caixas de seleção, a partir

das quais o usuário selecionará as cores de cada uma das quatro faixas que compõem o resistor. Os cliques no botão **CALCULAR** dispararão uma chamada ao serviço de cálculo de resistência, implementado na camada de modelo e referenciado no módulo *standalone* a partir de sua interface, e o resultado do processamento é finalmente exibido ao usuário em uma janela de pop-up.

Para interfaces de usuário, como é o caso do conteúdo do módulo *standalone* descrito nesta seção, a escrita de testes unitários nem sempre fará sentido. Outras formas de teste, como os de interface,

**Listagem 11.** CalculadoraResistores – Tela da aplicação gráfica (standalone) para cálculo de resistores.

```
package br.com.devmedia.articles.devops.view;

//imports omitidos...



```

são mais recomendadas. No cenário web, um framework bastante popular utilizado no desenvolvimento de testes deste tipo é o Selenium.

Neste nosso artigo, não teremos nenhuma cobertura de testes unitários associado à tela do sistema *standalone*. No próximo artigo, entretanto, introduziremos um módulo web exatamente para atacar esta frente de testes e trazer ao leitor, assim, um exemplo prático de como este framework funciona na prática, em um contexto automatizado de DevOps.

A **Listagem 12**, por fim, nos mostra como o sistema é executado. Por ela, vemos que o contexto Spring configurado no módulo *standalone* é carregado e, então, a tela é construída, carregada e exibida ao usuário.

**Listagem 12.** Main.java – Entry point da aplicação standalone, a partir de onde a execução do projeto se inicia.

```
package br.com.devmedia.articles.devops;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import br.com.devmedia.articles.devops.view.CalculadoraResistores;

public class Main {

    private static ApplicationContext applicationContext;

    public static void main (String [] args) {
        applicationContext = new ClassPathXmlApplicationContext
        ("devops_standalonecli-context.xml");
        ((CalculadoraResistores) applicationContext.getBean(CalculadoraResistores.class))
        .build();
    }
}
```

Podemos, seguramente, afirmar que DevOps é o novo paradigma de desenvolvimento de software. Um fato que reforça esta afirmação é que a cada dia é possível observar muitas empresas se dedicando intensamente para enquadrar seus processos de desenvolvimento e operações nesta nova filosofia, com o objetivo de torná-los mais eficientes interna e externamente.

A quantidade de ferramentas disponível – inclusive gratuitas – é imensa, para todas as atividades típicas em um contexto de DevOps. Isto é uma vantagem e, ao mesmo tempo, exige de nós

uma avaliação cuidadosa na hora de escolher o nosso ‘kit’. Alguns dos principais fatores que devem ser avaliados são a estabilidade, o suporte – da comunidade ou, no caso de ferramentas comerciais, da empresa – e o nível de integração de cada produto/serviço com os demais.

## Autor



Pedro E. Cunha Brigatto

[pedrobrigatto@gmail.com](mailto:pedrobrigatto@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-fonte do projeto.

[https://github.com/pedrobrigatto/devmedia\\_devops\\_series](https://github.com/pedrobrigatto/devmedia_devops_series)

### A história por trás do nome DevOps

<https://blog.newrelic.com/2014/05/16/devops-name/>

### Manual do Git

<https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes>

### Palestra sobre Artesanato de Software

<http://www.infoq.com.br/presentations/entrega-continua>

### Palestra sobre DevOps

<https://www.youtube.com/watch?v=CXEQ1dNp1Xo>

### Página oficial da ferramenta Eclemma (cobertura de código)

<http://eclemma.org>

### Tabela de cores para cálculo de resistência

[http://www.audioacustica.com.br/exemplos/Valores\\_Resistores/Calculadora\\_Ohms\\_Resistor.html](http://www.audioacustica.com.br/exemplos/Valores_Resistores/Calculadora_Ohms_Resistor.html)

## Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)

Ajude-nos a manter a qualidade da revista!



# Testes unitários, de integração e de aceitação na prática

Como implementar esses diferentes tipos de testes com as ferramentas Gradle, JUnit, Hamcrest e Concordion

No artigo “Dominando os tipos de testes automatizados”, publicado na Java Magazine 146, os conceitos relacionados a testes automatizados foram apresentados, considerando em especial os testes unitários, de integração e de aceitação. Além dessa análise, foram exploradas também as relações de testes automatizados com código legado e com times ágeis, bem como o uso de práticas como Test-Driven Development e Behavior-Driven Development. Contudo, o artigo não apresentou exemplos reais de como os conceitos poderiam ser aplicados.

Desse modo, neste artigo abordaremos como utilizar os conceitos previamente estudados em código Java, por meio da construção de um projeto exemplo. O intuito é demonstrar como testes unitários, de integração e de aceitação podem ser implementados, tendo como base suas características essenciais: teste unitário como um validador de implementação em termos de objetos sem dependências externas; teste de integração como validador de implementações que se relacionam com recursos (tais como banco de dados); e testes de aceitação como um validador de funcionalidades do cliente (descritas por meio da técnica de User Stories).

## Exemplificando com Java

A partir de agora será criado um pequeno sistema para demonstração dos diferentes tipos de testes que foram analisados. Para isso, o sistema deve possuir alguma lógica a ser validada (para testes unitários), persistência em banco de dados (para testes de integração) e *User*

## Fique por dentro

Este artigo apresenta a implementação de três tipos de testes automatizados na prática. Para isso, um projeto exemplo será criado e nele serão aplicados os testes unitários, de integração e de aceitação. Como um importante complemento, as técnicas de Test-Driven Development e Behavior-Driven Development serão empregadas na construção dos testes unitários e de aceitação, respectivamente. A adoção de testes permite identificar problemas assim que os mesmos são inseridos no código e é mais um importante elemento para se alcançar um design mais enxuto, principalmente quando optamos pelo TDD e pela automatização.

*Stories* (para testes de aceitação). Como os testes de aceitação estão vinculados apenas à camada de domínio (back-end da aplicação), não será necessária a criação de telas, da camada de visão.

## Estrutura inicial da aplicação

A estrutura da aplicação começará com elementos básicos, e conforme a implementação for evoluindo, novas funcionalidades serão adicionadas. Para o gerenciamento de pacotes e realização de builds, adotaremos o Gradle. Sendo assim, é necessário tê-lo instalado em seu ambiente. Na seção **Links** você encontrará o endereço para download da última versão.

A **Listagem 1** apresenta uma lista de comandos para criação de um projeto para os usuários da IDE IntelliJ. Estes comandos definem a estrutura de diretórios do projeto, adicionam o plugin *idea* ao arquivo principal do Gradle, *build.gradle*, e então executam o Gradle, que por consequência executa o plugin, o que gera os

# Testes unitários, de integração e de aceitação na prática

arquivos de projeto necessários à importação pela IDE. Caso utilize o Eclipse, o comando da **Listagem 2** deve ser utilizado, de forma análoga. Por sua vez, caso deseje criar a estrutura do projeto independentemente da IDE, execute o comando da **Listagem 3**.

## Listagem 1. Criação da estrutura do projeto para IntelliJ.

```
01 mkdir -p numerolog/src/main/java/br/com/devmedia numerolog/src/
main/resources
02     numerolog/src/test/java/br/com/devmedia numerolog/src/test/
resources
03     && printf"apply plugin:java\napply plugin:idea"
04         > numerolog/build.gradle && cd numerolog && gradle idea
```

## Listagem 2. Criação da estrutura do projeto para Eclipse.

```
01 mkdir -p numerolog/src/main/java/br/com/devmedia numerolog/src/
main/resources
02     numerolog/src/test/java/br/com/devmedia numerolog/src/test/
resources
03     && printf"apply plugin:java\napply plugin:eclipse"
04         > numerolog/build.gradle && cd numerolog && gradle eclipse
```

## Listagem 3. Criação da estrutura do projeto, independente de IDE.

```
01 mkdir -p numerolog/src/main/java/br/com/devmedia numerolog/src/
main/resources
02     numerolog/src/test/java/br/com/devmedia numerolog/src/test/resources
03     && touch numerolog/build.gradle
```

Feito isso, a estrutura do projeto será criada contendo os diretórios para o código de produção e código de testes, bem como o arquivo de configuração do Gradle.

Nas **Listagens 1 e 2**, o plugin da IDE é adicionado ao *build.gradle*, e o comando *gradle <nome da IDE>* cria os arquivos de configuração para que as IDEs possam importar o projeto corretamente. Na **Listagem 3**, o arquivo *build.gradle* também é criado, mas, como esperado, sem especificar um plugin para qualquer IDE.

Importe o projeto em sua IDE (se tiver escolhido executar os comandos da **Listagem 1** ou da **Listagem 2**) e edite o arquivo *build.gradle* para adicionar mais detalhes de configuração ao projeto, conforme o conteúdo apresentado na **Listagem 4**.

## Listagem 4. Conteúdo do arquivo build.gradle.

```
01 apply plugin:java
02 apply plugin:idea
03
04     sourceCompatibility = 1.8
05     targetCompatibility = 1.8
06
07 repositories {
08     mavenCentral()
09 }
```

As primeiras linhas definem os plugins do Java e da IDE escolhida para uso. Já nas linhas 4 e 5, é especificada a versão do Java a ser adotada no projeto (neste caso, o Java 8). E as linhas de 7 a 9 declaram que o repositório central do Maven será utilizado como o repositório de dependências do projeto.

## Definição do problema e das User Stories

Um requisito simples que foi escolhido para trabalharmos em nosso projeto exemplo é o da redução de letras e datas a algarismos decimais, de 0 a 9, por somas e divisões sucessivas. O sistema aqui proposto deve realizar o cálculo numerológico dos dados do usuário, como nome e data de nascimento. Para implementarmos este requisito, codificaremos um algoritmo para representação do nome e da data de nascimento de usuários em números, a ser testado com testes unitários. Para isso, o usuário será representado pela entidade **Usuario** e as ações do CRUD relacionado serão implementadas de forma que sejam verificadas com testes de integração. Por fim, utilizaremos *User Stories* (ou histórias de usuário, conforme conceitualizadas no outro artigo sobre testes) para escrever os testes de aceitação deste cálculo.

Começaremos definindo algumas User Stories relacionadas ao cálculo numerológico, sendo uma para o cálculo do nome e outra para o cálculo da data de nascimento:

### User Story: Usuário obtém a numerologia de seu nome completo

DADO um usuário com o nome completo "Maria da Silva"  
QUANDO o usuário calcula a numerologia do seu nome completo  
ENTAO o número obtido é 2

### User Story: Usuário obtém a numerologia de sua data de nascimento

DADO um usuário nascido em 07/10/1988  
QUANDO o usuário calcula a numerologia da sua data de nascimento  
ENTAO o número obtido é 7

## Escrevendo o modelo de domínio com TDD e testes unitários

Para que sejam escritos os testes unitários, as dependências para os mesmos devem ser declaradas no projeto. Na **Listagem 5** são apresentadas as dependências necessárias para que as classes e métodos de testes unitários possam ser utilizadas pela aplicação.

Como pode ser verificado, nas linhas 11 a 15 são adicionadas as dependências para execução dos testes. Dentre elas, o JUnit é o framework padrão para todos os testes unitários automatizados deste artigo. O Hamcrest, por sua vez, é a biblioteca que oferece uma sintaxe mais clara para a escrita dos testes (unitários, de integração e de aceitação), com métodos com nomes mais descriptivos e próximos da língua inglesa (por exemplo, *assertThat(object) is(true)* é mais descriptiva, ou fluente, do que *assertTrue(objeto)*). Já a biblioteca commons-lang3, da Apache, será empregada porque contém um objeto **Pair**, cuja funcionalidade de associar dois objetos distintos (por necessidade de implementação, a ser vista nas próximas listagens) será utilizada nos testes.

A partir disso, podemos escrever o código para calcular a numerologia do nome e escrever testes que o validem. Como esse código ainda não foi implementado, a técnica de TDD pode ser aplicada a partir deste ponto, respeitando sempre os ciclos nela descritos (vide livro *TDD By Example*, de Kent Beck).

**Listagem 5.** Arquivo build.gradle com as dependências para testes unitários.

```
01 apply plugin:'java'
02 apply plugin:'idea'
03
04     sourceCompatibility = 1.8
05     targetCompatibility = 1.8
06
07 repositories {
08     mavenCentral()
09 }
10
11 dependencies {
12     testCompile 'junit:junit:4.12'
13     testCompile 'org.apache.commons:commons-lang3:3.4'
14     testCompile 'org.hamcrest:hamcrest-all:1.3'@
15 }
```

**Listagem 6.** Classe de teste unitário.

```
01 package br.com.devmedia;
02 import org.apache.commons.lang3.tuple.Pair;
03 import org.junit.Before;
04 import org.junit.Test;
05 import org.junit.experimental.theories.DataPoints;
06 import org.junit.experimental.theories.FromDataPoints;
07 import org.junit.experimental.theories.Theories;
08 import org.junit.experimental.theories.Theory;
09 import org.junit.runner.RunWith;
10
11 import java.time.LocalDate;
12 import java.time.Month;
13
14 import static org.hamcrest.MatcherAssert.assertThat;
15 import static org.hamcrest.core.IsEqual.equalTo;
16
17 /**
18 * Considere a tabela abaixo para validação dos cálculos:
19 *
20 * 1 2 3 4 5 6 7 8 9
21 * -----
22 * A B C D E F G H I
23 * J K L M N O P Q R
24 * S T U V W X Y Z
25 */
26 @RunWith(Theories.class)
27 public class UsuarioTest {
28
29     @DataPoints("numerosParaReduzirBaseDecimal")
30     public static Pair<Integer, Integer>[] numerosParaReduzirBaseDecimal =
31         new Pair[] {
32             Pair.of(27, 9), Pair.of(95, 5), Pair.of(88, 7), Pair.of(177, 6)
33         };
34
35     @DataPoints("valorNumericoComAIniciandoEmUm")
36     public static Pair<Character, Integer>[] valorNumericoComAIniciandoEmUm =
37         new Pair[] {
38             Pair.of('a', 1), Pair.of('b', 2), Pair.of('E', 5), Pair.of('H', 8),
39             Pair.of('K', 11), Pair.of('o', 15), Pair.of('r', 18), Pair.of('s', 19),
40             Pair.of('V', 22), Pair.of('z', 26),
41         };
42
43     @DataPoints("valorDoCaractereNaTabela")
44     public static Pair<Character, Integer>[] valorDoCaractereNaTabela = new Pair[] {
45         Pair.of('A', 1), Pair.of('r', 9), Pair.of('o', 6), Pair.of('Z', 8),
46         Pair.of('s', 1), Pair.of('J', 1), Pair.of('f', 6), Pair.of('X', 6)
47     };
48 }
```

Para saber o que testar, deve-se ter como base as *User Stories*, artefatos que declaram o resultado esperado do cálculo sobre os dados do usuário. O resultado final da criação do código e dos testes pode ser verificado nas **Listagens 6 e 7**, que apresentam, respectivamente, a classe de teste unitário e a classe de produção.

Visto que o JUnit trabalha com o conceito de *Runners* (classes que processam os métodos de teste), vamos explorar este importante recurso em nosso exemplo. Sendo assim, na linha 26 é declarado o uso do *runner Theories*. Este permite que os métodos de teste validem um conjunto de teorias (um cenário com múltiplos valores distintos), em vez de um cenário com apenas um valor. Neste caso, por exemplo, existem três teorias a serem testadas, conforme declarado nas linhas 29, 34 e 41, por meio da anotação **@DataPoints**.

```
46
47     private Usuario usuario;
48
49     @Before
50     public void setup() {
51         this.usuario = new Usuario("Maria da Silva", LocalDate.of(
52             1980, Month.JANUARY, 15));
53
54     @Theory
55     public void deveReduzirNumeroParaBaseDecimal(
56         @FromDataPoints("numerosParaReduzirBaseDecimal")
57         Pair<Integer, Integer> item) {
58         assertThat(usuario.reducaoParaBaseDecimal(item.getLeft()),
59                     equalTo(item.getRight()));
60     }
61
62     @Theory
63     public void deveObterValorNumericoComAIniciandoEmUm(
64         @FromDataPoints("valorNumericoComAIniciandoEmUm")
65         Pair<Character, Integer> item) {
66         assertThat(usuario.valorNumericoComAIniciandoEmUm(item.getLeft()),
67                     equalTo(item.getRight()));
68     }
69
70     @Theory
71     public void deveObterOValorDoCaractereNaTabela(
72         @FromDataPoints("valorDoCaractereNaTabela")
73         Pair<Character, Integer> item)
74     {
75         assertThat(usuario.valorNumericoDoCaractere(item.getLeft()),
76                     equalTo(item.getRight()));
77     }
78
79
80     @Test
81     public void deveCalcularNumerologiaDoNome() {
82         usuario.calcularNumerologiaDoNome();
83         assertThat(usuario.getNumeroDoNome(), equalTo(2));
84     }
85
86 }
```

# Testes unitários, de integração e de aceitação na prática

A primeira teoria verifica se os números podem ser reduzidos a algarismos decimais pela soma de suas partes, e é testada pelo método da linha 55. A segunda teoria verifica se um conjunto de letras do alfabeto está associada a um correspondente numérico válido (i.e.: A1=1, B=2 e assim por diante), e é testada pelo método definido na linha 61. E a última teoria, testada pelo método na linha 68, verifica se um dado caractere está associado ao mesmo valor da tabela apresentada nas linhas 17 a 25.

Note que cada um desses três cenários testa métodos que receberam o modificador de acesso de pacote, de forma que eles não podem ser chamados diretamente por clientes de fora deste pacote. Apenas os testes declarados nas linhas 75 e 81 chamam métodos que efetivamente podem ser executados por diferentes clientes (têm o modificador de acesso público).

A classe de produção foi planejada dessa forma (fazendo uso do modificador de acesso de pacote) para que o usuário consiga calcular apenas a numerologia do nome e da data de nascimento. Note que todas as etapas para se alcançar estes resultados não são relevantes ao usuário e por isto foram implementadas em métodos encapsulados com visibilidade restrita, chamados internamente.

Apresentada na **Listagem 7**, a classe **Usuario** é bastante simples, contendo apenas dois métodos públicos (**numerologiaDoNome()** e **numerologiaDaDataNascimento()**), bem como outros métodos internos. Observe ainda que tanto o nome quanto a data de

nascimento precisam ser informados no construtor e os métodos públicos realizam os cálculos por meio de chamadas a métodos internos, retornando por fim os números inteiros esperados.

Graças aos lambdas do Java 8, a quantidade de linhas utilizadas foi bastante reduzida. Um exemplo de uso desse recurso pode ser visto na linha 19, mais precisamente na declaração **nome.chars()**, que retorna um **IntStream**. Os streams implementam um método **map()** que permite transformar o valor de um objeto em outro valor. Neste caso, para cada caractere da **String nome**, o método **map()** aplica a transformação para um número correspondente, através da chamada ao método **valorNumericoDoCaractere()**. Neste processo, o **map()** acumula todas as chamadas que recebe, mas não as soma imediatamente. Portanto, os valores relativos a cada caractere são acumulados, como 1, 2, 5, 8, mas sua soma ainda não é realizada. Para efetuar a soma dos valores acumulados, é preciso chamar o método **sum()**.

Note que esta sintaxe não disse como fazer a transformação, com loops e somas, mas sim o que queria transformar, e que ao final os valores devem ser somados. Por isto, é classificada como uma sintaxe declarativa, e não imperativa. Essa característica de sintaxe declarativa é relacionada à programação funcional, nova opção adicionada ao Java 8 através dos lambdas.

Agora, observe na linha 9 que a nova API de data do Java 8 também foi utilizada. Uma classe que representa uma data (**LocalDate**) foi declarada no método **numerologiaDaDataNascimento()**.

**Listagem 7.** Classe de produção – **Usuario**.

```
01 package br.com.devmedia;
02
03 import java.time.LocalDate;
04 import java.util.function.UnaryOperator;
05
06 public class Usuario {
07
08     private String nome;
09     private LocalDate dataNascimento;
10    private int numeroDoNome;
11    private int numeroDataDataNascimento;
12
13    public Usuario(String nome, LocalDate dataNascimento) {
14        this.nome = nome;
15        this.dataNascimento = dataNascimento;
16    }
17
18    public void numerologiaDoNome() {
19        int somaDosCaracteresDoNome = nome.chars()
20            .map(c -> valorNumericoDoCaractere((char)
21                c)).sum();
22        this.numeroDoNome = reducaoParaBaseDecimal(somaDosCaracteresDoNome);
23    }
24
25    public void numerologiaDaDataNascimento() {
26        this.numeroDataDataNascimento = reducaoParaBaseDecimal
27            (dataNascimento.getDayOfMonth() +
28             dataNascimento.getMonthValue() + dataNascimento.getYear());
29    }
30
31    public String getNome() {
32        return nome;
33    }
34    public LocalDate getDataNascimento() {
35        return dataNascimento;
36    }
37    public int getNumeroDoNome() {
38        return numeroDoNome;
39    }
40
41    public int getNumeroDaDataNascimento() {
42        return numeroDataDataNascimento;
43    }
44    int valorNumericoDoCaractere(char caractere) {
45        return caractere == '?' ? 0 :
46            reducaoParaBaseDecimal(valorNumericoComAlniciandoEm
47                Um(caractere));
48    }
49    int valorNumericoComAlniciandoEmUm(char caractere) {
50        return Character.getNumericValue(caractere) - 9;
51    }
52
53    int reducaoParaBaseDecimal (int valor) {
54        UnaryOperator<Integer> reducao = v -> v <= 9 ? v : this.reducao
55            .apply(ParabaseDecimal(v / 10 + v
56            % 10));
57        return reducao.apply(valor);
58    }
59}
```

Assim como um dos objetivos das principais novidades, a API de Datas do Java 8 oferece melhorias que simplificam o dia a dia do desenvolvedor.

Voltando à análise do nosso código, na linha 54 foi implementada a redução para a base decimal. A partir disso, para qualquer número maior do que 9, realizam-se reduções sucessivas até que o resultado seja menor do que 9.

Temos, portanto, a implementação dos cálculos numerológicos para nome completo e data de nascimento. Ao realizar essa etapa com TDD, alguns ganhos podem ser observados, a saber:

- **Definição de um design enxuto:** a prática de TDD garante que seja implementado o design relevante para o problema em questão, relacionado apenas à necessidade corrente. Deste modo, nenhum tempo é desperdiçado na implementação de um design que pode não ser utilizado;

- **Confiança em futuras modificações:** supondo que um desenvolvedor deseja estender a funcionalidade construída (por exemplo, implementar o cálculo numerológico somando apenas as vogais do nome completo), caso não existissem os testes ele poderia se sentir intimidado e com receio de adicionar um bug ao alterar o código, que já funciona. Com os testes unitários prontos, ele pode realizar a alteração e rodar novamente os testes. Caso executem com sucesso, o desenvolvedor sabe que sua alteração não quebrou o comportamento pré-existente.

Uma boa prática consiste em rodar os testes unitários a cada alteração do código. Para isso, execute a classe de teste na sua IDE ou use o comando `gradle clean test` no diretório raiz do projeto.

A automatização da execução dos testes unitários pode ser definida através de uma ferramenta de integração contínua, como o Hudson. Esta ferramenta permite, a cada build ou outra periodicidade configurável, testar o código de modo a garantir que o mesmo continue funcionando como esperado mesmo com sucessivas mudanças.

### Implementando o CRUD com testes de integração

A próxima etapa consiste na implementação das ações do CRUD relacionadas à entidade **Usuario**. Feito isso a aplicação deve ser capaz de salvar, recuperar e excluir os dados dessa entidade, como o nome, data de nascimento e os respectivos dados numerológicos calculados. Para tanto, faz-se necessário incorporar as dependências do Spring e do Hibernate ao projeto, o que é feito através do arquivo `build.gradle` (vide **Listagem 8**).

As novidades, agora, estão nas linhas 1 a 3 e nas linhas 16 a 18. Note que com elas adicionamos o Spring e o banco de dados `h2database` ao projeto para que possamos implementar – e testar – as novas funcionalidades utilizando um banco de dados. Ademais, para que não haja repetição no valor da versão do Spring Boot (note que as linhas 16 e 18 declararam a mesma versão), esta é declarada numa variável, na seção `ext`, localizada no topo do arquivo.

Vamos, então, configurar a aplicação para executar os testes de integração e criar as classes Java que estarão relacionadas ao CRUD. Na **Listagem 9**, a classe **Application**, responsável por

gerenciar o Spring, é criada. Esta classe é essencial para configuração do Sprint Boot, como veremos logo mais. Já na **Listagem 10** é implementado um repositório com as ações básicas do CRUD, necessárias para que seja possível persistir, alterar, consultar e excluir os usuários da base de dados. E na **Listagem 11** temos o código da classe de testes, que valida cada um destes métodos do repositório.

**Listagem 8.** Adição das dependências do Spring e Hibernate ao arquivo `build.gradle`.

```
01 ext {  
02     springBootVersion = '1.2.6.RELEASE'  
03 }  
04  
05 apply plugin:'java'  
06 apply plugin:'idea'  
07  
08 sourceCompatibility = 1.8  
09 targetCompatibility = 1.8  
10  
11 repositories {  
12     mavenCentral()  
13 }  
14  
15 dependencies {  
16     compile "org.springframework.boot:spring-boot-starter-data-jpa:${springBootVersion}"  
17     compile 'com.h2database:h2:1.4.189'  
18     testCompile "org.springframework.boot:spring-boot-starter-test:${springBootVersion}"  
19     testCompile 'junit:junit:4.12'  
20     testCompile 'org.apache.commons:commons-lang3:3.4'  
21     testCompile 'org.hamcrest:hamcrest-all:1.3'  
22 }
```

**Listagem 9.** Código da classe **Application**.

```
01 package br.com.devmedia;  
02  
03 import org.springframework.boot.SpringApplication;  
04 import org.springframework.boot.autoconfigure.SpringBootApplication;  
05  
06 @SpringBootApplication  
07 public class Application {  
08     public static void main(String[] args) {  
09         SpringApplication.run(Application.class, args);  
10     }  
11 }
```

**Listagem 10.** Código do repositório de ações do CRUD.

```
01 package br.com.devmedia;  
02  
03 import org.springframework.data.repository.CrudRepository;  
04  
05 import java.util.List;  
06  
07 public interface UsuarioRepository extends CrudRepository<Usuario, Long> {  
08  
09     List<Usuario> findByName(String nome);  
10  
11 }
```

# Testes unitários, de integração e de aceitação na prática

Listagem 11. Código da classe de teste do repositório.

```
01 package br.com.devmedia;
02
03 import org.junit.Test;
04 import org.junit.runner.RunWith;
05 import org.springframework.beans.factory.annotation.Autowired;
06 import org.springframework.boot.test.SpringApplicationConfiguration;
07 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
08
09 import javax.transaction.Transactional;
10 import java.time.LocalDate;
11 import java.time.Month;
12 import java.util.List;
13
14 import static org.hamcrest.CoreMatchers.not;
15 import static org.hamcrest.CoreMatchers.notNullValue;
16 import static org.hamcrest.MatcherAssert.assertThat;
17 import static org.hamcrest.core.IsEqual.equalTo;
18
19 @RunWith(SpringJUnit4ClassRunner.class)
20 @SpringApplicationConfiguration(classes = Application.class)
21 @Transactional
22 public class UsuarioRepositoryIT {
23
24     @Autowired private UsuarioRepository repository;
25
26     @Test
27     public void deveCadastrarUsuario() {
28         Usuario usuario = new Usuario("Maria da Silva",
29                                         LocalDate.of(1980, Month.JANUARY, 15));
30         repository.save(usuario);
31         List<Usuario> usuariosDoBanco = repository.findByNome("Maria da Silva");
32         assertThat(usuariosDoBanco.size(), equalTo(1));
33     }
34     @Test
35     public void deveConsultarUsuarioPorNome() {
36         Usuario usuario = new Usuario("Maria da Silva",
37                                         LocalDate.of(1980, Month.JANUARY, 15));
38         repository.save(usuario);
39
40         assertThat(repository.findByNome("Maria da Silva").size(), equalTo(1));
41     }
42     @Test
43     public void deveConsultarTodosOsUsuarios() {
44         Usuario maria = new Usuario("Maria da Silva",
45                                     LocalDate.of(1980, Month.JANUARY, 15));
46         repository.save(maria);
47
48         Usuario joao = new Usuario("Joao da Silva",
49                               LocalDate.of(1980, Month.MARCH, 12));
50         repository.save(joao);
51
52         repository.findAll().forEach(c -> assertThat(c, notNullValue()));
53     }
54     @Test
55     public void deveModificarUsuario() {
56         Usuario usuario = new Usuario("Maria da Silva",
57                                     LocalDate.of(1980, Month.JANUARY, 15));
58         repository.save(usuario);
59         usuario.setNome("Maria da Silva");
60         repository.save(usuario);
61         usuario = repository.findByNome("Maria da Silva").get(0);
62         assertThat(usuario.getNumeroDoNome(), not(equalTo(0)));
63     }
64     @Test
65     public void deveExcluirUsuario() {
66         Usuario usuario = new Usuario("Maria da Silva",
67                                     LocalDate.of(1980, Month.JANUARY, 15));
68         repository.save(usuario);
69         repository.delete(usuario);
70         assertThat(repository.findByNome("Maria da Silva").size(), equalTo(0));
71     }
72 }
```

Conforme mencionado, a classe **Application** é aquela responsável por iniciar o contexto do Spring para a aplicação, assim como a própria aplicação a partir do Spring Boot. Esta solução da Spring provê diversos recursos que facilitam a integração da aplicação a bancos de dados relacionais e NoSQL, o uso de containers, o mapeamento de URLs REST para aplicações web, dentre outras vantagens. Quando a classe recebe a anotação **@SpringBootApplication**, o Spring assume algumas configurações default e as configura na aplicação. Para saber mais sobre o Spring Boot, leia o artigo publicado na Java Magazine 135.

Observando agora a **Listagem 10**, nota-se que a interface **UsuarioRepository** estende **CrudRepository** (uma interface do Spring para a implementação de CRUDs). A partir disso, o Spring Boot implementará esta interface automaticamente, com métodos comuns convencionados, como **save()**, **findAll()** e **delete()**, que não aparecem na interface. Os parâmetros **<Usuario, Long>** representam a entidade em questão e o tipo de seu atributo **Id**. A interface que herda **CrudRepository** ainda pode declarar seus próprios métodos. No nosso caso, a interface **UsuarioRepository**

declara o método **findByNome()**, para que possa buscar informações pelo nome da pessoa.

Já a classe de teste apresentada na **Listagem 11**, por meio da anotação **@RunWith**, declara que vai usar o Runner do Spring (conforme já mencionado, um Runner é responsável pelo processamento dos métodos de teste). Com isso, a classe imediatamente ganha os benefícios da injecão de dependência e, assim, passa a ser capaz de realizar testes de integração, pois conseguirá injetar recursos dependentes de bases de dados, por exemplo, e efetuar os testes de acordo. Note também que é declarada a anotação **@SpringApplicationConfiguration**. Esta define qual é a classe que provê o **ApplicationContext**, necessário para a execução do teste utilizando os recursos oferecidos pelo Spring (como a própria injecão de dependência). Outra anotação a ser declarada é **@Transactional**. Esta garante o comportamento de independência dos dados entre os testes, visto que um rollback é feito ao fim de cada execução de teste para que os dados de um teste não interfiram nos demais. Observe ainda que o nome da classe possui o sufixo “IT”, em vez do sufixo “Test”, utilizado para os testes unitários,

por causa de uma convenção tradicionalmente adotada. O objetivo dessa distinção por sufixo é separar a execução dos testes unitários dos de integração, por conta de suas características distintas.

Com as classes prontas, para executar os testes, rode-os em sua IDE ou via comando *gradle clean test*, no diretório raiz do projeto. Note que não foi escrita nenhuma classe de implementação para a interface **UsuarioRepository**. Mesmo assim, ao rodar os testes, algo interessante ocorre: os testes finalizam com sucesso. Isto ocorreu porque o Spring Boot gera esse código dinamicamente através do conceito de convenção sobre configuração. Para tanto:

- O Spring Boot identifica a interface que estendeu **CrudRepository** e então cria os métodos desta interface dinamicamente;
- O método que foi adicionado, **findByName()**, também é implementado dinamicamente, obedecendo à convenção: **findBy[nomeAtributo]**. Neste momento o Spring procura pelo atributo **nome** na classe e então implementa uma consulta utilizando este atributo como parâmetro;
- O Spring Boot também identifica a referência ao banco de dados **h2database**, ao analisar as dependências do projeto descritas no arquivo *build.gradle*, e então configura toda a parte de conexão com o banco de forma padrão, evitando assim a necessidade de escrever qualquer linha para isso.

A ideia da convenção sobre configuração prega que as atividades repetitivas sejam abstraiadas (pelo Spring Boot, neste caso) considerando suas convenções. Assim, a declaração de um método em um repositório utilizando **findBy[atributo]**, por exemplo, permite criar uma consulta com tal atributo, e o acesso ao banco se torna transparente a partir do momento em que é encontrada uma referência a alguma tecnologia relacionada, como o **h2database**.

Voltando ao nosso exemplo das **Listagens 10 e 11**, verifica-se que as ações básicas do CRUD foram implementadas e já possuem testes – de integração – que validam o correto funcionamento. Como os métodos do CRUD foram implementados pelo próprio Spring, todos os testes relacionados podem ser considerados desnecessários, correto? Contudo, valem para fins didáticos neste artigo.

Ainda falando sobre testes de integração, seguem algumas considerações:

- **Maior velocidade em relação aos testes funcionais:** uma opção tradicional para testar as ações do CRUD é de forma funcional, isto é, subir a aplicação num servidor, navegar pelas telas e então validar cada ação. Para executar esse teste, no entanto, é preciso que tanto as telas quanto a implementação do CRUD estejam prontas. Em nosso caso, já validamos todas as ações do CRUD com os testes de integração, sem que haja sequer uma tela. Os testes de integração oferecem essa vantagem. Assim, quando a tela estiver pronta, será possível dar mais atenção às funcionalidades diretamente relacionadas a ela (como validação de campos em termos de caracteres válidos e casas decimais, posicionamento de layout, etc.);
- **Qualidade na implementação:** quando o teste feito por tela é realizado de forma manual, pode acontecer de nem todos os que são necessários serem realizados. Ademais, esforços manuais estão

mais sujeitos a erros. Com os testes de integração automatizados é possível iniciar com um conjunto de casos de teste pequeno e ir aumentando progressivamente, até ter um número grande de métodos que avaliam, de forma abrangente, a classe de produção. A automatização permite que os testes sejam executados sempre da mesma maneira, de forma a garantir o mesmo nível de qualidade. Os testes unitários e de aceitação também possuem esta característica. Nos testes de integração, contudo, o foco está na relação com os recursos externos, dos quais a aplicação depende.

A execução automatizada dos testes de integração pode ser viabilizada da mesma forma que a supracitada para os testes unitários, fazendo uso de uma ferramenta de integração contínua, como o Hudson. Contudo, existe uma complexidade adicional: o servidor de integração contínua precisa ter acesso aos recursos necessários para a execução dos testes de integração (como, por exemplo, acesso à rede onde fica o servidor de banco de dados utilizado pelos testes).

### Implementando os testes de aceitação

Criaremos agora os testes de aceitação. Para isso, a primeira coisa a fazer é configurar a aplicação para suportar o Concordion, framework de testes de aceitação que utilizaremos em nosso exemplo. Deste modo, ajuste o arquivo *build.gradle* de acordo com o conteúdo apresentado na **Listagem 12**.

**Listagem 12.** Configuração do build.gradle para suportar o Concordion.

```
01 ext {  
02     springBootVersion = '1.2.6.RELEASE'  
03 }  
04  
05 apply plugin:'java'  
06 apply plugin:'idea'  
07  
08 test {  
09     systemProperty'concordion.output.dir', "$buildDir/reports/concordion"  
10 }  
11  
12 sourceCompatibility = 1.8  
13 targetCompatibility = 1.8  
14  
15 repositories {  
16     mavenCentral()  
17 }  
18  
19 dependencies {  
20     compile "org.springframework.boot:spring-boot-starter-data-jpa:${springBootVersion}"  
21     compile 'com.h2database:h2:1.4.189'  
22     testCompile "org.springframework.boot:spring-boot-starter-test:${springBootVersion}"  
23     testCompile 'org.concordion:concordion:1.5.1'  
24     testCompile 'junit:junit:4.12'  
25     testCompile 'org.apache.commons:commons-lang3:3.4'  
26     testCompile 'org.hamcrest:hamcrest-all:1.3'  
27 }
```

# Testes unitários, de integração e de aceitação na prática

Observe que entre as linhas 08 e 10 é definida uma **System-Property** para indicar ao Concordion em qual diretório ele deve gravar o relatório gerado após a execução dos testes. Por default, o Concordion usa o diretório temporário do usuário, mas queremos que ele use o diretório de relatórios, *reports*, que é automaticamente criado no diretório *build* da aplicação após o processo de build. Na linha 23, adicionamos a sua dependência.

Com a configuração pronta, nosso próximo passo é especificar as histórias que queremos automatizar, conforme verificamos a seguir:

## User Story: Usuário obtém a numerologia de seu nome completo

DADO um usuário já cadastrado com o nome completo “Maria da Silva”

E data de nascimento 15/01/2015

QUANDO o usuário obtém o número respectivo ao seu nome completo

ENTAO o número obtido é 2

## User Story: Usuário obtém a numerologia de sua data de nascimento

DADO um usuário já cadastrado com o nome completo “Maria da Silva”

E data de nascimento 15/01/2015

QUANDO o usuário obtém o número respectivo à sua data de nascimento

ENTAO o número obtido é 6

A implementação dos testes relacionados a estas histórias deve recuperar os números de nome completo e de data de nascimento do banco de dados e compará-los com os valores indicados nas histórias. Com as user stories em mãos, vamos agora mapeá-las para os métodos de teste na aplicação. Antes disso, iniciemos executando o comando a seguir na raiz do projeto para criar o diretório de especificação das histórias:

```
cd src/test/java/resources && mkdir -p br/com/devmedia
```

Logo após, ainda neste diretório, crie dois arquivos para a especificação das histórias e os codifique com o conteúdo apresentado nas **Listagens 13 e 14**.

Na **Listagem 13** é declarado o arquivo de história da numerologia do nome completo. Este possui o texto da história instrumentada com tags HTML para permitir a automatização. Para tanto, logo na linha 1 é adicionado o namespace do Concordion, para que seja possível, dentre outras coisas, usufruir de funções, como permitir à IDEA que realize o *autocomplete* das tags do Concordion. A linha 6 define uma **div** com a **class="example"**, classe que renderiza uma linha azul clara ao redor da história, como será visto posteriormente, na análise do relatório. Na linha 8, a tag **concordion:set** define o valor da variável **nomeCompleto** como sendo “Maria da Silva”, e na linha 10 temos o mesmo comportamento, mas com a variável **dataNascimento**, assumindo a data “15/01/2015”.

**Listagem 13.** Conteúdo do arquivo *UsuarioObtemNumerologiaNomeCompleto.html*.

```
01 <html xmlns:concordion="http://www.concordion.org/2007/concordion">
02 <body>
03
04 <h1>User Story: Usuário obtém numerologia do nome completo</h1>
05
06 <div class="example">
07   DADO um usuário já cadastrado com o nome completo
08     <span concordion:set="#nomeCompleto">Maria da Silva</span><br />
09   E data de nascimento
10    <span concordion:set="#dataNascimento">15/01/2015</span><br />
11    <span concordion:execute="criarUsuario(#nomeCompleto,
#dataNascimento)" />
12    <span concordion:execute="cadastrarNumerologiaDoNomeDoUsuario()" />
13
14  QUANDO o usuário obtém o número respectivo ao seu nome completo
15    <span concordion:execute="#resultado = obterNumerologiaNome
Completo()" /><br />
16
17  ENTAO o número obtido é <span concordion:assertEquals="#resultado">2
</span><br />
18
19  <span concordion:execute="excluirUsuario()" />
20 </div>
21
22 </body>
23 </html>
```

**Listagem 14.** Conteúdo do arquivo *UsuarioObtemNumerologiaDataNascimento.html*

```
01 <html xmlns:concordion="http://www.concordion.org/2007/concordion">
02 <body>
03
04 <h1>User Story: Usuário obtém numerologia da data de nascimento</h1>
05
06 <div class="example">
07   DADO um usuário já cadastrado com o nome completo
08     <span concordion:set="#nomeCompleto">Maria da Silva</span><br />
09   E data de nascimento
10    <span concordion:set="#dataNascimento">15/01/2015</span><br />
11    <span concordion:execute="criarUsuario(#nomeCompleto,
#dataNascimento)" />
12    <span concordion:execute="cadastrarNumerologiaDaDataDeNascimento
DoUsuario()" />
13
14  QUANDO o usuário obtém o número respectivo à sua data de nascimento
15    <span concordion:execute="#resultado = obterNumerologiaData
NascimentoDoBanco()" /><br />
16
17  ENTAO o número obtido é <span concordion:assertEquals="#resultado">6
</span><br />
18
19  <span concordion:execute="excluirUsuario()" />
20 </div>
21
22 </body>
23 </html>
```

Já na linha 11, é chamado um método para criar uma instância do usuário para o teste, utilizando os valores das duas variáveis supracitadas. Na linha 12, por sua vez, a instância do usuário é salva no banco de dados. Na linha 15, o número respectivo ao nome completo do usuário é recuperado do banco, e na linha 17, a condição é testada, para verificar se o usuário previamente salvo e recuperado do banco de dados tem um número de nome

completo igual a 2. Por fim, na linha 19 o usuário criado para o teste é excluído do banco de dados, pois foi criado e salvo apenas para fins deste teste, e não queremos que permaneça no banco de dados.

Note que a User Story está escrita com uma série de tags que chamam métodos para realizar lógicas específicas. Por exemplo, na linha 19 temos uma chamada ao método `excluirUsuario()` para executar a lógica de excluir um usuário. Esta lógica é descrita no atributo `concordion:execute` da tag `span`. Mas, como este mapeamento é feito para o código Java correspondente? Para que isto seja feito o Concordion procura por uma classe Java com o mesmo nome do arquivo HTML, mas com a extensão `Fixture`, estabelecendo assim a ligação entre o arquivo HTML e a classe Java. Portanto, se o nome do arquivo com a descrição da User Story é `UsuarioObtemNumerologiaNomeCompleto.html`, o Concordion precisa que exista uma classe Java de nome `UsuarioObtemNumerologiaNomeCompletoFixture` no pacote de testes da aplicação, porque é desta forma que a ligação é formada.

Vamos, então, criar esta classe. Ou melhor, como a solução da **Listagem 13** é semelhante à da **Listagem 14**, vamos criar as duas classes `Fixture` – e também uma classe pai abstrata, auxiliar, para evitar a duplicação de código. Esse código pode ser verificado nas **Listagens 15 a 17**. Crie essas classes dentro de `src/test/java;br/com/devmedia`.

Note que as classes `UsuarioObtemNumerologiaDataNascimentoFixture` e `UsuarioObtemNumerologiaNomeCompletoFixture` estendem `FixtureBase`, classe abstrata que agrega o uso dos recursos do Spring e dos conceitos do Concordion aos testes, por meio das anotações nas linhas 10 e 11. Observe ainda que `FixtureBase` faz uso de um `Runner` diferente do utilizado nas classes de testes de integração (conforme declarado na linha 10 da **Listagem 15**), de nome: `ConcordionSpringJUnit4ClassRunner`. Como este `Runner` não é provido pelo Spring, ele precisa ser criado. O código da classe `ConcordionSpringJUnit4ClassRunner` está presente no projeto completo, disponível para download.

Analizando agora a **Listagem 16**, observe que o método `cadastrarNumerologiaDaDataDeNascimentoDoUsuario()` efetua o cálculo numerológico e salva o usuário no banco de dados. Ainda nessa listagem, no método `obterNumerologiaDataNascimentoDoBanco()`, o usuário é recuperado do banco de dados e o número previamente calculado é retornado, para comparação. Estes métodos isolados estão vinculados à especificação da história (vide **Listagem 13**).

A implementação é análoga na **Listagem 17**, mas ao invés do nome, é a data de nascimento que é considerada. Seguindo as boas práticas de teste, os *fixtures* das **Listagens 16 e 17** podem ser executados independentemente, de forma que a execução de um (e seu efeito no sistema, como a alteração de dados no banco) não influencia na execução do outro.

Para verificar os testes de aceitação, execute o comando `gradle clean test` na pasta raiz do projeto. Com isso, o Concordion irá rodá-los e gerar os arquivos HTML de relatório, conforme apresentado nas **Figuras 1 e 2**.

#### **Listagem 15.** Código da classe `FixtureBase`.

```
01 package br.com.devmedia;
02
03 import org.junit.runner.RunWith;
04 import org.springframework.beans.factory.annotation.Autowired;
05 import org.springframework.boot.test.SpringApplicationConfiguration;
06
07 import java.time.LocalDate;
08 import java.time.format.DateTimeFormatter;
09
10 @RunWith(ConcordionSpringJUnit4ClassRunner.class)
11 @SpringApplicationConfiguration(classes = Application.class)
12 public abstract class FixtureBase {
13
14     @Autowired
15     protected UsuarioRepository repository;
16     protected Usuario usuario;
17
18     public void criarUsuario(String nome, String dataNascimento) {
19         this.usuario = new Usuario(nome, converter(dataNascimento));
20     }
21
22     protected Usuario obterUsuario(String nome, LocalDate dataNascimento) {
23         Usuario usuario = repository.findByNome(nome).get(0);
24
25         if (usuario.getDataNascimento().equals(dataNascimento)) {
26             return usuario;
27         } else {
28             throw new IllegalStateException();
29         }
30     }
31
32     protected LocalDate converter(String dataNascimento) {
33         DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd/MM/yyyy");
34         return LocalDate.parse(dataNascimento, formatter);
35     }
36
37     public void excluirUsuario() {
38         repository.delete(usuario);
39     }
40 }
```

#### **Listagem 16.** Código da classe `UsuarioObtemNumerologiaDataNascimentoFixture`.

```
01 package br.com.devmedia;
02
03 public class UsuarioObtemNumerologiaDataNascimentoFixture extends
04     FixtureBase {
05
06     public void cadastrarNumerologiaDaDataDeNascimentoDoUsuario () {
07         usuario.calcularNumerologiaDaDataNascimento();
08         repository.save(usuario);
09     }
10
11     public int obterNumerologiaDataNascimentoDoBanco() {
12         this.usuario = obterUsuario(usuario.getNome(), usuario.getDataNascimento());
13         return usuario.getNumeroDaDataNascimento();
14     }
15 }
```

Tais relatórios podem ser encontrados em: /  
`build/reports/concordion/br/com/devmedia/UsuarioObtemNumerologiaDataNascimento.html` e /`build/reports/concordion/br/com/devmedia/UsuarioObtemNumerologiaNomeCompleto.html`.

Os valores em verde representam validações com sucesso (`assertEquals=true`). No caso de falhas, o relatório seria exibido de forma semelhante à **Figura 3**. Isto é, o texto fica em vermelho, e o valor esperado (no caso, 9) fica riscado.

# Testes unitários, de integração e de aceitação na prática

## Verificando os relatórios de testes do Gradle

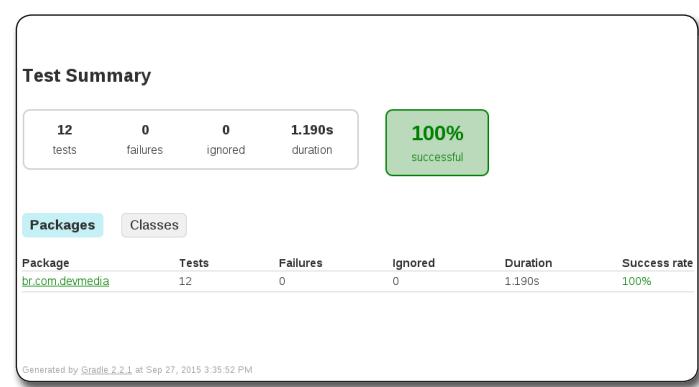
O Gradle também possui um conjunto de relatórios de testes que são gerados em HTML. De acordo com o funcionamento deste gerenciador, relatórios são gerados quando os testes implementados são executados pelo comando `gradle clean test`. As Figuras 4, 5 e 6

apresentam alguns desses relatórios, que podem ser acessados a partir do arquivo `/build/reports/tests/index.html`.

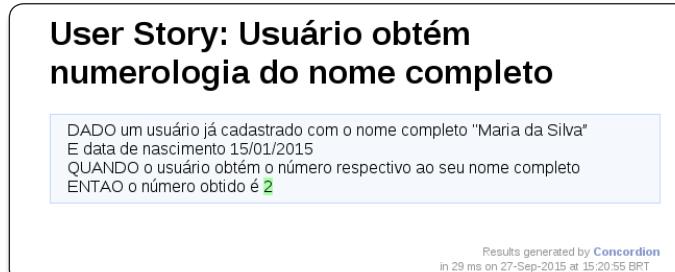
Assim como indicado para os testes unitários e de integração, para automatizar a execução dos testes de aceitação podemos adotar o Hudson.

**Listagem 17.** Código da classe `UsuarioObtemNumerologiaNomeCompletoFixture`

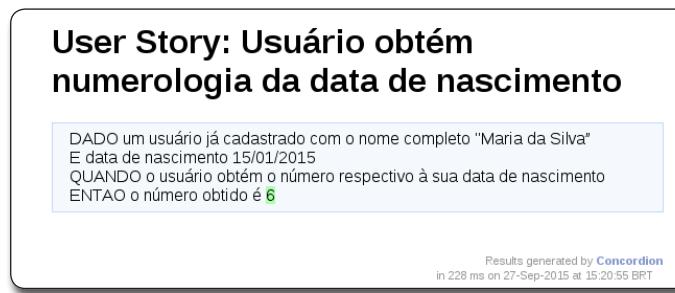
```
01 package br.com.devmedia;
02
03 public class UsuarioObtemNumerologiaNomeCompletoFixture extends
04     FixtureBase {
05
06     public void cadastrarNumerologiaDoNomeDoUsuario() {
07         usuario.calcularNumerologiaDoNome();
08         repository.save(usuario);
09     }
10
11     public int obterNumerologiaNomeCompleto() {
12         this.usuario = obterUsuario(usuario.getNome(), usuario.getDataNascimento());
13         return usuario.getNumeroDoNome();
14     }
```



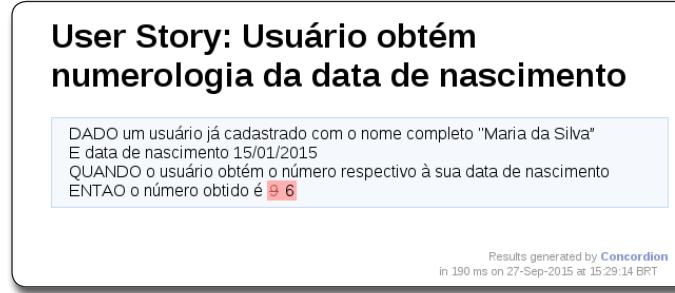
**Figura 4.** Relatório de pacotes testados pelo Gradle



**Figura 1.** Relatório de sucesso da execução da fixture `UsuarioObtemNumerologiaNomeCompleto`



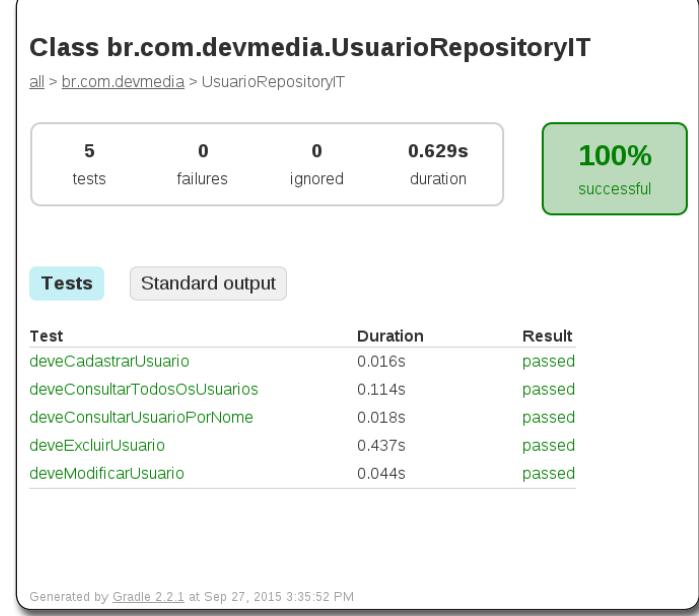
**Figura 2.** Relatório de sucesso da execução da fixture `UsuarioObtemNumerologiaDataNascimento`



**Figura 3.** Relatório de falha da execução da fixture `UsuarioObtemNumerologiaDataNascimento`



**Figura 5.** Relatório de classes testadas pelo Gradle



**Figura 6.** Relatório de execução dos métodos da classe de teste de integração `UsuarioRepositoryIT`

Encerrada essa análise prática, o leitor terá importantes fundamentos para a etapa de testes dos seus projetos. Ademais, as contribuições aqui apresentadas servem também para desmistificar algumas ideias ainda disseminadas, como a de que os testes são um trabalho restrito à equipe de testes e a de que todos os testes automatizados podem ser planejados e construídos sem levar em consideração as regras dos diferentes tipos de teste e suas particularidades.

Como extensão a este trabalho, recomenda-se a análise dos testes funcionais de caixa preta e como automatizá-los (analisar a opção Selenium), a execução dos testes via integração contínua (com ferramentas como o Hudson) e a análise da cobertura dos testes, com o auxílio de plataformas como o SonarQube.

## Autor



### Daniel Medeiros de Assis

[daniel.medeiros.assis@gmail.com](mailto:daniel.medeiros.assis@gmail.com)

<https://br.linkedin.com/in/dmassis>



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

## Links:

### Livro “Test Driven Development: By Example”.

<http://www.amazon.com/Test-Driven-Development-By-Example/dp/0321146530>

### Livro “xUnit Test Patterns”.

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

### Endereço para download do Gradle.

<https://gradle.org/gradle-download/>

### Código-fonte da classe ConcordionSpringJUnit4ClassRunner.java (utilizada no projeto).

<https://gist.github.com/eeichinger/2719776>

## Você gostou deste artigo?

Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)



Ajude-nos a manter a qualidade da revista!

# Injeção de dependências em aplicações Android

Aprenda neste artigo como reduzir o acoplamento de suas aplicações Android e facilite a compreensão e manutenção do código

**A**injeção de dependências (DI) ou Inversão de Controle (IoC) são termos utilizados como sinônimos, no entanto, o termo IoC é aplicado dentro e fora do contexto de injeção de dependências. Esta terminologia refere-se à inversão da responsabilidade que uma classe tem de buscar e controlar suas próprias dependências. A inversão faz com que essa responsabilidade seja transferida a um *framework* capaz de buscar as dependências automaticamente. Ou seja, o controle sobre as dependências é invertido – deixa de ser implementado pelo desenvolvedor e passa a ser gerenciado por um componente especializado. Essa característica de inversão pode ser empregada a qualquer *framework* que retire a responsabilidade do programador de codificar um determinado código para realizar uma tarefa específica.

Um framework de injeção de dependências, ou contêiner *IoC*, é capaz de construir desde um simples objeto até uma longa cadeia de objetos dependentes entre si.

A injeção de dependências implementada pelos *frameworks* mais populares soa como mágica pela maneira transparente de instanciar e atribuir objetos aos locais que são requisitados. Fazendo uma analogia com o mundo real, é possível imaginar que ao invés caminhar todos os dias para comprar pão e tomar café, seria bem mais fácil se o pão aparecesse em cima da mesa, não seria?

Em um sistema de pequeno porte o esforço para instanciar manualmente cada uma das classes envolvidas e atribuí-las ao local correto pode até ser trivial, mas à medida que o sistema cresce, outros requisitos surgem, a complexidade aumenta e a tarefa de manter todos os objetos corretamente construídos e relacionados torna-se mais onerosa para o programador, além de ser bastante suscetível a falhas por qualquer esquecimento.

## Fique por dentro

Neste artigo será apresentado um padrão de projeto que é bastante utilizado no mundo da programação orientada a objetos e que pode agregar bastante produtividade ao desenvolvimento de aplicações móveis com Android. A injeção de Dependências é um design pattern útil para a criação e atribuição de objetos que tenham interdependência entre si. Essas dependências representam o relacionamento dos objetos em um sistema, relacionamento este que pode se tornar bastante complexo para ser realizado manualmente pelo programador.

A injeção de dependências é uma solução que proporciona transparência na construção dos objetos envolvidos, possibilitando desacoplar e variar a implementação sem necessariamente codificar para isso. No entanto, alguns problemas podem surgir se os padrões citados anteriormente não forem seguidos como, por exemplo, a dependência circular, que ocorre quando dois objetos dependem um do outro para inicializarem. Perceba que o simples fato de um objeto depender do outro não é exatamente uma limitação que a injeção de dependências seja incapaz de atender, contudo, na maioria das vezes este cenário não é um bom desenho OO e se ambos os objetos necessitarem da execução de um código de inicialização, o contêiner IoC não poderá determinar quem chamar primeiro. Esta situação hipotética fere o princípio do baixo acoplamento, pois um objeto impacta diretamente no comportamento do outro, ocasionando um fluxo de comunicação anormal entre eles.

Apesar da injeção de dependências ter se tornado uma alternativa popular para fazer ligações indiretas entre objetos, o padrão *Service Locator* costumava ser uma solução bastante utilizada para o mesmo problema. Nesta abordagem, a atribuição de descobrir e criar objetos é delegada a uma classe que centraliza a instânciação ou procura (*lookup*) dos objetos. Assim, toda classe chama



diretamente o *Service Locator* para obter uma instância de outro objeto que ela dependa. Desta forma, este centralizador termina acoplado com todas as classes do sistema, o que não é um desenho orientado a objetos apropriado. Devido a este efeito colateral, o *Service Locator* vem caindo em desuso e abrindo lugar para a injeção de dependências. Entretanto, no âmbito da API do Android, o *Service Locator* ainda é um padrão muito utilizado. Isso se deve à antiguidade da API, que é evoluída sempre um passo antes em relação à evolução do Java, linguagem na qual as aplicações Android são desenvolvidas.

Após esta contextualização sobre o padrão de projeto Injeção de Dependências, será apresentado um exemplo de utilização do RoboGuice em um projeto demonstrativo. No entanto, antes de iniciar o projeto, iremos entrar em uma breve explicação sobre o ambiente de desenvolvimento, com o qual será desenvolvido o exemplo.

## Android Studio

Antes de iniciar um projeto de desenvolvimento de software é necessário escolher a IDE que será utilizada. A IDE (*Integrated Development Environment*) é a ferramenta que auxiliará o programador a desenvolver o software com maior produtividade e integração.

Durante muito tempo a IDE oficial para desenvolvimento de aplicações em Android foi o ADT (*Android Development Tools*), que era uma solução baseada na plataforma Eclipse. Estas ferramentas poderiam ser baixadas em um pacote único, ou adicionadas ao Eclipse SDK, por meio de plug-ins.

Após algum tempo realizando testes com versões *betas*, em dezembro de 2014 o Google lançou a versão final do Android Studio. Esta nova plataforma de desenvolvimento Android é baseada na IDE IntelliJ e possui diversas melhorias em relação ao Eclipse ADT. À primeira vista, nota-se um desempenho aparentemente melhor que o Eclipse, além de contemplar uma ferramenta de gerenciamento de bibliotecas e dependências, que é o Gradle.

O gerenciamento de bibliotecas com o Eclipse ADT deixava muito a desejar. Esta função não estava disponível na instalação original. Deste modo, fazia-se necessário instalar a extensão m2e para se obter um gerenciamento de bibliotecas com o Maven, mas ainda assim, esta solução era muito inconstante. Com base nisso, os exemplos deste artigo utilizaram o Android Studio 1.0.1, que pode ser obtido a partir do endereço indicado na seção **Links**.

## RoboGuice

O RoboGuice (pronuncia-se “Robo juice”) é um framework que faz uso do Google Guice, uma API do Google, e que traz a simplicidade e facilidade de uso da Injeção de Dependências ao Android. Seu funcionamento é parecido com outros frameworks de IoC, como o Spring ou o CDI, no entanto, é otimizado e adaptado para aplicações Android.

Assim como foi explicado na introdução, a Injeção de Dependências auxilia o programador a reduzir a quantidade de linhas

de código, além de facilitar o entendimento, e no RoboGuice isso não poderia ser diferente.

Antes de entender o funcionamento do RoboGuice, no entanto, é necessário saber o que são recursos no Android, pois o principal recurso deste framework é facilitar a ligação do código com os elementos da tela.

Recursos são artefatos além do código utilizado nas aplicações Android. Eles podem ser classificados em imagens, ícones, layouts, elementos de tela, XML, strings, etc. Estes recursos são armazenados em subpastas previamente categorizadas na pasta *res* de um projeto Android, local onde todo e qualquer recurso deve residir. Durante o desenvolvimento, o Android procura todos os recursos na referida pasta, pré-compila e gera automaticamente o arquivo *R.java* com todos os recursos encontrados. Este procedimento é realizado instantaneamente sempre que um recurso é alterado. Portanto, a classe **R** jamais deve ser modificada manualmente.

Os recursos são tratados de maneira especial no Android porque são compilados em um tipo binário que o torna eficiente em relação ao tamanho e o acesso pelo aplicativo, com exceção dos recursos que já são binários ou que não se deseja essa otimização, os quais devem ser incluídos na pasta de recursos *raw* (recursos não compilados).

Sabendo que a classe **R** contém todos os identificadores dos recursos disponíveis na aplicação, é suficiente chamar o método *findViewById()* passando o identificador do recurso na tela para que este retorne o referido componente, conforme é possível visualizar no código da **Listagem 1**.

**Listagem 1.** Código Android sem injeção de dependências.

```
01. class AndroidWay extends Activity {  
02.     TextView name;  
03.     ImageView thumbnail;  
04.     LocationManager loc;  
05.     Drawable icon;  
06.     String myName;  
07.  
08.     public void onCreate(Bundle savedInstanceState) {  
09.         super.onCreate(savedInstanceState);  
10.         setContentView(R.layout.main);  
11.         name = (TextView) findViewById(R.id.name);  
12.         thumbnail = (ImageView) findViewById(R.id.thumbnail);  
13.         loc = (LocationManager) getSystemService(Activity.LOCATION_SERVICE);  
14.         icon = getResources().getDrawable(R.drawable.icon);  
15.         myName = getString(R.string.app_name);  
16.         name.setText("Hello, " + myName);  
17.     }  
18. }
```

A API do Android como um todo foi toda desenvolvida tendo como base o padrão *Service Locator*. Este fato é observado pela presença de métodos que recebem identificadores e retornam serviços, a exemplo do método *getSystemService()* (vide linha 13). Além disso, o Android tem relação muito íntima com o JDK, pois a compilação para a plataforma móvel é realizada a partir de arquivos *.class* gerados pelo mesmo.

# Injeção de dependências em aplicações Android

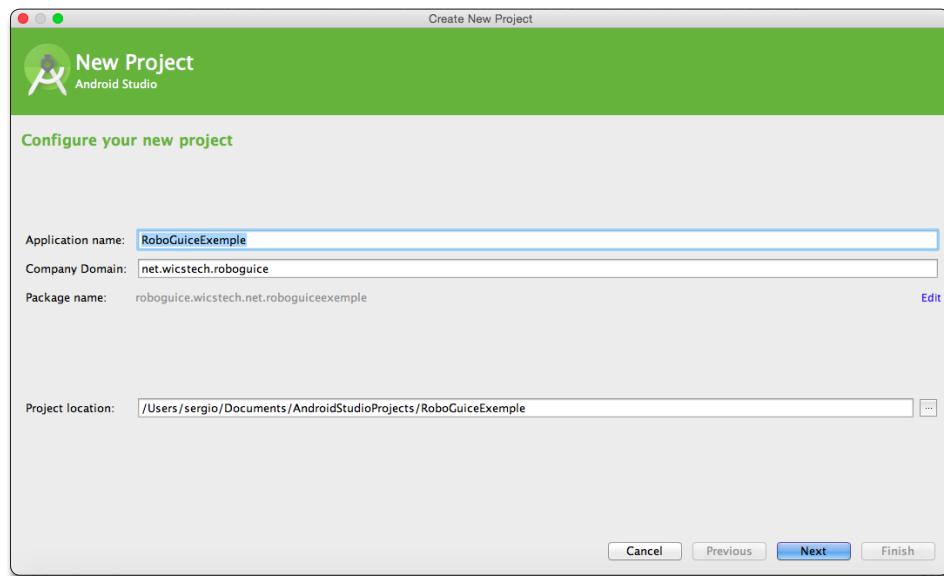


Figura 1. Criação de um projeto no Android Studio

Listagem 2. Android Activity com uso do RoboGuice.

```
01. @ContentView(R.layout.main)
02. class RoboWay extends RoboActivity {
03.     @InjectView(R.id.name)
04.     TextView name;
05.     @InjectView(R.id.thumbnail)
06.     ImageView thumbnail;
07.     @InjectResource(R.drawable.icon)
08.     Drawable icon;
09.     @InjectResource(R.string.app_name)
10.    String myName;
11.    @Inject
12.    LocationManager loc;
13.
14.    public void onCreate(Bundle savedInstanceState) {
15.        super.onCreate(savedInstanceState);
16.        name.setText("Hello, " + myName);
17.    }
18. }
```

Deste modo, as versões do Android possuem dependência com versões específicas do JDK e sua API foi escrita com padrões de codificação aderentes a versões mais antigas do Java. Por este motivo, faz muito uso de *Service Locator* e pouco uso de anotações.

Sabendo que o *Service Locator* é um padrão que funciona de maneira oposta à injeção de dependências, conforme foi apresentado na introdução, o uso do RoboGuice em aplicações Android transforma as inúmeras chamadas aos métodos que retornam os serviços de sistemas e elementos de tela em declarações de classe anotadas. Desta forma, o mesmo código apresentado na **Listagem 1** poderia ser reescrito com anotações no código apresentado na **Listagem 2**.

Em resumo, o método `onCreate()`, apresentado inicialmente na **Listagem 1**, é reduzido de oito para duas linhas, tornando-se ainda mais legível. No tópico a seguir, será montado um projeto básico

demonstrando o uso e a configuração do RoboGuice.

## Montando um projeto com RoboGuice e Android Studio

Como mencionado anteriormente, utilizaremos o Android Studio em nosso projeto exemplo. Sendo assim, após baixar e realizar a instalação, abra a ferramenta e crie um novo projeto, iniciando-o com os mesmos parâmetros apresentados na **Figura 1**. Após informar esses dados, clique em *Next*. Na tela seguinte, selecione a opção *Phone and Tablet*, o Android 4.4 (KitKat, API Level 19) e clique em *Next*. Agora, selecione o *template Blank Activity*, clique em *Next* e depois em *Finish*.

Com o projeto criado, o próximo passo é adicionar a biblioteca do RoboGuice ao projeto. Para isso, utilizaremos o Gradle

como ferramenta de gerenciamento de bibliotecas. Ao criar qualquer projeto, o Android Studio já cria um arquivo com as configurações básicas do Gradle. Este arquivo está presente em cada módulo do aplicativo. Em nosso caso, o projeto criado foi definido automaticamente com uma estrutura multi-módulo, onde existe o projeto principal, com configurações mais abrangentes, e um projeto filho (*Module: app*), onde os fontes do aplicativo propriamente dito serão armazenados. Essa estrutura propicia uma divisão mais apropriada do projeto, sendo possível, inclusive, aproveitar fontes de um projeto J2SE em outro módulo e adicioná-lo ao projeto.

O Gradle possui muitas outras funções, além de gerenciar dependências, contudo, para o objetivo deste artigo será necessário apenas incluir a biblioteca do RoboGuice, para que o Gradle faça o *download* e adicione a dependência ao projeto em questão. Para isso, procure pelo arquivo *build.gradle* do módulo *app* e abra-o. Ao observar sua estrutura, pode-se notar que muitas configurações do projeto estão constantes neste arquivo, a exemplo do identificador do aplicativo e o nível mínimo de API suportada (*API Level*). Ao final deste arquivo, existe a seção *dependencies*, que deve conter todas as bibliotecas necessárias ao projeto. Para adicionar a biblioteca do RoboGuice, acrescente a linha a seguir neste trecho e depois salve o arquivo. Feito isso, aguarde a configuração do projeto e então a biblioteca estará pronta para uso.

```
1 dependencies {
2     ...
3     compile 'org.roboguice:roboguice:3.0.1'
4 }
```

A diretiva `compile` indica que a referida biblioteca será compilada e empacotada com o APK do aplicativo. Esta notação de biblioteca contém as mesmas informações que são utilizadas no Maven;

a propósito, o Gradle pode fazer uso de repositórios Maven. As informações que definem a biblioteca são compostas pelo *groupId* (primeira parte antes do caractere ":"), que representa a organização que criou a biblioteca, o *artifactId*, o qual se refere ao nome da biblioteca e, por último, a versão da dependência.

No momento da criação do projeto, foi solicitada a criação de uma *Activity* padrão, a qual deve ter sido criada sob o nome de **MainActivity**. Acrescentaremos dois campos textos, dois rótulos e um botão nesta atividade. Assim, procure pelo arquivo *MainActivity.java* e altere-o para que fique parecido com a **Listagem 3**. Em seguida, procure pelo arquivo *activity\_main.xml* e substitua seu conteúdo pelo da **Listagem 4**.

Observando o código desta atividade, a principal alteração realizada e necessária para que as anotações sejam efetivamente lidas, é fazer com que a *activity* herde a classe **roboguice.activity.RoboActionBarActivity**. Assim como esta, existem diversas classes bases para implementação de *services*, *Broadcast Receivers*, *Fragments*, *Async Tasks*, entre outras, as quais devem ser herdadas para que as anotações declaradas sejam lidas.

As anotações possuem nomes intuitivos e que, normalmente, são os mesmos nomes utilizados nos métodos, a exemplo do método **setContentView()** da *Activity*, o qual é substituído pela anotação **@roboguice.inject.ContentView** (vide linha 15, **Listagem 3**). Já os campos que serão manipulados pelo código são declarados como atributos de instância da atividade e anotados com **@roboguice.inject.InjectView**. Observe que a injeção é realizada nas linhas 21 e 24, mesmo que os campos estejam marcados como privados.

Continuando com o código da **Listagem 3**, o método **mostreValores()** (linha 49) é acionado pelo botão constante na **Figura 2**. O bloco estático da linha 17 foi necessário porque no momento da escrita deste artigo a biblioteca do RoboGuice apresentava um erro de execução e essa foi a solução de contorno recomendada no *bug tracking* do framework.

Como foi visto neste exemplo, algumas anotações são próprias do RoboGuice e moldadas especificamente para o Android. No entanto, outras anotações são utilizadas pelo RoboGuice e várias outras bibliotecas de injeção de dependências, como é o caso das anotações do pacote **javax.inject** (JSR 250), que padronizam o uso de injeção de dependências para todas as soluções desenvolvidas em Java.

### Anotações da JSR 250

O RoboGuice suporta as anotações da JSR 250 – *Common Annotations for the Java Platform*, que trata-se de uma especificação para a Java EE e Java SE e agora aproveitada para frameworks Android. Esta JSR define anotações de uso geral com o intuito de manter a semântica de conceitos utilizados em ambas as plataformas. No entanto, em razão das especificidades da plataforma Android, nem todas as anotações deste pacote foram aproveitadas.

No âmbito desta JSR, um bean significa o objeto que foi construído pelo contêiner de injeção de dependências e distribuído aos seus pontos de injeção.

Além de buscar e atribuir recursos da própria plataforma Android, o RoboGuice também é capaz de instanciar e atribuir

qualquer objeto Java, por meio da anotação **@javax.inject.Inject**, que instrui ao framework IoC a instanciar e buscar a declaração anotada. Este recurso é muito útil quando se tem uma cadeia de objetos muito complexa, contudo, em alguns casos é necessário controlar a quantidade de vezes que um determinado objeto é instaciado, e para este fim, é necessário estabelecer o escopo de criação deste objeto.

**Listagem 3.** Código da classe *MainActivity*.

```
01. package roboguice.wicstech.net.roboguiceexample;
02.
03. import android.view.Menu;
04. import android.view.MenuItem;
05. import android.view.View;
06. import android.widget.EditText;
07. import android.widget.Toast;
08.
09. import roboguice.RoboGuice;
10. import roboguice.activity.RoboActionBarActivity;
11. import roboguice.inject.ContentView;
12. import roboguice.inject.InjectView;
13.
14.
15. @ContentView(R.layout.activity_main)
16. public class MainActivity extends RoboActionBarActivity {
17.     static {
18.         RoboGuice.setUseAnnotationDatabases(false);
19.     }
20.     @InjectView(R.id.editTextNome)
21.     private EditText editTextNome;
22.
23.     @InjectView(R.id.editTextSobreNome)
24.     private EditText editTextSobreNome;
25.
26.
27.     @Override
28.     public boolean onCreateOptionsMenu(Menu menu) {
29.         // Inflate the menu; this adds items to the action bar if it is present.
30.         getMenuInflater().inflate(R.menu.menu_main, menu);
31.         return true;
32.     }
33.
34.     @Override
35.     public boolean onOptionsItemSelected(MenuItem item) {
36.         // Handle action bar item clicks here. The action bar will
37.         // automatically handle clicks on the Home/Up button, so long
38.         // as you specify a parent activity in AndroidManifest.xml.
39.         int id = item.getItemId();
40.
41.         //noinspection SimplifiableIfStatement
42.         if (id == R.id.action_settings) {
43.             return true;
44.         }
45.
46.         return super.onOptionsItemSelected(item);
47.     }
48.
49.     public void mostreValores(View target) {
50.         Toast.makeText(getApplicationContext(),
51.             String.format("%s %s", editTextNome.getText(),
52.                         editTextSobreNome.getText()), Toast.LENGTH_LONG).show();
53.     }
54. }
```

# Injeção de dependências em aplicações Android

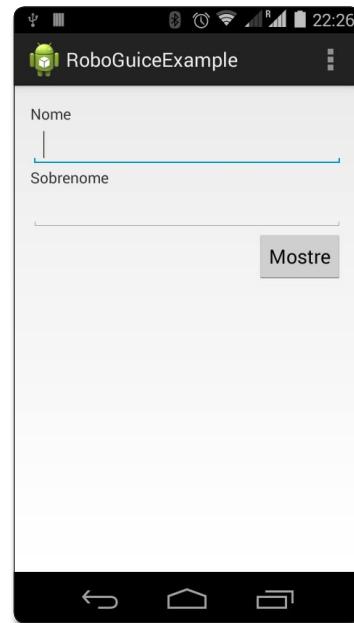
**Listagem 4.** Conteúdo do arquivo activity\_main.xml.

```
01. <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
02.     xmlns:tools="http://schemas.android.com/tools" android:layout_width=
03.         "match_parent"
04.     android:layout_height="wrap_content" android:paddingLeft=
05.         "@dimen/activity_horizontal_margin"
06.     android:paddingRight="@dimen/activity_horizontal_margin"
07.     android:paddingTop="@dimen/activity_vertical_margin"
08.     android:paddingBottom="@dimen/activity_vertical_margin"
09.     tools:context=".MainActivity"
10.    android:orientation="vertical"
11.    android:baselineAligned="false">
12.
13.    <TextView
14.        android:layout_width="fill_parent"
15.        android:layout_height="wrap_content"
16.        android:text="@string/label_nome"
17.        android:id="@+id/textViewNome"/>
18.
19.    <EditText
20.        android:layout_width="fill_parent"
21.        android:layout_height="0dp"
22.        android:inputType="textPersonName"
23.        android:ems="10"
24.        android:id="@+id/editTextNome"
25.        android:layout_weight="1"/>
26.
27.    <TextView
28.        android:layout_width="fill_parent"
29.        android:layout_height="wrap_content"
30.        android:text="@string/label_sobrenome"
31.        android:id="@+id/textViewSobrenome"/>
32.
33.    <EditText
34.        android:layout_width="match_parent"
35.        android:layout_height="wrap_content"
36.        android:inputType="textPersonName"
37.        android:ems="10"
38.        android:id="@+id/editTextSobrenome"/>
39.
40.    <Button
41.        android:layout_width="wrap_content"
42.        android:layout_height="wrap_content"
43.        android:text="Mostre"
44.        android:id="@+id/buttonMostre"
45.        android:layout_gravity="right"
46.        android:onClick="mostreValores"/>
47. </LinearLayout>
```

## Escopos

Um escopo determina como um *bean* será visto por outros e, ao mesmo tempo, controla o número de instâncias deste objeto por contexto, bem como estabelece o seu ciclo de vida. Quando o escopo de um *bean* não é especificado, ele assume o escopo padrão, que relaciona o ciclo de vida deste objeto ao ciclo de vida do objeto que o utiliza. Ademais, embora em outras arquiteturas e frameworks mais maduros existam diversos tipos de escopo, no RoboGuice existe apenas os escopos padrão, *Singleton*, *ContextSingleton* e *FragmentSingleton*.

Para determinar o escopo de um objeto, deve-se anotar a classe a ser gerenciada pelo RoboGuice com a anotação de escopo desejada, as quais serão apresentadas a seguir. A injeção é sempre realizada a partir de classes que herdam a API do RoboGuice, ou seja, para



**Figura 2.** Tela do exemplo

cada classe da API do Android, existe uma subclasse do RoboGuice responsável por realizar a injeção de dependências, a exemplo da classe **RoboActionBarActivity**, apresentada na **Listagem 3**.

Ao anotar a classe com **@javax.inject.Singleton**, o RoboGuice irá instanciar a referida classe apenas uma vez e distribuí-la aos pontos de injeção sempre que requisitado. Importante ressaltar que *beans* anotados com este escopo somente serão destruídos quando a aplicação finalizar, ou seja, objetos deste tipo permanecerão na memória mesmo quando sua *activity* não estiver visível. O uso indevido desta anotação pode causar estouros de memória na aplicação (*memory leaks*).

Em alguns casos, o uso do escopo **@ContextSingleton**, por sua vez, pode reduzir o tempo de vida de um *bean*. Enquanto os *beans* *singletons* são atrelados à aplicação, o escopo **ContextSingleton** é atrelado a um contexto específico (o qual pode se referir a uma *activity* ou serviço da aplicação). Deste modo, esses objetos são destruídos juntos com o contexto.

Por fim, o escopo **@FragmentSingleton**, introduzido na versão 3.x do RoboGuice, garante a unicidade de instâncias por *Fragments* de tela. Os escopos **ContextSingleton** e **FragmentSingleton** são próprios do RoboGuice, enquanto o escopo **Singleton** é definido pela JSR 250.

## Classes de Suporte da API

Assim como foi visto na **Listagem 3**, para que ocorra a injeção de dependências nas declarações da classe, é necessário que a *activity* herde a **RoboActionBarActivity**, que é uma classe de suporte para a **ActionBarActivity** do Android. Da mesma forma, existem classes de suporte do RoboGuice para diversos recursos do Android, como as listadas a seguir:

- **roboguice.activity.RoboListActivity**;
- **roboguice.activity.RoboFragmentActivity**;

- `roboguice.service.RoboService;`
- `roboguice.service.RoboIntentService;`
- `roboguice.util.RoboAsyncTask;`
- `roboguice.receiver.RoboBroadcastReceiver.`

Esta é apenas uma lista exemplificativa. Existem mais classes de suporte no pacote `roboguice.activity` para diversas especializações da classe `Activity`. Não obstante, caso o programador deseje realizar a injeção a partir de qualquer outra classe, tendo uma referência do contexto (`android.content.Context`), é necessário apenas realizar a chamada a seguir, onde o `this` representa o objeto no qual se deseja realizar a injeção:

```
RoboGuice.getInjector(context).injectMembers(this);
```

Outro recurso bastante útil, oriundo da injeção de dependências com RoboGuice, é o uso de eventos, que será abordado a seguir.

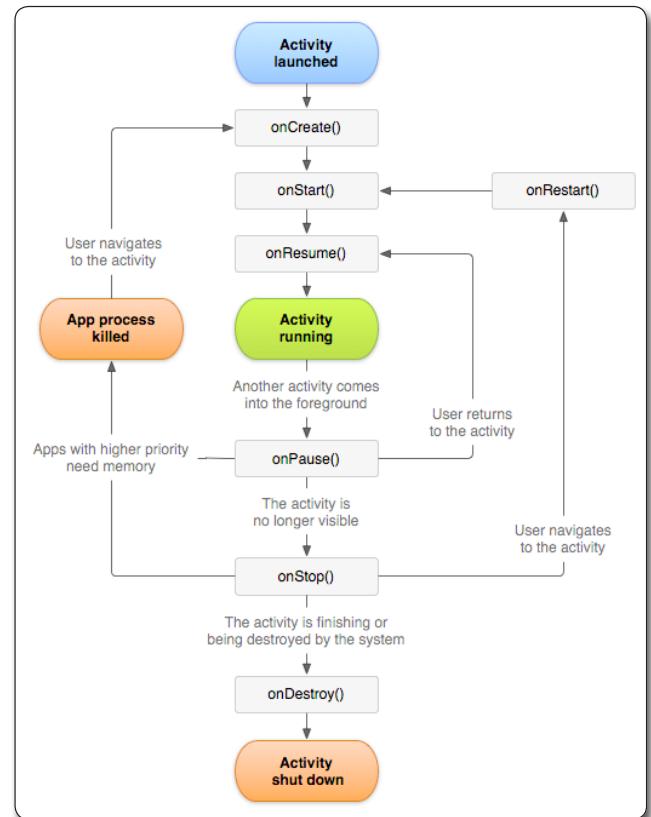
## Eventos

A API do Android naturalmente utiliza bastante a orientação a eventos. Estes ajudam a capturar momentos do ciclo de vida de uma atividade ou serviço para moldar o comportamento do aplicativo. Neste sentido, é possível demonstrar esta característica por meio do ciclo de vida de uma atividade, apresentado na **Figura 3**, o qual é composto por métodos de *call-back* que são chamados pelo próprio Android.

Para fazer uso dos eventos descritos na figura, a funcionalidade deve ser implementada em métodos sobrescritos na própria atividade. Por exemplo, o evento `onPause()` é acionado sempre que a atividade é parcial ou totalmente sobreposta por outra. Assim, quando se desejar que determinado código seja executado quando este evento ocorrer, deve-se sobrescrever o método `onPause()` da classe `android.app.Activity` na atividade do aplicativo. É importante ressaltar que para todos os métodos do ciclo de vida no Android, deve-se sempre chamar a implementação original com `super` (exemplo: `super.onPause()`). Essa recomendação normalmente é válida para qualquer framework codificado em Java que use algum ciclo de vida.

Em uma implementação que não faz uso de eventos do RoboGuice, o código a ser executado é escrito dentro do método sobrescrito do evento respectivo e quando este código representa mais de uma funcionalidade, é recomendável que seja separado em métodos ou modularizado da maneira mais intuitiva possível, de modo a facilitar futuras manutenções. No entanto, quando a implementação atinge certo grau de complexidade, a leitura do código atrelado a eventos torna-se igualmente complicada, pois não é possível distinguir corretamente a qual evento pertence um determinado código sem que seja necessário acompanhar o fluxo da execução ou procurar na hierarquia de chamadas.

No intuito de melhorar a legibilidade do código, o RoboGuice permite que o código atrelado a eventos de sistema seja modularizado em classes anotadas, sendo possível também criar eventos customizados.



**Figura 3.** Ciclo de vida de um Atividade no Android – Fonte: Google

Modularizar é distribuir o código em artefatos independentes de maneira a deixar mais clara a distribuição das responsabilidades no código. No âmbito do RoboGuice, a modularização pode ocorrer em nível de classes, ou seja, cada funcionalidade deve ser codificada em uma classe coesa, e também ocorrer em métodos anotados dentro da própria atividade, sem que seja necessário sobrescrever o evento desejado, mas ainda assim deixando o código intuitivo no que diz respeito a qual evento ele se relaciona.

Para criar métodos atrelados a eventos de sistema, basta criar um método que tenha um parâmetro com um dos tipos a seguir, a ser escolhido de acordo com o evento de sistema que se pretende observar:

- Eventos da atividade:
  - `roboguice.activity.event.OnActivityResultEvent`;
  - `roboguice.activity.event.OnContentChangedEvent`;
  - `roboguice.activity.event.OnNewIntentEvent`;
  - `roboguice.activity.event.OnPauseEvent`;
  - `roboguice.activity.event.OnRestartEvent`;
  - `roboguice.activity.event.OnResumeEvent`;
  - `roboguice.activity.event.OnSaveInstanceStateEvent`;
  - `roboguice.activity.event.OnStopEvent`.
- Eventos associados com o contexto:
  - `roboguice.context.event.OnConfigurationChangedEvent`;
  - `roboguice.context.event.OnCreateEvent`;
  - `roboguice.context.event.OnDestroyEvent`;
  - `roboguice.context.event.OnStartEvent`.

# Injeção de dependências em aplicações Android

Essas classes de evento armazenam informações sobre o evento ocorrido.

Por fim, o parâmetro que contém as informações do evento (a exemplo dos dispostos nas linhas 3, 7 e 11 da **Listagem 5**) deve ser anotado com `@roboguice.event.Observe`, o que indica ao RoboGuice o registro destes eventos assim que a classe que herda uma das classes de suporte (`RoboActionBarActivity`, `RoboService`, `RoboAsyncTask`, etc.) é inicializada. A referida anotação só pode ser utilizada em parâmetros de métodos – não funciona se colocada em parâmetros de construtores.

Em relação à modularização dos eventos por classes, quando se quiser distribuir a funcionalidade em classes coesas além da própria atividade, é possível definir uma classe contendo métodos “ouvintes” separada da *activity*. O formato destes métodos é o mesmo que o apresentado na **Listagem 5** e explicado nos parágrafos anteriores, contudo, a classe deve ser injetada, por meio de uma declaração com `@javax.inject.Inject`, em uma subclasse de uma das classes de suporte, tal como é mostrado na **Listagem 6**, onde a classe `MeusOuvintes` (linha 7) é injetada na classe `MinhaAtividade2` (linhas 2 e 3).

## Listagem 5. Observando eventos de sistema na própria Atividade.

```
01. public class MinhaAtividade extends RoboActivity {  
02.  
03.     public void doSomethingOnResume(@Observes OnResumeEvent onResume) {  
04.         Ln.d("método onResume chamado");  
05.     }  
06.  
07.     public void doSomethingElseOnCreate( @Observes OnCreateEvent onCreate ) {  
08.         Ln.d("onCreate foi chamado savedInstanceState é %s",  
09.             onCreate.getInstanceState());  
10.  
11.     public void xxx( @Observes OnCreateEvent onCreate ) {  
12.         Ln.d("On create chamado!");  
13.     }  
14. }
```

## Listagem 6. Listeners em classe separada da Activity.

```
01. public class MinhaAtividade2 extends RoboActivity {  
02.     @Inject  
03.     private MeusOuvintes myListener;  
04.  
05. }  
06.  
07. class MeusOuvintes {  
08.     public void doSomethingOnResume(@Observes OnResumeEvent onResume) {  
09.         Ln.d("Called doSomethingOnResume in onResume");  
10.     }  
11. }
```

O rastreio dos eventos é realizado no momento da criação da *activity* (ou outro recurso, como o contexto) durante a chamada do método `onCreate()`. Além disso, deve-se ter em mente que estes eventos são propagados em um contexto apenas. Eles não ultrapassam contextos diferentes, ou seja, um evento registrado em uma atividade não será ouvido por outra.

Quando se desejar disparar eventos que sejam ouvidos em múltiplos contextos, use a instância `singleton` da classe `EventManager`, a qual deve ser obtida por meio da declaração a seguir:

```
@Named(DefaultRoboModule.GLOBAL_EVENT_MANAGER_NAME)  
@Inject  
private EventManager eventManager;
```

O registro e o cancelamento de manipuladores de eventos globais devem ser realizados por meio dos métodos `registerObserver()` e `unregisterObserver()` da respectiva classe, os quais recebem o tipo do evento (classe) e a instância a ser associada a este evento. Os eventos são propagados para todos os “ouvintes” registrados para um determinado evento e podem ser disparados pelo código da própria aplicação por meio do método `fire()` do `EventManager`.

O objetivo da injeção de dependências é reduzir o tamanho e a complexidade do código, tornando-o mais intuitivo e limpo por meio do uso de anotações. A anotação é um recurso poderoso que existe em diversas linguagens modernas, como o próprio Java e o C#. Normalmente, possuem configurações que restringem o local onde podem ser alocadas (sobre um método, classe, parâmetro de método, etc.) e esta característica permite que se defina o escopo para a mesma. À primeira vista, podem funcionar como um complemento da documentação do código, mas também podem ser lidas e alterar o comportamento do código, servindo como um meta-dado.

Apesar da API do Android não utilizar anotações, em razão da idade da referida API, conforme dito anteriormente, o uso de anotações já é um padrão bastante comum entre os frameworks empregados em aplicações corporativas desenvolvidas na mesma linguagem e tende a ser um padrão para Android, possivelmente até com suporte nativo da plataforma, sem a necessidade de inclusão de bibliotecas terceiras.

Neste contexto, o RoboGuice é uma biblioteca leve que contempla recursos de injeção de dependências além do trivial, a exemplo do uso de eventos, que possibilita o desenvolvimento de um código desacoplado e modularizado. Por fim, no que concerne a performance, o uso da injeção não onera o desempenho da aplicação, o que pode ser visto como algo bastante positivo, por impactar positivamente na produtividade e manutenção do código.

## Autor



### Sergio Eduardo Dantas de Oliveira

É pós-graduado em Dispositivos Móveis e graduado em Sistemas de Informação pela Unieuro. Possui certificações SCJP e SCWCD. Trabalha há 11 anos com TI, sendo nove anos com desenvolvimento de Software e cinco anos como Arquiteto de Sistemas e atualmente é servidor público na CONAB (Companhia Nacional de Abastecimento).



## Você gostou deste artigo?

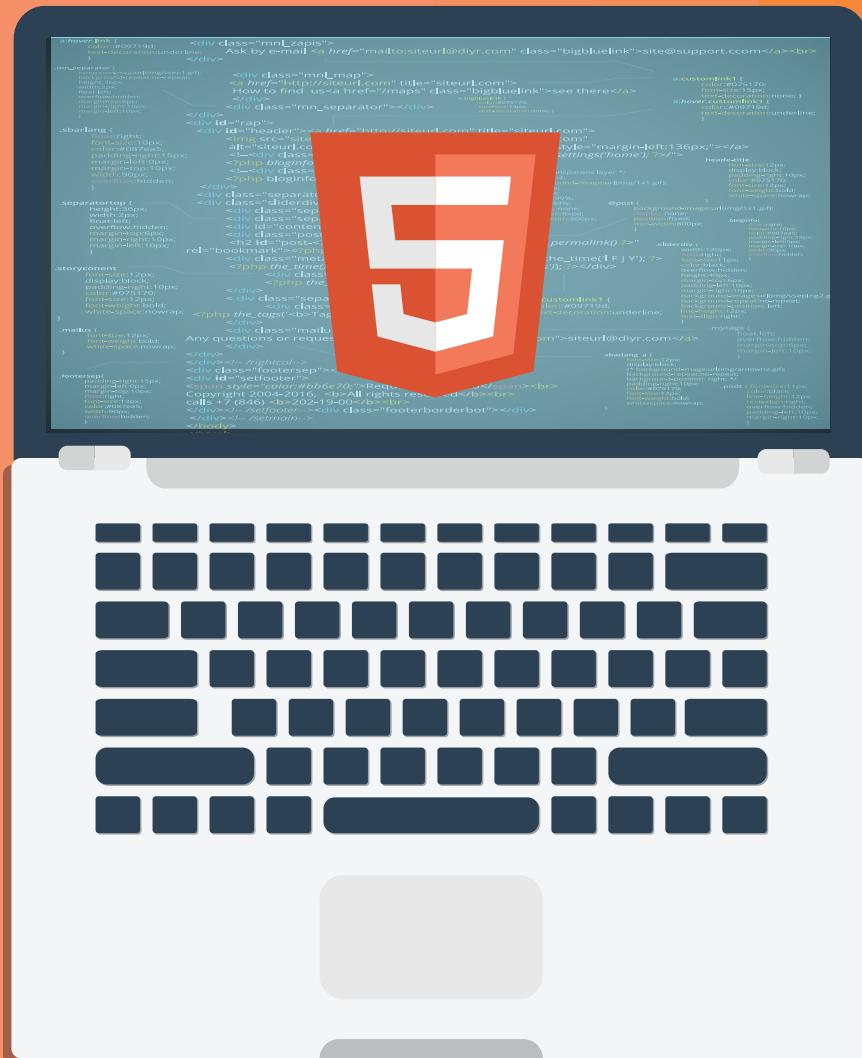
Dê seu voto em [www.devmedia.com.br/javamagazine/feedback](http://www.devmedia.com.br/javamagazine/feedback)

Ajude-nos a manter a qualidade da revista!



# Guia HTML 5

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



DEVMEDIA

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

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

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

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

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



## Estrutura

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

## Supporte diferenciado

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

## Serviços

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

## 1-to-1

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



# Porta 80

WEB HOSTING

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

[porta80.com.br](http://porta80.com.br) | [comercial@porta80.com.br](mailto:comercial@porta80.com.br) | [twitter.com/porta80](http://twitter.com/porta80)

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