

ThoughtWorks®

PRÁTICAS E TENDÊNCIAS EM TESTE

*Um conjunto de experiências
entregando software de alta qualidade*

2015

compartilhe este ebook



O

AUTORES



DANIEL
AMORIM



MARCOS
BRIZENO



NEIL PHILIP
CRAVEN



RAFAEL
GARCIA



RAQUEL
LIEDKE



MAX
LINCOLN



LUCAS
MEDINA



FABIO
PEREIRA



NICHOLAS
PUFAL



TAISE
SILVA



LEONARDO
STEFFEN



RODRIGO
TOLLEDO



JURACI
VIEIRA

CONTEÚDO

<i>6 regras de ouro para ser um QA arretado*</i>	6
<i>Testador ágil 3.0</i>	9
<i>Prevenção de defeitos usando técnicas ágeis</i>	13
<i>Mockar ou não mockar, eis a questão</i>	16
<i>3 noções básicas essenciais para a criação de uma suíte de automação para aplicativos web</i>	20
<i>Escreva testes melhores em 5 passos</i>	26
<i>Três faláncias sobre BDD</i>	29
<i>Apenas execute o build novamente - ele deve ficar verde</i>	33
<i>Entrega Contínua com build quebrado e consciência limpa</i>	35
<i>Melhorando a qualidade de projetos através do poder das nuvens</i>	38
<i>Melhoria da qualidade com a nuvem - parte II</i>	43
<i>Protractor: testando aplicações angularJS com uma solução integradora</i>	47
<i>Protractor na prática em 3 passos</i>	49
<i>Gatling: uma ferramenta de testes de performance</i>	54
<i>Alternativas ao Testflight para Android</i>	57



OK, O CÓDIGO DE TESTE TESTA O CÓDIGO FUNCIONAL. MAS QUEM TESTA O CÓDIGO DE TESTE?

Anos atrás, seguidamente me faziam tal pergunta. E eu me enrolava todo tentando responder. Eram os anos iniciais de métodos ágeis. TDD e integração contínua ainda eram novidade para a maioria das pessoas nas conferências de TI. E eu ali, tentando falar de um pedacinho da parte visível do iceberg.

Nessa época, a indústria de software já se sentia evoluída. UML, base de dados relacional, Java (escreva uma vez e execute em qualquer lugar!). Ah, e os cargos. Todos bem definidos e de acordo com as fases de desenvolvimento de software. Analista do negócio, desenvolvedor e testador. E, para garantir o sucesso, arquitetos e gerentes.

Mas o nosso mundo mudou, e rápido! Está tudo nas nuvens. Os métodos são ágeis, e estamos no caminho da Internet das Coisas. Ferrou! Desculpem o meu francês, mas ferrou. Como vamos construir software de qualidade? “Constrói aí que depois alguém testa” não cola mais. Tampouco aquela minha visão simplista de que teste automatizado com integração contínua iria resolver todos os seus problemas.

Era apenas a ponta do iceberg e, quando você ia mergulhar para entender a parte imersa, veio o tal do Global Warming e derreteu tudo. O bloco de gelo não está mais separado. Derreteu e se juntou ao mar que o rodeava.

A metáfora era falha! A separação entre departamentos também. Assim como a clareza das fases de desenvolvimento de software e seus respectivos guardiões. Qualidade e, portanto, testes e verificação não são mais atividades separadas da criação do software.

A frase estava errada. Não é que o código de teste testa o código funcional. Mas sim que o time preza por qualidade, e está colaborando para gerar valor mais rápido e com maior frequência.

E eis que surgem pessoas interessadas nessa nova filosofia. E elas não se identificam claramente por um cargo ou papel: testadores com skills de programação, analistas de qualidade, analistas do negócio, DevOps ou desenvolvedores com perfil analítico e de qualidade.

E é isto que meus prezados colegas compartilham neste e-Book, por meio de contos, histórias e dicas sobre software e qualidade: compartilham como fazemos para alcançar valor mais rápido e com maior frequência.

Boa leitura!

PAULO CAROLI

Consultor principal da Thoughtworks e cofundador da AgileBrazil.



6 REGRAS DE OURO PARA SER UM QA ARRETADO*

Leonardo Steffen



Você sabe o que te faz querer ir para o trabalho, mesmo quando o trabalho não está tão divertido assim? Sabe o que te mantém forte quando todos os testes estão falhando e as pessoas ao seu redor ficam o tempo todo perguntando o que está acontecendo de errado?

Eu não sabia. Ou pelo menos não tinha consciência.

Mas a vida como consultor nos ensina muitas coisas sobre

pessoas, suas interações com os membros da equipe e a forma como decisões são tomadas e aplicadas.

Prestei atenção no meu próprio modo de agir e no que me fazia feliz ao longo do dia, mesmo quando o dia não era dos melhores. Isso me ajudou a criar uma lista de princípios que eu utilizo como guia, e sempre que algo não está bem eu me pergunto o quanto estou me esforçando para aplicar esses princípios ao meu dia. Experimente e me conte se funcionou para você também.

Princípio # 0 | Tenha a mente aberta

Preso a um aplicativo legado? Até mesmo a pior tecnologia já teve seu momento de glória.

É difícil concordar com tudo o que se passa com times que trabalham em aplicações legadas. Mas as pessoas trabalham fazendo o seu melhor, e é isso que devemos ter em mente. Técnicas, ferramentas, plataformas, linguagens e metodologias são escolhidas com as melhores intenções. Antes de abandonar uma tecnologia, entenda o contexto das escolhas anteriores e veja se é possível melhorar a forma ela é utilizada, dado o contexto atual.

Princípio # 1 | Pense no futuro

Prepare-se para o futuro. Testes são a mais valiosa documentação de um sistema.

Aprenda a escrever testes. Aprenda a entendê-los também. Compartilhe o seu conhecimento e ajude outros membros da equipe. Reveja os testes e elimine aqueles que não são mais úteis. Modifique testes sempre que necessário - eles não são imutáveis e existem para te ajudar, não para te dar mais trabalho.

Princípio # 2 | Pense grande

Projete seus testes pensando em mais do que os critérios de aceitação.

Ao projetar um teste, entenda as ações dos usuários do sistema e construa testes robustos ao invés de simples scripts focados em estórias de usuários. E, sempre que você se deparar com um teste, pergunte a si mesmo: "o que esse teste vai me dizer daqui a cinco

meses?". "Esse teste está testando um nível apropriado? Deveria ser unitário, de serviço ou via interface gráfica?".

Princípio # 3 | Pense com sabedoria

Não automatize tudo só porque você pode e porque a ferramenta é legal, ou porque o seu gerente pediu.

É difícil não colocar as mãos naquele novo framework do qual todos estão falando. A questão é: precisa mesmo automatizar tanto assim? Escreva testes demais e você vai acabar com muitas horas de trabalho extra apenas para mantê-los passando. A dica é (clichê, mas sempre boa): pense antes de automatizar. Decisões técnicas cabem a você.

Princípio # 4 | Mantenha a calma

Quebrar build não é uma coisa ruim.

Desde que você saiba por que quebrou. Se você não sabe, preste atenção nos princípios # 1, # 2 e # 3. Pergunte-se: "nossa pipeline de build é realmente uma pipeline?" ou "é possível saber quais alterações de código estão envolvidas nessa falha?".

Princípio # 5 | Seja gentil

Trate o seu código de automação de teste com todo o respeito dado ao seu código de aplicação.

Não existe essa história de código de teste e código de desenvolvimento. Testes e aplicação podem ter focos diferentes, mas têm o mesmo objetivo final - entregar valor para o cliente. E enquanto o seu sistema existir, esse código também existirá.

Esses são meus princípios para poder contribuir consistentemente no trabalho como um QA. Você já pensou nos seus? Pense e me diga o que você descobriu! É incrível como um pedaço de papel e alguns minutos de introspecção podem ajudar a transformar um dia ruim em uma grande oportunidade de consultoria.

*Não pude encontrar palavra melhor do que essa para o termo *Awesome*, então aí vai uma dica de *podcast tecnologicamente arretado*.

*Outra dica de leitura - o artigo *Agile Tester 3.0*. Esse artigo descreve uma visão com a qual simpatizo, e diz respeito aos perfis de atuação de um QA. Após ler os 6 princípios, reflita sobre qual dimensão cada um deles se enquadra. Vale a pena!

TESTADOR ÁGIL 3.0

Daniel Amorim



Testadores ágeis são muitas vezes conhecidos como analistas de qualidade (QAs), engenheiros de software em testes, engenheiros de testes, líderes de qualidade, entre outras variações. Eu tenho trabalhado como um Agile QA por um tempo - em português podemos traduzir como analista de qualidade ágil. Eu gostaria de compartilhar meu ponto de vista sobre como um QA trabalha em um time ágil. Nesse artigo eu vou usar a terminologia QA para representar o testador ágil.

A maioria das pessoas, mesmo em times ágeis, trata QAs como sub-papéis ou papéis separados do time. Eu acredito que isso

esteja totalmente fora de moda. A diferença entre um QA e um desenvolvedor (também chamado de Dev) é apenas o jeito de pensar (ou mindset para os que estão acostumados com o termo).

Mas o que distingue QAs entre eles mesmos? Perfis de QA podem ser classificados em 3 categorias: Negócio, Técnico e DevOps. Eu chamo essas três de "3 dimensões de perfis de QA". QAs podem ter tanto um desses perfis como também podem combinar os três perfis de acordo com o seu nível de conhecimento em cada uma das dimensões.

Vamos mergulhar mais a fundo nessas dimensões e entender melhor cada uma delas:

Dimensão de negócio

Os QAs nessa dimensão são realmente dirigidos a negócio. Eles têm habilidades que ajudam seus times a entender o contexto de negócio dado pelo cliente. Eles têm boas habilidades de comunicação que auxiliam o time a focar no problema de negócio durante o projeto todo.

Extrair testes de aceitação do cliente é uma das especialidades e BDD é uma das técnicas usadas para quebrar a barreira entre contexto de negócio vindo do cliente e contexto técnico vindo dos engenheiros do time.

Eles trabalham em par com os desenvolvedores para alinhar o que precisa ser feito com o cliente antes de começar a jogar as estórias. Durante esse período eles guiam o par para escrever testes de aceitação que certifiquem que a estória estará testada antes de moverem adiante.



Esses QAs geralmente leem livros como:

Specification by Example: How Successful Teams Deliver the Right Software

Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing

The Cucumber Book: Behaviour Driven-Development for Testers and Developers

Dimensão técnica

Eu me identifico com essa dimensão porque os QAs aqui são muito técnicos e têm boas habilidades de programação. No mundo ideal, não deveria existir nenhuma diferença entre um QA e um desenvolvedor. Em um time ágil, todo mundo é um engenheiro e deveria ser tratado como tal.

Os QAs técnicos trabalham em par com desenvolvedores para construir a aplicação sem gap técnico. Eles codam juntos. Eles também ajudam os desenvolvedores a desenvolver usando TDD, promovendo boas práticas como código limpo e padrões de desenvolvimento, garantindo um código de alta qualidade.

Eles têm muito conhecimento em automação de testes e ajudam o time a escolher o melhor framework de testes para o projeto. Eles também são responsáveis por garantir que o time tenha uma boa estratégia de testes em mãos.

Os QAs na dimensão técnica podem também trabalhar com testes de performance e segurança, dependendo do quanto avançado é o conhecimento deles em testes não funcionais.

Para testes de performance, eles trabalham com o cliente para descobrir os SLAs (Service Level Agreement). Dadas essas informações, eles criam os testes de performance para mensurar e rastrear as melhorias feitas na aplicação relacionadas aos SLAs.

Esses QAs também são envolvidos em segurança. Eles entendem o contexto de negócio com o cliente e analisam possíveis vulnerabilidades. Com isso, eles criam testes de segurança para garantir que essas possíveis vulnerabilidades estão sendo cobertas por algum mecanismo de segurança.



QAs nesta dimensão geralmente leem livros como:

[*Test Driven Development: By Example*](#)

[*Clean Code: A Handbook of Agile Software Craftsmanship*](#)

[*Selenium Testing Tools Cookbook*](#)

[*The Art of Application Performance Testing: Help For Programmers and Quality Assurance*](#)

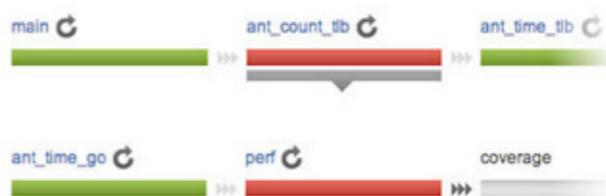
Dimensão de DevOps

Como DevOps está relacionado a testes? Existem várias coisas que QAs podem fazer para ajudar o seu time através do seu conhecimento em DevOps.

Eles introduzem a prática de Entrega Contínua e ajudam o time a criar um pipeline de integração contínua para receber um feedback mais rápido após cada commit. Isso ajuda o time a fazer deploy de novas funcionalidades para produção mais vezes. Em alguns casos, cada commit vai diretamente para produção após executar todo o pipeline com sucesso. Esse pipeline irá também executar o build e empacotar a aplicação, ferramentas de qualidade de código, testes unitários, testes de componente e testes funcionais.

Os QAs DevOps configuram scripts para o time rodar mais facilmente os testes em suas máquinas locais. Em alguns casos, máquinas virtuais são necessárias e nelas os testes são configurados para executar em paralelo.

Eles usam task runners para que o time execute tarefas repetitivas sem muito esforço, tais como auto watches para executar testes automatizados depois de cada vez que alguém salva o código fonte. Isso diminui o tempo de feedback durante o desenvolvimento de novas funcionalidades.



Esse QA geralmente lê livros como:

Continuous Integration: Improving Software Quality and Reducing Risk

Continuous Delivery: Reliable Software Releases through Build, Test and Deployment Automation

As melhores referências em português são:

DevOps na prática: entrega de software confiável e automatizada

Entrega Contínua: Como Entregar Software de Forma Rápida e Confiável!

O que é comum para todos os QAs?

QAs em todas essas três dimensões mantêm o time focado em entregar o valor certo para o cliente durante cada ciclo de desenvolvimento. Ao mesmo tempo, eles devem estar preocupados com a qualidade do produto que está sendo entregue.

Além de compartilharem a responsabilidade dos testes com o time, também transmitem para o time todo o conhecimento que eles têm sobre testes. Através dessa abordagem, cada membro do time pensará sobre testes independentemente de seu papel. QAs vestem muitos chapéus, mas seu foco principal deveria ser em ajudar o time a entregar valor de negócios frequentemente e com qualidade.

Livro que todos os QAs ágeis deveriam ler (que é a atual bíblia do Agile Testing)

Agile Testing: A Practical guide for Testers and Agile Teams

E você? Onde você está nessas dimensões? Em qual delas você tem interesse em melhorar?

...e vamos começar!

PREVENÇÃO DE DEFEITOS USANDO TÉCNICAS ÁGEIS

Lucas Medina & Raquel Liedke



Inúmeros podem ser os deslizes cometidos ao descrever uma estória. Eles podem levar a defeitos de implementação caso não sejam validados antes da estória ser desenvolvida. Detalhes que aparentemente estão claros na cabeça do analista de negócio, ou mesmo do cliente, acabam não sendo completamente detalhados na descrição da estória.

Prevenir defeitos no software o mais cedo possível é um objetivo sempre almejado por qualquer projeto de desenvolvimento de software e muitas técnicas são utilizadas para achar esses

defeitos precocemente. Uma delas nos tem dado um retorno bem interessante na manutenção da qualidade no projeto: o kick-off e o desk-check de estórias.

Como estórias em análise ainda estão abertas a sugestões, uma proposta seria uma reunião para discutir as próximas estórias, na qual o analista de negócio (ou um integrante com contexto), apresenta as próximas estórias, comenta a origem, o valor de negócios e o porquê são necessárias. O time discute detalhes técnicos, faz questionamentos sobre os requisitos e avalia se

o tamanho da estória está adequado. Esse feedback pode ser usado para dividir a estória onde for apropriado, ou adicionar mais detalhes e exemplos para clarificar áreas que causaram confusão.



O kick-off, que na tradução literal seria ‘ponta-pé inicial’, trata-se de uma lista de checagem com vários itens a serem verificados antes de iniciar o desenvolvimento da estória. Essa lista inclui validações como: a estória está completa e realmente pronta para o desenvolvimento? Que considerações técnicas precisam ser observadas durante o desenvolvimento? Já sabemos sobre o design visual? E quanto a abordagens para controlar erros e mensagens de ajuda?

Outro problema muito comum são premissas que estão na cabeça do desenvolvedor e não compartilhadas entre desenvolvedor e analista (falha de comunicação). Sendo assim, o bate papo inicial

traz à tona os conceitos e objetivos sem deixar passar detalhes importantes, além de enriquecer o contexto com o ponto de vista de pessoas em diferentes papéis.

Em um cenário ideal, a participação de analistas de negócio, do analista de qualidade e dos desenvolvedores é fundamental no kick-off para que todos fiquem a par da funcionalidade a ser desenvolvida e para que o debate seja mais produtivo e proveitoso. Confira uma sugestão de kick-off abaixo:

Checklist para o Kick-off

Envolvidos: Analista de Negócios, Analista de Teste, Desenvolvedores

- A análise da estória está completa?
- Houve revisão de QA na análise?
- A estória está completa com detalhes e toda a informação relevante?
- O valor da estória foi bem compreendido?
- Há dependência em futuras estórias?
- Há algum débito técnico relacionado?
- Há design visual para a estória?
- A estória possui mensagens de erros detalhadas?
- Há mensagens de ajuda e outros labels definidos na estória, já revisados?
- O tamanho da estória é apropriado?

Pelas mesmas razões justificadas no kick-off, o analista de negócio, o analista de qualidade e desenvolvedores devem estar presentes para ampliar o debate sobre dúvidas e problemas durante o desenvolvimento. Seria importante considerar que para estórias grandes, podem ser feitas checagens durante o desenvolvimento, afim de garantir que se está no caminho certo. Deve-se ter atenção para que estes feedbacks não ultrapassem o escopo da estória e,

se eles fizerem sentido, provavelmente a análise da estória não foi acurada.

O desk-check (teste de mesa), trata-se de uma lista de checagem a ser utilizada na validação da estória após o desenvolvimento. Na listagem abaixo, vemos validações como: foram implementados os testes unitários e de aceitação? Já foi feita uma revisão do código com outro par? Funciona nos principais navegadores? Tais perguntas são importantes para a prevenção de dores de cabeça no futuro. Exemplo: a funcionalidade está linda no Google Chrome, mas sequer aparece no Internet Explorer porque as bibliotecas ou a versão do html não funcionam lá.

Checklist para o Desk-Check

Envolvidos: Analista de Negócios, Desenvolvedores e pessoas interessadas

- Há cobertura de teste suficiente?
- Outro par foi convidado para revisar o código?
- A estória foi testada manualmente?
- A verificação da estória pode ter impactado em outra coisa?
- Todos os critérios de aceitação foram cobertos?
- Será que a estória precisa de algum tipo de feedback ou correção?
- Foi feita uma completa jornada de usuário através da estória?
- A interação com a UI está consistente com o resto da aplicação?
- Funciona nos principais navegadores?
- O texto na estória está consistente com outros textos e labels na aplicação?
- O conteúdo dos textos possui erros gramaticais?
- O esquema de cores está consistente com outras cores na app?

Pode-se dizer que a maioria dos defeitos é de fato encontrada durante o desk-check, por razões óbvias, visto que é quando a estória está sendo entregue e quando realmente é possível fazer um teste preliminar na funcionalidade desenvolvida. De qualquer forma, é recomendável uma atenção especial ao kick-off, pois é onde o mal entendido ocorre e as coisas são implementadas diferentemente do ideal.

Vale salientar que estas listas de checagem não devem ser rígidas, escritas na pedra e seguidas à risca. O importante é que o benefício dessa abordagem está em verificar se aqueles passos foram seguidos durante o desenvolvimento da estória. Outro aspecto relevante é que essas listas devem ser personalizadas, uma vez que cada projeto tem suas particularidades e necessidades. Crie a sua lista e veja os benefícios surgindo logo no início. Mantenha o hábito para que isso não seja esquecido com o tempo.

MOCKAR OU NÃO MOCKAR, EIS A QUESTÃO

Fabio Pereira



O recente e polêmico debate “*TDD Morreu?*” entre DHH, Martin Fowler e Kent Beck trouxe à tona o nível de insatisfação relacionado ao **uso excessivo de mocks e stubs** em testes automatizados. DHH expressou fortemente sua opinião sobre o fundamentalismo a respeito do nível de isolamento das classes sendo testadas e criticou a necessidade demasiada que alguns testes têm de segregar completamente todos os seus colaboradores. Esse também foi um dos pontos mais importantes *deste post recente de Martin Fowler*. Durante o hangout, os 3 afirmaram “quase não usar mocks”.

O teste que te faz dormir tranquilo

Kent Beck enfatizou que, no final do dia, como desenvolvedores/programadores, é nossa responsabilidade ter a certeza de que podemos dormir tranquilos a noite sabendo que não quebramos nada. Uma mentalidade bem diferente de antigamente, quando desenvolvedores simplesmente comitavam o seu código e esperavam por outro grupo de pessoas - os testadores - para ter certeza de que tudo ainda funcionava. Este é um dos principais objetivos de uma suíte de testes automatizados,

independentemente desses testes terem sido escritos antes do código (com a prática de “test first”) ou depois que o código havia sido escrito.

A finalidade de testes automatizados é verificar, de forma rápida e confiável, que o sistema ainda funciona e que o novo código escrito não afeta negativamente o código já existente. É este tipo de confiança e feedback que não é alcançado quando mocks e stubs são exageradamente utilizados.

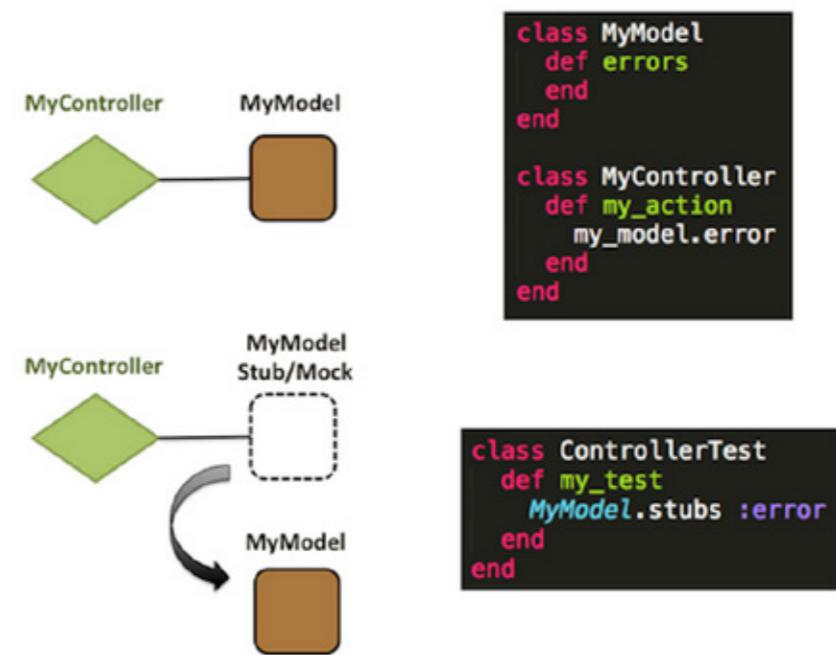
Por que mockar demais é perigoso?

A culpa não é dos *mocks* e *stubs*. Esses dois, assim como qualquer outra ferramenta, são meras vítimas do seu uso indevido. São técnicas extremamente úteis quando precisamos isolar pontos de integração externos, como Web Services e bancos de dados, por exemplo. O perigo existe quando usamos mocks e stubs para isolar classes e métodos que pertencem ao nosso próprio código em camadas que não necessariamente precisariam ser isoladas.

Tautological TDD é um anti-padrão que explica algumas situações nas quais o uso excessivo de mocks e stubs é perigoso. Durante o hangout foi dito: “se faço TDD, eu posso refatorar meu código”. Testes “tautológicos”, que são muito caixa branca, checam mais interações do que comportamento, geralmente precisam ser modificados quando refatoramos nosso código. Se você precisa mudar o seu teste pra cada refatoração que fizer no seu código, como saber se a mudança do código não quebrou nada? Já vi muita gente mudar o código e mudar o teste só pra fazer o teste passar.

TTDD já é perigoso em linguagens como Java e C# e a situação se agrava quando passamos para linguagens dinâmicas como Ruby e JavaScript, nas quais pode-se mockar um método que nem mesmo

existe. Vou ilustrar a seguir um exemplo real, não o único, que já vi acontecer diversas vezes. Digamos que existe um controller (*MyController*) cuja responsabilidade é validar um model (*MyModel*) e exibir os seus erros. O model possui um método “errors”. A imagem abaixo ilustra esse exemplo:



Ao testar o controller, mockistas tendem a isolar o model criando um mock ou stub para o método “errors”. Ruby, com seu dinamismo, e frameworks de testes como *Mocha*, nos permite atingir este nível de isolamento.

Se prestarmos atenção, o método no model é chamado “errors” (plural). Entretanto, o código do controller tem um problema, chama o método no singular, mas o mock/stub faz com que tudo funcione, porque o método mockado também está errado. O teste passa! O que temos aqui? Um falso positivo. Um teste verde dando ao desenvolvedor uma sensação falsa de confiança, quando o

código está, na verdade, quebrado. Esse é apenas um, entre vários, dos exemplos que mostra o perigo do mau uso de mocks e stubs.

Recentemente, depois de fazer uma atualização de uma dependência, descobrimos que o retorno de um método que estava sendo mockado na maioria dos testes unitários havia mudado: antes retornava nil, agora retorna um array vazio. Mais uma vez, todos os nossos testes passaram, mas o código estava quebrado.

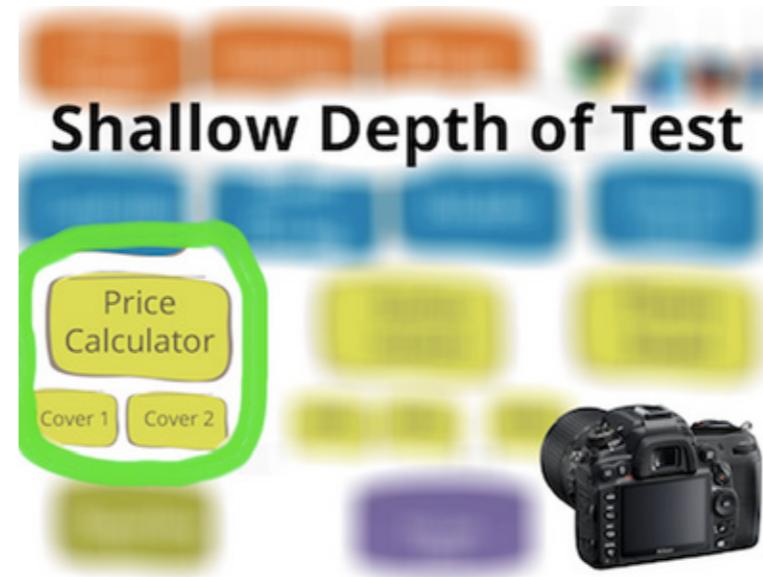
Mais importante do que nomes de métodos errados e valores de retorno é quando o comportamento de uma determinada classe ou entidade é mockada levando em consideração premissas erradas. Quando o propósito dos testes é focado principalmente na verificação da interação entre colaboradores (testes muito caixa branca), essas interações e as expectativas dos mocks nos testes farão todos os testes passarem. Ao ver todos os testes passando, os desenvolvedores irão pra casa dormir tranquilos pensando que nada está quebrado, quando na verdade, alguns destes problemas vão direto para produção, sendo identificados por usuários reais. Já vi isso acontecer várias vezes, problemas que poderiam ter sido identificados por um teste não tão caixa branca, mas acabaram indo para produção. E você, já?

O perigo de ter testes assim é que eles nos dão uma falsa sensação de segurança e confiança. Vemos um build passando, com todos os testes verdes e temos a certeza de que não há nada quebrado. Precisamos pensar melhor quando escrevemos um teste, às vezes é melhor escrever um teste que acesse um grupo de classes e não use tanto mock assim para termos a segurança de que tudo funciona. Outra discussão interessante durante o hangout e alguns outros posts foi o que é uma “unidade” em testes unitários?

Como definir uma “unidade”?

A “unidade” a ser testada é um dos grandes pontos de confusão e debate. Martin nos alerta sobre algumas definições de unidade utilizadas: “o design orientado a objetos tende a tratar uma classe como uma unidade, abordagens procedurais e funcionais consideram uma função como sendo uma unidade”. Uma unidade deve ser um **comportamento (behavior)**. É mais importante testar **O QUE** uma entidade faz, do que **COMO** essa unidade consegue fazê-lo. O critério do que deve ser uma unidade tem que ser definido pelo desenvolvedor escrevendo o código e o teste. Muitas vezes, um grupo de classes pode alcançar um comportamento, portanto, este grupo de classes é a unidade. Pense e defina a *profundidade dos seus testes*, sem nenhum dogma ou fundamentalismo definido pelo paradigma da linguagem que está usando, de forma que eles garantam a certeza de que o comportamento daquela unidade está funcionando.

A imagem abaixo ilustra o conceito de *profundidade de teste*:



Uma unidade não é necessariamente uma classe ou um método ou uma função, mas é o que VOCÊ, desenvolvedor escrevendo o teste e o código, decidir que seja, baseado no seu design e nas suas fronteiras. E, obviamente, se depois de definida a profundidade do seu teste você achar sensato utilizar um mock ou um stub para alguns dos seus colaboradores, vá em frente! Não vamos mockar algo simplesmente porque nos sentimos

obrigados por alguma definição embasada em um paradigma de uma linguagem. Vamos parar com afirmações do tipo: “um teste unitário da classe A não pode acessar a classe B porque não é um teste unitário”. Vamos mockar quando acharmos que devemos, baseados no nosso próprio julgamento de forma que os nossos testes ainda sejam úteis e alcancem o seu objetivo: feedback rápido sobre o funcionamento do nosso código.

3 NOÇÕES BÁSICAS ESSENCEIAIS PARA A CRIAÇÃO DE UMA SUÍTE DE AUTOMAÇÃO PARA APLICATIVOS WEB



Taise Silva

Este artigo foi escrito a fim de compartilhar que existem padrões e ferramentas que, quando combinados, podem oferecer testes automatizados com alto valor de negócios e baixo custo em termos de manutenção do código.

O *Cucumber* é uma ferramenta que suporta Behavior Driven Development (*BDD*), que consiste em descrever o comportamento de um usuário. Dessa forma, as necessidades reais do usuário são descritas. *Selenium WebDriver* é uma ferramenta que simula ações do usuário em navegadores web. Este artigo descreve como

usar o Cucumber, juntamente com Selenium WebDriver, para implementar testes automatizados com alto valor de negócios e de baixa manutenção.

Cucumber é usado para descrever o valor do negócio em uma linguagem natural, por isso permite que equipes de desenvolvimento de software descrevam como o software deve se comportar em texto simples, escrevendo especificações através de exemplos. Uma boa vantagem de escrever especificações com Cucumber é que qualquer um na equipe consegue ler e

entender as especificações em texto simples - de pessoas de negócios a desenvolvedores de software. Além disso, ele ajuda a obter feedback dos stakeholders de negócios para que a equipe construa a coisa certa antes mesmo de começar. Ele também ajuda a equipe a fazer um esforço intencional para desenvolver uma linguagem ubíqua compartilhada para falar sobre o sistema. Outra vantagem é que a especificação é uma documentação viva, porque apesar de ser escrita em texto simples, origina testes automatizados executáveis, como você verá mais adiante no artigo.

A estrutura que o Cucumber usa para as especificações é o formato *Given/When/Then* em conformidade com gramática da linguagem *Gherkin*. A parte Given (Dado) descreve uma pré-condição existente do estado de software antes de começar o comportamento que você está especificando. A seção When (Quando) é o próprio comportamento. O Then (Então) descreve o resultado esperado do comportamento. Por exemplo: dado que me inscrevi para Loja de Livros Online, quando eu acesso a loja com minhas credenciais, então vejo uma mensagem "Bem-vinda à Loja de Livros Online!". Continue lendo para encontrar mais exemplos.

Selenium WebDriver simula as ações do usuário definidas pelas descrições do Cucumber em um browser. Ele é usado para testar automaticamente aplicações web, uma vez que conduz um browser utilizando recursos nativos de cada browser. Selenium ainda é movido por código, de modo que os testes automatizados são escritos em uma linguagem de programação que define as ações do usuário e controla Selenium WebDriver. Há muitas linguagens de programação diferentes que podem ser usadas com Selenium WebDriver, tais como Java, Ruby, Python e JavaScript. No entanto, existe uma linguagem ainda mais simples para descrever cenários de teste: a linguagem natural.

É possível escrever testes automatizados com Cucumber em diferentes linguagens naturais, tais como Português e Inglês, entre mais de 40 outras línguas. Por outro lado, Cucumber não interage diretamente com a aplicação de software. É por isso que é normalmente utilizado em conjunto com ferramentas como o Selenium WebDriver. Desta forma, os testes automatizados servem como documentação, porque podem ser escritos em uma linguagem específica de um domínio e legível do ponto de vista de negócios. Eles também podem ser executáveis, porque possuem a camada do Selenium WebDriver rodando por trás da camada do Cucumber; e podem ser de fácil manutenção porque a camada do Cucumber não precisa saber como as ações do usuário são simuladas, já que este é o trabalho do Selenium WebDriver. Portanto, se a aplicação web muda a forma como o usuário interage com ela, tudo o que precisamos fazer é alterar a camada do Selenium WebDriver e manter a camada do Cucumber como estava. Leia mais para ver exemplos das camadas de Cucumber e Selenium Webdriver.

Existem outras opções de ferramentas para a gestão de casos de teste, tais como *TestLink*, que armazena os casos de teste escritos em linguagens naturais. Essas ferramentas tendem a ser desconectadas do código real e os testes ficam desatualizados rapidamente. A maior vantagem do Cucumber sobre TestLink é que os testes escritos em texto simples são também testes automatizados, que sempre irão falhar quando a aplicação sai de sincronia com os scripts do Cucumber. Isso ajuda a manter a documentação atualizada, porque falhas na execução serão notificadas caso ela não esteja. Cucumber também suporta a execução do caso de teste, uma vez que é mais fácil de identificar quais cenários de funcionalidades estão em execução através da leitura de linguagem natural, em vez de ler a linguagem de programação.

Seguem três passos para escrever testes automatizados com alto valor de negócio e de baixa manutenção: definir o valor do negócio, automatizar testes e refatorar para baixa manutenção.

Definir o valor do negócio

Seguem alguns princípios de BDD utilizados em Cucumber para escrever testes automatizados com alto valor de negócio.

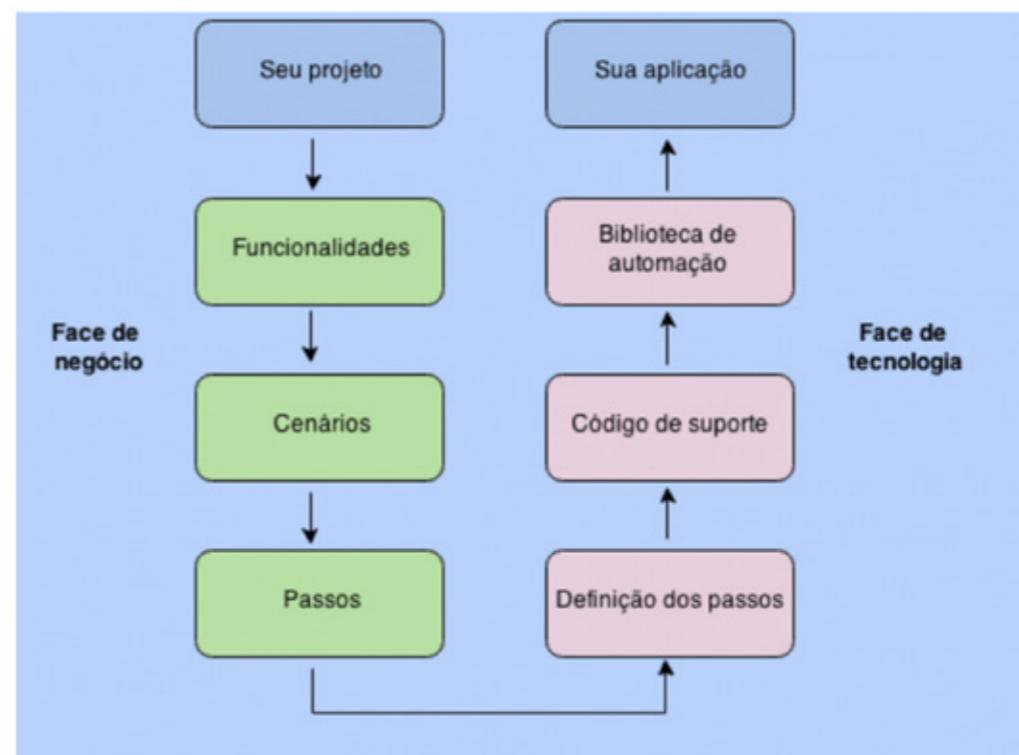
É útil escrevê-los em texto puro antes de implementá-los e garantir que as partes interessadas no negócio dêem feedback sobre os testes descreverem ou não o comportamento correto do software. Se as partes interessadas no negócio derem o feedback após o comportamento do software já ter sido implementado, provavelmente vai levar muito mais tempo para corrigir o código do que seria necessário para corrigir um texto simples que descreve o teste. Então, testes automatizados escritos antes de construir o software trazem valor através da economia de bastante tempo do time de desenvolvimento de software.

Escrever narrativas também traz valor para o teste porque descreve em uma frase qual o motivo de implementar a funcionalidade em primeiro lugar, e ajuda a entender sobre o que são os cenários da funcionalidade. Por exemplo: para que eu possa identificar dinossauros, como um colecionador de ossos, eu quero acessar informações sobre os dinossauros.

Outra qualidade importante em testes automatizados valiosos é que o texto simples usa um vocabulário específico do domínio do negócio para que eles sejam compreendidos por qualquer pessoa da equipe: usuários, stakeholders de negócios e o time de desenvolvimento de software. Isso traz valor para a comunicação da equipe, porque faz com que ela desenvolva uma linguagem ubíqua compartilhada para falar sobre o software e focar no

negócio em vez de descrever testes usando termos técnicos que são apenas mais uma variante de uma linguagem de programação.

Para ter testes automatizados com alto valor de negócio, Cucumber consiste em uma face de negócio e uma face de tecnologia.



Fonte: Tradução da imagem do livro *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*, by Matt Wynne and Aslak Hellesøy

A face de negócio é definida dentro de um arquivo de funcionalidade com a extensão .feature. Ela contém a descrição da funcionalidade e os cenários com os passos escritos em linguagem natural, como o português. Por exemplo:

```

# Language: pt
Funcionalidade: Registro de usuário e login

Como leitor de livros
Eu quero me registrar e logar na Active Store
Para que eu possa ver livros à venda

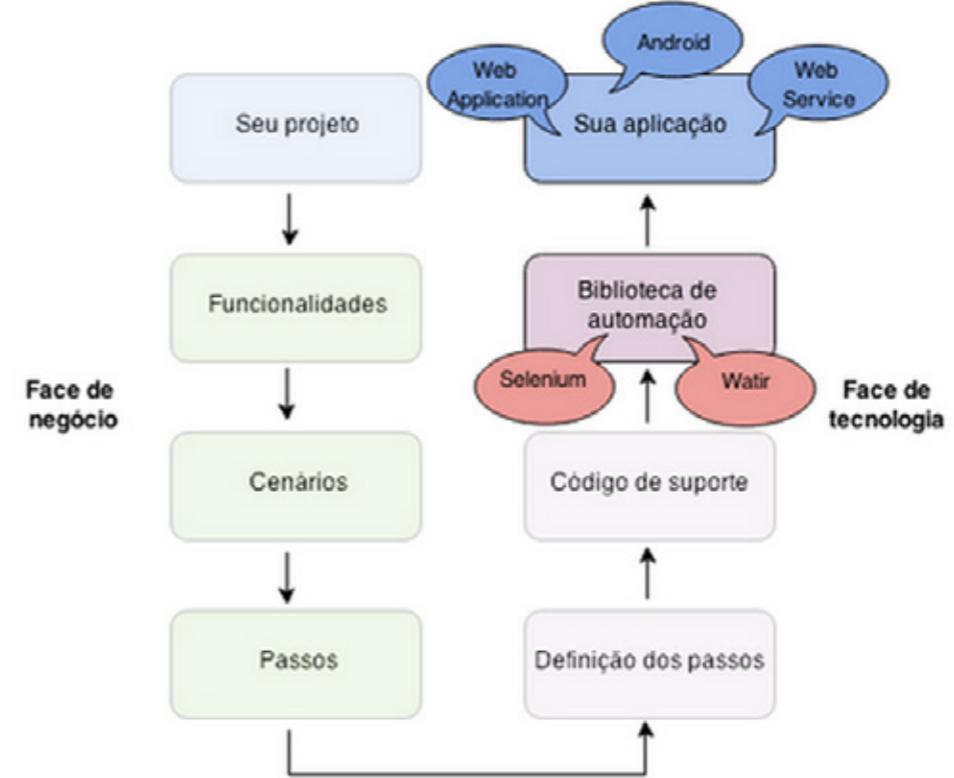
Cenário: Registro de usuário
Quando eu registro o seguinte usuário:
| username | email | password | confirmPassword |
| Taise | taise@gmail.com | 1234 | 1234 |
Então vejo a seguinte mensagem: "Thank you for signing up! You are now logged in."

Cenário: Logar com um usuário já cadastrado
Quando eu logo com os dados:
| email | password |
| taise@gmail.com | 1234 |
Então vejo a mensagem de boas vindas "Welcome Taise!"
```

Fonte: <https://github.com/taisedias/selenium-cucumber>

Automatizar testes

A face de negócio por si só não é um teste automatizado executável que exerce a funcionalidade da aplicação (que pode ser Web, Android ou WebService). A fim de torná-la executável, Cucumber tem uma face de tecnologia que consiste em definições de passo, código de suporte e de biblioteca automação. A definição do passo é implementada utilizando uma linguagem de programação para cada passo do cenário nos arquivos de funcionalidades. Esse código usa uma biblioteca de automação de teste, como Selenium ou *Watir* (outro exemplo de WebDriver usado com Ruby), para acessar a aplicação e executar os testes automatizados.



Fonte: <http://www.slideshare.net/taisedias/cucumber-qanight>

O exemplo seguinte mostra a face de tecnologia implementada em Java. Cada método Java é uma definição do passo descrito no arquivo de funcionalidade do exemplo anterior. O corpo do método usa o código de suporte que chama a biblioteca de automação, que chama o Selenium, que contém o WebDriver que, finalmente, vai acessar a aplicação:

Um padrão amplamente utilizado na implementação de testes automatizados é o padrão *PageObject*, que consiste basicamente em mapeamentos entre os elementos da página da aplicação e uma classe. Ele também define as ações do usuário na página usando seus elementos. O exemplo a seguir é um PageObject LoginPage, que mapeia os elementos da página de login (como o campo de nome de usuário, campo de senha e botão), bem como define as ações como logar na página. As ações conduzem o Selenium para acessar e acionar os elementos da LoginPage.

O exemplo a seguir é um PageObject representando a página de login. Esse objeto mapeia seus elementos individuais com seus

identificadores para o driver usar. Esse deve ser o único lugar onde o identificador é registrado.

```
public class LoginPage {
    private static final String LOGIN_URL = HomePage.HOME_URL + "/login";

    private static final By USERNAME = By.id("login");
    private static final By PASSWORD = By.id("password");
    private static final By SUBMIT_BUTTON = By.name("commit");

    public static void access() {
        get(LOGIN_URL);
    }

    public static void login(User user) {
        waitForElement(USERNAME).sendKeys(user.getEmail());
        waitForElement(PASSWORD).sendKeys(user.getPassword());
        waitForElement(SUBMIT_BUTTON).click();
    }
}
```

Refatorar para baixa manutenção

Código de teste é código, por isso é tão importante refatorar código de automação de teste quanto refatorar código da aplicação. Caso contrário, os testes vão ser tão difíceis de manter que vai ser melhor jogar tudo fora e começar de novo. Esse é um custo que pode ser evitado se seguirmos algumas dicas que ajudam a reduzir os custos de manutenção do código, arquivos de features e definições de passos.

O uso do padrão PageObject torna mais fácil a manutenção de testes automatizados porque as alterações em elementos de página são apenas alteradas uma vez no PageObject. Arquivos de funcionalidades e definições de passo não possuem informações específicas sobre a página e, portanto, não precisam ser atualizados. Em outras palavras, se qualquer elemento na página de login muda (como caminho de URL ou o nome do botão), a

manutenção será feita no LoginPage.java somente, e não haverá necessidade de atualizar o arquivo de funcionalidade (que contém os cenários) e a classe LoginStepDefinition.java.

É importante escrever funcionalidades declarativas, de modo que os cenários sejam descritos como um usuário poderia descrevê-los, o que os torna altamente manutêveis quando comparados com cenários que descrevem apenas clicar em links e preencher os campos de formulário. Por exemplo:

```
Feature: Dinosaurs app access
Scenario: List dinos names
  Given I go to the home page
  When I click the link "List"
  Then the page should contain a list of Dinosaurs
```

```
Feature: Dinosaurs app access
Scenario: List dinos names
  Given I am in Dino app
  When I list dinos names
  Then I see a list of Dinosaurs
```

Outra dica para reduzir o custo de manutenção em cenários de Cucumber é evitar passos que contêm duas ações, porque um deles pode ser reutilizável em diferentes cenários:

```
# Scenario 1
...
When I compose an email to "john@john.com" and send it
...
# step definition
When(/^I compose an email to "(.*)" and send it$/) do |email_address|
  email = Email.new recipient: email_address
  email.send
end
```

```
# Scenario 2
...
When I compose an email to "john@john.com"
And I add "mary@mary.com" as recipient
And I send the email
...
# step definition
When(/^I compose an email to "(.*)"$/) do |email_address|
  @email = Email.new recipient: email_address
end
When(/^I send an email$/) do
  @email.send
end
```

Algumas outras boas práticas que ajudam a manter os testes de Cucumber podem ser encontradas aqui: <http://blog.codeship.com/cucumber-best-practices/>.

Conclusão

Em suma, este artigo descreve brevemente o uso do Cucumber com Selenium WebDriver para implementar testes automatizados com alto valor de negócio e baixa manutenção usando os padrões BDD e PageObject. Todo o código apresentado está no github. Apesar de termos falado sobre Cucumber e Selenium, existem também outras ferramentas de BDD e WebDriver semelhantes que podem ser combinadas para criar uma suíte de automação de alto valor de negócio e de baixa manutenção, desde que você use as práticas sugeridas neste artigo.

ESCREVA TESTES MELHORES EM 5 PASSOS

Marcos Brizeno



Os pontos que vou discutir aqui me ajudaram bastante a ter mais consciência dos meus testes e o que eu poderia melhorar neles. Se você tem experiência em escrever testes, provavelmente sabe do que eu vou falar, mas ainda assim será bom refrescar a memória e adicionar mais referências ao assunto. Se você começou a escrever testes a pouco tempo e está procurando maneiras de melhorar, então veio ao lugar certo!

Antes de começar, gostaria de avisá-lo que está não é uma lista extensiva com tudo o que você precisa saber para se tornar um

especialista em escrever testes, mas eu tentei ao máximo colocar referências para permitir um aprofundamento maior em cada tópico. Adicione essa página aos favoritos e volte de tempos em tempos para investir em aprender um pouco mais sobre cada área específica.

#1 Trate Código de Teste como Código de Produção

Geralmente escutamos pessoas falando sobre a importância de

ter código limpo. O mesmo princípio deve ser aplicado para código de testes. O feedback fraco que você recebe de uma suíte de teste “suja” não lhe permite saber quando você quebrou algum teste ou se basta executar a suíte novamente.

[Este artigo](#) de Robert Martin (Uncle Bob) apresenta uma boa discussão sobre tratar código de teste como código de produção. Ele também dedicou um capítulo inteiro do livro “*Clean Code*” para falar sobre Testes Unitários. Testes são uma das ferramentas mais poderosas para alcançar flexibilidade, então devem ser robustos e limpos. Mudanças acontecem todos os dias e precisamos estar preparados.

A melhor característica para testes limpos é a legibilidade. Se você pode ler e entender um caso de teste, você sabe como o código funciona, como as regras de negócios são aplicadas e consegue achar o que está quebrado. Eu prefiro ter códigos de teste que sejam claros do que códigos sem repetição, conhecidos como DRY - Don't Repeat Yourself (no entanto as ferramentas atuais permitem alcançar ambos). Ter código legível é o objetivo principal.

#2 Utilize Padrões de Testes Para Ótima Legibilidade

Padrões melhoram código de teste da mesma forma que melhoram código de produção. Um padrão que eu gosto bastante é o [Arrange Act Assert](#) (Organizar, Agir e Verificar), ou apenas 3-As. Ele basicamente diz que:

- Arrange (Organizar): Configure suas informações e qualquer outro dado necessário ao teste;
- Act (Agir): Execute a ação que o teste vai validar;
- Assert (Verificar): Veja que o que era esperado realmente aconteceu - ou não;

Outro padrão sobre organização dos testes é o clássico do BDD - Given, When, Then. Martin Fowler faz uma [boa descrição](#) desta técnica. Também existem padrões que vão além de organizar o código. O livro [XUnit Test Patterns](#) de Gerard Meszaros tem vários Padrões, Técnicas e Heurísticas (Code Smells) para testes e pode ser lido [online](#).

#3 Evite Testes Instáveis

O teste realmente quebrou ou basta executá-lo novamente? Se você chegar ao ponto de escutar alguém falando isso ou até mesmo você fazer essa pergunta, provavelmente você tem um problema. Neil Craven tem um [ótimo post](#) sobre o assunto, com algumas dicas de como se livrar de testes não determinísticos, por exemplo reescrevendo o teste um nível mais baixo, entre outras.

Martin Fowler também tem um ótimo [post sobre testes não determinísticos](#) explicando em detalhes o dano que eles podem causar e como melhorá-los, por exemplo colocar testes em quarentena e outras ideias.

O livro XUnit Test Patterns também inclui uma profunda discussão sobre [testes frágeis](#) e as possíveis causas, como falta de isolamento, ou alta sensibilidade da interface.

#4 Teste no Nível Apropriado

Todo teste que você escreve tem um custo. E não é apenas o custo de escrever que é apontado logo de cara, mas também o custo de executá-lo. Pode ser que seja bem pequeno, como testes de JavaScript que rodam em 10 segundos, bem como testes que utilizam Selenium e são executados em paralelo demorando 1 hora para terminar.

Martin Fowler faz uma simples divisão de testes em 3 grupos, Unitários, Serviço e Aceitação, e os organiza em um modelo de *Pirâmide de Testes*. A Pirâmide sugere que você tenha uma grande número de Testes Unitários com uma boa cobertura e feedback rápido, menos testes de Serviço e uma porção bem pequena de testes de Aceitação. Fabio Pereira escreveu um bom *caso de estudo* da Pirâmide de Testes.

Alguns anti-padrões são fáceis de identificar, como a *casquinha de sorvete* de Alister Scott, que parece com uma pirâmide invertida, com muitos testes de aceitação ou manual, e o *cupcake de teste*, que parece um quadrado onde se busca o máximo de cobertura em todos os níveis.

Uma boa metáfora, feita por Fabio Pereira, descrevendo a importância de focar no que é importante para o teste é descrita *neste post*.

#5 Utilize Dublês de Teste

Dublês de Teste - ou Mocks como são mais conhecidos - ajudam a reduzir o custo dos testes ignorando comportamentos desnecessários. Então, eu acho que você deve sim utilizar Dublês de Testes para conseguir testar no nível apropriado. O problema aparece quando os Dublês são muito utilizados e você acaba por não utilizar o que você deveria testar!

Uncle Bob publicou um excelente artigo sobre os *vários tipos de Dublês de Testes* e os seus objetivos. Com certeza conhecer qual tipo de Dublê utilizar em cada situação vai te ajudar bastante a evitar atirar no próprio pé.

Outros artigos que discutem o isolamento de testes e os prós e contras do uso de Dublês de Testes são apresentados e discutidos em mais detalhes por *Gary Bernhardt* e *Fabio Pereira*. Eles devem de dar ideias sobre como utilizar a dose certa.

Conclusão

Além dos pontos discutidos aqui, tem muito mais informação sobre TDD (a série recente de Google Hangouts por Martin, DHH e Kent sobre as vantagens e desvantagens do TDD, chamada de "Is TDD Dead" é bem esclarecedora), BDD, ADD e várias outras maneiras de abordar design de software e testes automatizados. Eu realmente gosto do fluxo de desenvolver testes, pensando sobre design e implementação que TDD e BDD dão, bem como mantendo a atenção na qualidade do código de teste. Mas depende de você achar qual funciona melhor no seu contexto.

Como dito antes, mesmo não sendo uma lista completa, ela vai prover informações suficientes para continuar a ler e aprender mais sobre testes. E, não importa qual caminho siga, vai ser uma experiência de aprendizado útil.

TRÊS FALÁCIAS SOBRE BDD

Nicholas Pufal e Juraci Vieira



BDD, do inglês Behavior Driven Development ou “desenvolvimento orientado por comportamento”, se tornou um termo da moda entre desenvolvedores, analistas de qualidade e analistas de negócios. Apesar de suas fortes ideias, é geralmente mal compreendido. Seguidamente escutamos times que afirmam estar se utilizando de BDD, mas ao olhar mais de perto vemos que o que o time acaba fazendo é usar uma ferramenta de BDD para automação de testes - e não aplicando os conceitos em si. No final das contas, nós escutamos pessoas reclamando dessas

ferramentas, e não sobre as ideias que inspiraram a criação dessas ferramentas. O resultado disso é uma série de queixas que vemos em diversos blogs pela internet - pessoas que passam a rejeitar toda a ideia por trás do BDD, tendo em vista que tentaram usar uma ferramenta sem antes mudar de atitude com relação à forma que desenvolvem software.

Frequentemente escutamos essas três reclamações sobre BDD:

#1 O cliente não se importa com testes

Essa é a principal reclamação. Faz todo o sentido afirmar isso, visto que para o cliente o que realmente importa é um software que atenda às suas necessidades e que funcione. Se você começar uma discussão sobre testes, é muito provável que as pessoas envolvidas com o negócio vão acender a luz verde para se desligar do assunto. Além disso, a palavra **teste** infelizmente carrega consigo uma conotação negativa na comunidade de desenvolvimento de software.

Mas espere um pouco, nós estamos falando de BDD, que é desenvolvimento orientado por comportamento, e isso de nada tem a ver com testes. Testar é algo que você não pode fazer enquanto o software não existir. Testar significa verificar, e em BDD nós estamos tratando de especificar antes de mais nada.

BDD é uma atividade de design, na qual você constrói partes da funcionalidade de maneira incremental guiado pelo comportamento esperado. Em BDD nós saímos da perspectiva orientada a testes e entramos na perspectiva orientada a especificações, o que significa que essa reclamação nasceu mal colocada.

#2 O cliente não quer escrever as especificações

Essa é a segunda reclamação mais usada. Vamos falar sobre ela em duas partes.

"O cliente deve escrever as especificações por conta própria"

Quem faz uso dessa reclamação está afirmando que é esperado que o cliente proponha a solução para o seu próprio problema - problema esse que o seu software é que deveria resolver.

Se o cliente escrever as especificações ele não irá se beneficiar de algo chamado **diversidade cognitiva**, e essa diversidade só aparece em grupos heterogêneos de pessoas trabalhando juntas.

Ele precisa do conselho de engenheiros que sabem os aspectos técnicos do problema que ele está tentando resolver. Ele também precisa do paradigma de um analista de qualidade, o qual vai auxiliar na criação de cenários que ninguém pensou antes. Caso contrário, a solução na qual ele pensou pode ser muito mais complexa do que ela precisa ser.

É injusto reclamar sobre algo que nós, como o time de desenvolvimento, é que deveríamos ser os responsáveis por ajudar nossos clientes.

"O cliente precisa interagir diretamente com a ferramenta"

Essa não é a ideia. O que ele realmente precisa fazer, é prover ao time informações sobre o problema que ele quer resolver, e juntos podem pensar nos exemplos concretos que vão nortear o processo de desenvolvimento.

#3 Você consegue alcançar o mesmo resultado sem uma linguagem específica de domínio (DSL)

Esse argumento é encontrado comumente entre desenvolvedores. A maior parte desses desenvolvedores argumenta que não existe um benefício real em acrescentar mais essa camada - que descreve comportamento em linguagem natural - visto que ela apenas adiciona complexidade e faz com que o conjunto de testes fique lento.

Se nós olharmos essa reclamação focando em uma tech stack em Ruby, isso geralmente significa que ao invés de usar o *Cucumber*,

você pode usar o *Capybara + RSpec* para obter o mesmo benefício e ainda por cima ter uma melhor performance ao rodar seus testes.

```
feature "Signing in" do
  background do
    User.make(:email => 'user@example.com', :password => 'caplin')
  end

  scenario "Signing in with correct credentials" do
    visit '/sessions/new'
    within("#session") do
      fill_in 'Login', :with => 'user@example.com'
      fill_in 'Password', :with => 'caplin'
    end
    click_link 'Sign in'
    expect(page).to have_content 'Success'
  end

  given(:other_user) { User.make(:email => 'other@example.com', :password => 'rous') }

  scenario "Signing in as another user" do
    visit '/sessions/new'
    within("#session") do
      fill_in 'Login', :with => other_user.email
      fill_in 'Password', :with => other_user.password
    end
    click_link 'Sign in'
    expect(page).to have_content 'Invalid email or password'
  end
end
```

A verdade é que essa comparação não faz sentido. É como comparar maçãs e laranjas: são coisas totalmente distintas.

O benefício em se utilizar de uma linguagem específica de domínio que pessoas do negócio podem ler - como as especificações que escrevemos no Cucumber nesse caso - vai além do que a perspectiva de um desenvolvedor é capaz de compreender. Não se trata de código, se trata de aprimorar a comunicação entre todos os membros do time.

Ou seja, é ter os analistas de negócio dialogando com os desenvolvedores e analistas de qualidade, todos eles aprimorando

aquele único arquivo que é uma maneira não abstrata de demonstrar ideias - o arquivo das especificações, que descreve todos os cenários da funcionalidade. Além disso, estarão usando suas diferentes capacidades cognitivas para juntos pensarem em qual o melhor caminho para transformar uma especificação na concretização das necessidades do negócio.

Um caso de sucesso usando BDD

O quanto complicado seria para você explicar para uma criança de 3 anos de idade como uma transação bancária funciona? O mesmo desafio se aplica durante o desenvolvimento de software, visto que o domínio do cliente pode, por diversas vezes, ser bastante vago e nebuloso para o time de desenvolvimento.

O ser humano precisa de exemplos para compreender um tópico. Exemplos reais são uma excelente forma de se comunicar, e no nosso dia-a-dia nós os usamos, sem nem mesmo perceber. Ao trabalhar com exemplos reais nós nos comunicamos melhor, pois as pessoas serão capazes de se relacionar com eles mais facilmente.

Isso tudo é muito mais perceptível quando o domínio do negócio é complexo.

Um bom exemplo disso é um projeto no qual trabalhamos, de um banco de investimentos. Como é de se esperar em um domínio desses, as terminologias eram muito complicadas, tornando um tanto quanto difícil a vida dos desenvolvedores na hora de manter um diálogo com os analistas de negócios do banco.

No intuito de nos comunicarmos melhor, parte do nosso processo era, antes de começar uma história, ter o par de desenvolvedores fazendo uma rápida chamada de áudio/vídeo

com o analista de negócios responsável por ela. Esse analista por sua vez compartilhava com os desenvolvedores o arquivo com as especificações - também conhecido como feature file - o qual continha todos os cenários nos quais ele pensou.

Visto que neste time não haviam analistas de qualidade, é importante mencionar que durante essa sessão um dos desenvolvedores deveria manter um pensamento mais alinhado com o de um analista de qualidade - focando em aprimorar os cenários já criados e sugerindo a criação de cenários que ainda não haviam sido explorados - enquanto o outro desenvolvedor focaria mais nos desafios técnicos da implementação, como por exemplo sugerir mover um cenário para uma especificação em nível de código - que oferece um feedback mais rápido no conjunto de testes. Todos também podiam sugerir mudanças aos passos de um cenário, caso isso fizesse mais sentido à compreensão de todos.

Ao utilizar esse processo os analistas de negócios expandiam o seu conhecimento ao compreender melhor os desafios técnicos de cada cenário e os desenvolvedores conseguiam ter uma ideia mais clara das necessidades do negócio, facilitando na compreensão do que realmente precisava ser desenvolvido. Além disso, sempre que mudanças fossem necessárias, apenas aquele único pedaço de informação seria mexido, o que significa que todo o time estaria sempre atualizado.

Indo mais fundo no assunto

Para finalizar, a principal ideia por trás do BDD é o seu foco em prevenir falhas de comunicação, o que significa ter todos no time comunicando de maneira mais frequente, melhor e baseados em exemplos reais - não somente abstrações e requisitos imperativos.

Esperamos que este artigo ajude as pessoas a entender melhor os benefícios por trás do BDD - e que fique claro que as ferramentas de BDD são apenas complementares a essa metodologia ágil completa que o BDD é.

Se você quer ir mais a fundo no assunto, nós sugerimos as seguintes fontes:

- “Specification by example” do Gojko Adzic
- “The RSpec Book” do David Chelimsky
- [*Dave Astels and Steven Baker on RSpec and Behavior-Driven Development*](#)
- [*Gojko on BDD: Busting the Myths*](#)

APENAS EXECUTE O BUILD NOVAMENTE - ELE DEVE FICAR VERDE



Neil Philip Craven

"Apenas execute o build novamente - ele deve ficar verde".

Você escuta isso no seu projeto? Seus testes são *flaky* (instáveis ou não-determinísticos)? O build muda de verde para vermelho e vice-versa sem nenhuma razão aparente? O que você tem feito a respeito?

Recentemente ingressei em um projeto que tinha esse problema. É relativamente normal encontrar esse problema e é ainda mais comum contornar esse problema apenas executando os testes novamente. Alguns times inclusive adicionam scripts

para, automaticamente ao final do build, re-executar testes que falharam.

Testes flaky ou sistema flaky?

Mas como você é capaz de diferenciar um teste *flaky* de um sistema *flaky*? Nós certamente não conseguimos - mais adiante descobriu-se que o sistema era *flaky* e nossos usuários tiveram problemas com isso. Esse não é um problema simples de se resolver. Aqui estão duas abordagens que nos ajudaram:

1. Identifique os testes não determinísticos:

Você deve identificar se o problema está no teste ou se está no sistema. Pense a respeito da sua suíte de testes - você possui algum teste não-determinístico? Você provavelmente possui, a menos que tenha sido muito cuidadoso. Você pode tentar descontruir seus testes não determinísticos e implementá-lo em um nível mais baixo. Você perderá um certo grau de cobertura mas os resultados do teste serão confiáveis. Você pode acompanhar os logs em tempo real e, quando apropriado, verificar a interface gráfica da máquina de testes, enquanto os testes estão sendo executados, para saber se eles se comportam da maneira esperada.

2. Não os ignore, conserte-os.

Entender o seu problema não faz com que ele desapareça. Você precisa parar de adicionar novas funcionalidades ao redor e consertar esse problema. Usando a abordagem de ignorar o problema, nossa suíte de testes funcionais passou de praticamente verde para praticamente vermelha. O dia em que tivemos 13 dos 17 builds vermelhos foi a gota d'água e nós decidimos que medidas drásticas deveriam ser tomadas. Qual foi nossa ação? Simplesmente passamos a ter mais disciplina em relação ao processo que nós impusemos. Nos decidimos que removeríamos a opção de re-executar a suíte de testes caso o código-fonte não tivesse sido alterado, e nós reforçamos a regra de que você apenas poderia enviar suas modificações, para o servidor de controle de versão, se elas estivessem relacionadas à correção do build.

Como era de se esperar, no final do dia seguinte haviam 7 pessoas ao redor da máquina de build tentando consertar esses builds instáveis. Como resultado, nós descobrimos alguns testes mal escritos e encontramos seis problemas reais com o sistema. O build ainda estava vermelho, mas ao invés de cruzarmos os dedos nós tínhamos seis novos erros para consertar.

ENTREGA CONTÍNUA COM BUILD QUEBRADO E CONSCIÊNCIA LIMPA

Lucas Medina



Já pensou em fazer Entrega Contínua no seu projeto? E Entrega Contínua com o build eventualmente quebrado? Pode parecer perigoso, mas na verdade não é o fim do mundo e é mais comum no dia a dia dos projetos do que você pensa. Concordamos que a Entrega Contínua deve ser feita com a pipeline de build toda verde, e esse é o nosso objetivo. Este artigo compartilha a experiência de um projeto específico.

Pense em um cenário no qual em torno de 5 builds vão para ambiente de produção todos os dias, mesmo que o build esteja quebrado; no qual fazemos o deploy da produção mesmo que o

build de desenvolvimento não esteja todo verde e nem todos os testes estejam passando.

Pense em um projeto de gerenciamento de infraestrutura na nuvem que recebe aproximadamente 10 mil visitas únicas diárias, com 5 times desenvolvendo e empurrando código de dois locais diferentes. Manter o build verde num projeto em que qualquer desenvolvedor trabalha em qualquer parte do código é um grande desafio, porque todos os times têm autonomia para desenvolver e entregar.

Para iniciar os trabalhos, criamos um branch de vida curta (short lived branch) no master e uma flag de funcionalidade que permite aos desenvolvedores continuar empurrando código mesmo que a funcionalidade não esteja completa, uma vez que ela fica escondida pela flag até estar completa - e desenvolvemos a estória. Ao final integramos novamente ao master, rodamos os testes e mandamos ver, empurramos para produção. É importante reforçar que este deve ser um branch de vida curta. Se cada time ou funcionalidade acabar tendo um branch vida longa, isso poderia gerar um problema em si e um verdadeiro inferno de merges.

Para fazer tudo isso acontecer precisamos de uma suíte de testes boa e estruturada. Essa suíte não é de responsabilidade de um time específico, mas de todos os times envolvidos. Uma estória entregue significa necessariamente testes unitários e de aceitação feitos e integrados. Sendo assim, toda nova funcionalidade está coberta por testes automatizados, garantindo a prevenção de defeitos quando novas funcionalidades são adicionadas.

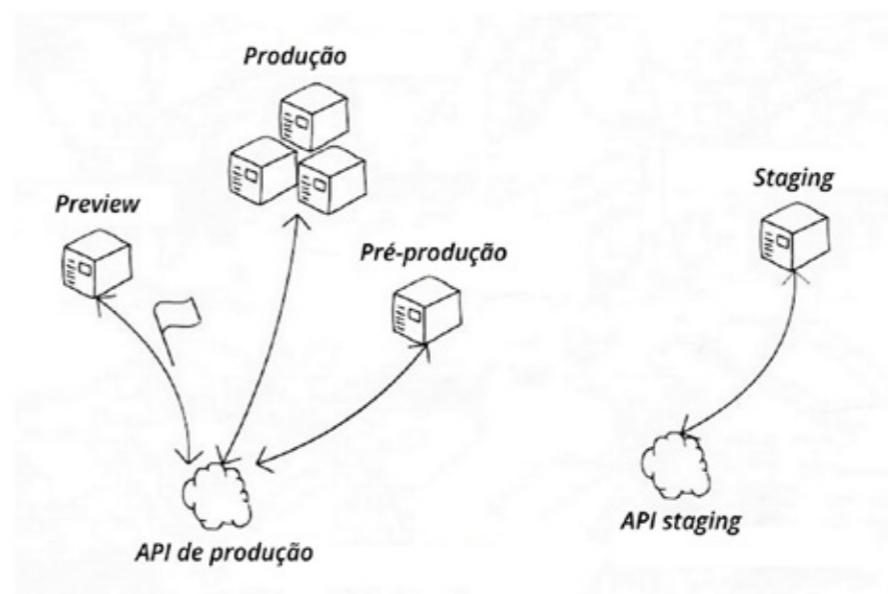
Um dos nossos times de desenvolvimento nesse projeto era composto apenas por testadores. Estes eram responsáveis por automatizar áreas que não foram cobertas no desenvolvimento das estórias, manter as suítes de testes estáveis e aprimorá-las, além de implementar os chamados “caminhos não felizes” e “corner cases”.

E quais eram os ambientes em que estávamos trabalhando? Bom, havia quatro tipos:

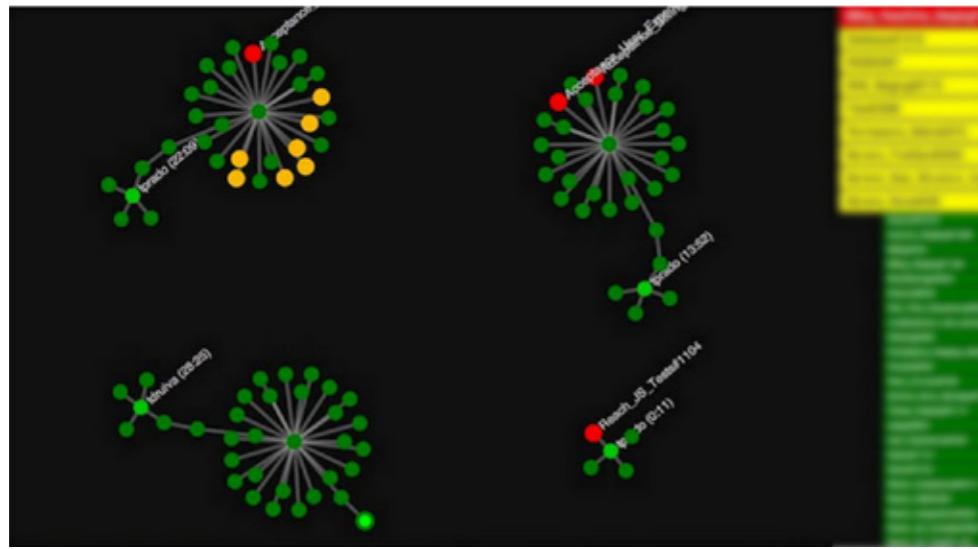
- Um deles era o staging, um ambiente de adaptação, e tinha o objetivo principal de encontrar erros de integração com APIs externas.
- O ambiente de pre-produção era exatamente igual ao de produção, ali tudo já precisava estar funcionando

perfeitamente.

- Lembra das flags de funcionalidade? Pois é, enquanto as estórias estavam sendo desenvolvidas, o ambiente “em preview” era onde as histórias que estavam escondidas no ambiente de pré-produção eram testadas.
- Por fim, produção era onde a festa acontecia. Era o ambiente no qual o cliente usava a aplicação.



Todo commit que integramos ao master passa pela suíte de teste. Através do fabuloso radiador de integração, os times acompanham o estado da integração. Caso tudo passe, maravilha, vai direto pra produção. Caso alguma suíte falhe, investigamos a causa e decidimos sobre fazer a entrega assim mesmo ou não. Se decidimos que sim, o build quebrado vai para produção com a segurança de que se trata ou de um teste/uma suíte instável, de uma parte do código que não foi alterada ou simplesmente de que o teste que falhou não era expressivo o bastante para bloquear a entrega.



Como o número de integrações aumentou muito à medida que os times crescam, rodar a suíte inteira toda vez tornou-se insustentável. Por isso, criamos o conceito de “ônibus” de integração. Esse “ônibus” “recolhe” os “passageiros” a cada hora, integra em um pacote único e executa a suíte de testes.

Como sabemos, nem tudo são flores, vários são os desafios nessa modalidade de Entrega Contínua:

- suítes muito lentas retardam a entrega;
- testes que falham aleatoriamente são inúteis, pois não são confiáveis;
- dependência de APIs externas causa falhas em testes que não são parte da camada em desenvolvimento;
- a quebra das funcionalidades em produção acontece e precisamos pensar na causa raiz para que o problema não se repita.

Algumas considerações finais sobre essa abordagem de Entrega Contínua: quando testes de aceitação estão levando tempo demais, talvez seja um sintoma de que sua aplicação está grande demais ou que a unidade de deployment não é granular o suficiente. Crie flags de funcionalidade: isso facilita sua vida se o time precisa desabilitar uma funcionalidade em produção. Mantenha o foco e a mentalidade nos testes, porém não seja tão rígido e religioso, avalie os erros e use seu conhecimento para tomar decisões bem embasadas.

DevOps, que foca na preparação dos ambientes e na Engenharia de Release ao lidar com a pipeline de deploy no processo de desenvolvimento, não é opcional; o ideal é ter uma ou duas pessoas dedicadas a essa área, para que você possa fazer Entrega Contínua com menos dores de cabeça. Isso porque às vezes precisamos de um par de olhos dedicado especialmente à qualidade da Automação de Infraestrutura e da Engenharia de Release durante um tempo um pouco maior. Para garantir que esse par não trave, e também para que todos no time entendam e adquiram experiência com a Engenharia de Release e o trabalho com deploy para produção, recomendamos fazer uma rotação entre vários membros do time ao longo do tempo.

A combinação da alternância de funcionalidades com a habilidade de ativar suítes de teste correspondentes sob demanda ajudam a garantir que tanto as funcionalidades completas como as incompletas possam ir ao ar sem comprometer a qualidade.

MELHORANDO A QUALIDADE DE PROJETOS ATRAVÉS DO PODER DAS NUVENS

Max Lincoln



Seus processos de qualidade e ferramentas mudam para construir uma aplicação na nuvem? A resposta óbvia é - "depende". Mesmo tendo a mais clara definição entre as buzzwords recentes, existem vários tipos de nuvem e muitas formas de integrá-la com sua aplicação ou processo de desenvolvimento. Se você tem um time experiente, capaz de escolhas bem informadas, então essas opções são uma grande recurso em favor da qualidade. O time pode usar a nuvem com uma ferramenta para assegurar qualidade, ao invés de torná-la apenas mais um aspecto que precisa ser testado.

O que é a nuvem?

Existem algumas poucas definições de nuvem que eu acredito serem úteis.

A primeira, do Gartner Group, descreve um caso singular de utilização que é uma ótima forma de promover qualidade da nuvem:

Ambiente de laboratório virtual. Esses consumidores geralmente tentam prover infraestrutura auto-gerida para um grupo de usuários técnicos, como desenvolvedores, cientistas ou engenheiros, para o propósito de teste e desenvolvimento, computação científica ou outro tipo de computação em lote.

(“Quadrante Mágico para Infraestrutura Pública de Nuvem como Serviço” - Gartner)

A natureza auto-gerida é muito importante. Se você precisa um ambiente por apenas 30 minutos para um experimento, então é um desperdício gastar mais de 30 minutos orçando ou agendando a criação de tal ambiente. Um dos meus colegas enfatiza: “Eu posso iniciar máquinas virtuais em minutos, destruí-las, e repetir esse processo quanto frequentemente eu precisar. Todo o resto é infraestrutura do passado. Se eu tenho que mandar e-mail para alguém, abrir um chamado, submeter um pedido para conseguir uma máquina virtual, etc. então é infra-estrutura do passado.”

A outra definição importante é a *definição oficial de computação em nuvem* do Instituto Nacional de Padrões e Tecnologia (NIST). A definição (de acordo com o *sumário feito por Martin Fowler*) contém:

- Cinco características essenciais: serviço por demanda auto-gerida, ampla conectividade, resource polling, elasticidade rápida e mensurável.
- Três modelos de serviço: software, plataforma e infraestrutura (tudo como serviço).
- Quatro modelos de instalação: privada, comunitária, pública e híbrida.

Vantagens da Nuvem em favor da Qualidade

Vejamos como podemos tirar vantagem dessas características da nuvem em favor das *8 dimensões da verificação de qualidade*. Na parte I iremos examinar como a nuvem ajuda com Performance, Funcionalidades, Confiabilidade e Conformidade. Na parte II iremos descrever os efeitos positivos da nuvem em termos de Durabilidade, Serviabilidade, Estética e Qualidade Percebida.

#1 Performance

Performance refere-se às características primárias de operação de um produto e envolve atributos mensuráveis. Eu acredito que performance é a funcionalidade de negócio #1. Têm havido muitos estudos mostrando que mesmo a menor redução de performance pode ter um grande impacto no negócio. A nuvem ajuda você a manter sua performance de várias formas.

a. Resource pooling e elasticidade rápida lhe dão uma extra força quando você precisa, e sem gastar mais do que o necessário.

Já que elasticidade rápida é uma característica essencial de computação em nuvem, você tem a opção de adicionar mais recursos computacionais para atender picos de utilização ou acompanhar tendências recorrentes na demanda.

Por exemplo, você pode ter mais servidores rodando durante a semana do que nos finais de semana. É possível fazer esses ajustes finos sem sair do orçamento, porque outra característica essencial - ser mensurável - permite que você seja cobrado por hora (ou mesmo em intervalos menores de tempo) pelos recursos.

b. Ampla conectividade e Redes de Distribuição de Conteúdo (CDN) permitem a você estar mais próximo de seus clientes e parceiros de negócio.

A característica de estar amplamente disponível globalmente torna fácil tirar vantagem de redes de alta velocidade, rápidos serviços de resolução de nomes (DNS), e CDNs como Akamai para acelerar sua aplicação. Isso é especialmente importante para aplicações móveis, nas quais redes são mais lentas, nas quais cada byte ou milisegundo de latência entra na conta.

Um CDN pode ajudar você a cachejar conteúdo próximo de seus usuários, comprimir seu conteúdo, e garantir que o mesmo suporta requisições condicionais. Todas essas medidas podem tornar seu site显著mente mais rápido.

c. Empresas de “Testing as a Service” permitem que você possa rapidamente alocar mais recursos e rodar testes de performance.

Existem muitos serviços na nuvem que ajudam a testar ou aumentar a performance de seu site. É difícil e caro criar seu framework para testes de performance com o poder de rapidamente aumentar as requisições para simular um grande número de usuários. Geralmente frameworks caseiros passam por problemas porque eles enfrentam desafios relacionados a rede e performance antes mesmo da aplicação a ser testada, ou porque faltam funcionalidades para simular usuários de outras partes do mundo, ou por conexões lentas. Ao invés de criar, é possível usar um serviço como o [Blitz.io](#). Blitz pode de forma econômica simular 1000 usuários na rede de sua escolha por 1 minuto ou mesmo escalar isso para 50000 usuários por 20 minutos.

Se você precisa testar um fluxo mais complexo, você pode usar algo como [BlazeMeter](#), que permite rodar testes com JMeter ou Selenium a partir da nuvem.

#2 Funcionalidades

Funcionalidades são características adicionais para tornar um produto ou serviço mais atraente e entregar mais valor para o usuário. Todavia, essa é apenas uma hipótese até que você ponha a funcionalidade na frente de usuários reais e valide seu valor. Funcionalidade que é bem testada mas não útil não é uma funcionalidade, é uma “inutilidade de alta qualidade”. A nuvem pode te ajudar a não tornar seu produto “bloated” e rapidamente testar quais funcionalidades estão implementadas corretamente.

a. Ferramentas na nuvem permitem que você sempre tenha poder de teste suficiente.

Para testar funcionalidades rapidamente, você pode tirar vantagem das características elásticas da nuvem. Você pode usar ferramentas como [jclouds-jenkins] para se certificar que seu pipeline de Entrega Contínua pode lidar com um pico em commits sem ficar sem máquinas. Você pode usar ferramentas como Vagrant (com sua escolha de providers como VMWare ou vagrant-rackspace, e provisionamento com Puppet, Chef ou Ansible) para rapidamente criar um ambiente de testes e destruí-lo assim que os testes terminarem. Você também pode usar serviços de teste SaaS, como SauceLabs, Appium, Xamarin, Appium ou Soasta, para tornar a execução de testes mais rápida através de paralelização ou mais abrangente rodando contra vários browsers ou dispositivos móveis.

b. Os vários modelos *aaS lhe dão várias opções de construir-vs-comprar. IaaS toma controle completo, PaaS lhe permite evitar algumas decisões arquiteturais, DBaaS, Logging-as-a-Service, etc. deixam certo “aspecto” de sua aplicação a cargo de um fornecedor.

Uma estratégia para evitar desperdício é manter sua aplicação

focada em resolver o problema específico de seu domínio, e confiar em parceiros SaaS para serviços relacionados. Um bom exemplo é o envio de e-mail. Não é tão fácil processar rapidamente um template de e-mail e enviá-lo para um grande número de interessados. Adicione a isso a necessidade de processar pedidos de usuários que não querem mais receber e-mails, reclamações de spam, agendamento de campanhas via e-mail, analytics e muito mais. Provedores como [Mailgun](#) lhe permitem terceirizar esses problemas e focar em seu negócio. Existem provedores disponíveis para várias necessidades auxiliares, tais como processamento de vídeo através do [Zencoder](#) ou processamento de pagamento com o Paypal.

c. Deployments fáceis tornam testes A/B, MAB (Multi-Armed Bandit) e outras opções mais viáveis.

Aplicações como [Mailgun](#) suportam analytics e testes A/B, dois métodos que ajudam a garantir a qualidade das funcionalidades entregues. Analytics, A/B e testes multivariados tornam possíveis experimentos para verificar quais funcionalidades ou conteúdo proveem mais valor. Outros serviços na nuvem que ajudam a implementar esses experimentos incluem [Google Analytics Content Experiments](#), [Optimizely](#) and [Visual Website Optimizer](#). Você também pode integrá-los diretamente em sua aplicação, usando bibliotecas como a Ruby gem [split](#).

Usando essas técnicas, você pode:

- Otimizar a relação funcionalidade-desperdício utilizando A/B ou testes multivariados para provar que novas funcionalidades entregam valor antes de decidir por lançá-las para todos os usuários.
- Minimizar a quantidade de código auxiliar e testes que você mantém através da utilização de serviços como Mailgun e Zencoder.

- Rapidamente testar as funcionalidades restantes através de escalonamento elástico da infraestrutura de teste ou utilizando serviços na nuvem de provedores de testes.

A razão pela qual recomendamos tantas soluções SaaS é porque eles são altamente confiáveis. PayPal compartilhou que a *infraestrutura de nuvem é o segredo para a confiabilidade de seu serviço*.

#3 Confiabilidade

Confiabilidade é a probabilidade de um produto não falhar dentro de um período específico de tempo, algo que pode ser especialmente crítico para certos domínios. Confiabilidade pode ser a dimensão da qualidade onde a nuvem provê a maior vantagem. Mesmo que componentes individuais na nuvem possam falhar, a nuvem torna fácil “projetar para a falha”. Dessa forma você terá aplicações resilientes que podem sobreviver mesmo aos problemas mais severos.

a. A nuvem permite que você distribua sua aplicação através de múltiplos data centers ou mesmo provedores, garantindo Recuperação de Desastres.

A maioria de provedores públicos de nuvem oferecem serviços de vários data centers ao redor do mundo. Você pode tirar vantagem disso para facilmente construir uma aplicação altamente redundante ou criar vários sites para recuperação de desastre. Se isso não for suficiente, você pode usar RightScale, que ajuda a gerenciar múltiplos serviços de nuvem no mundo inteiro, como também seus serviços de nuvem privada. Se você quiser garantir que sua aplicação nunca cairá, você pode executá-la de vários lugares do mundo com Rackspace, Amazon e com sua própria instância de OpenStack.

b. A nuvem também lhe dá capacidades de back-up e armazenamento infinito, então retenção de backup é apenas uma decisão de custo.

A nuvem torna extremamente simples gerenciar backups. Serviços Object Storage como Amazon S3 e Rackspace Cloud Files oferecem redundância de dados e armazenamento virtualmente ilimitado. Você pode simplesmente clicar (ou agendar) sempre que quiser um backup.

c. Provedores de Nuvem e empresas de *aaS oferecem monitoramento de todos os tipos.

Existem ótimos sistemas de monitoramento baseados na nuvem, então você pode detectar a deterioração de um serviço antes que isso se torne um grande problema. Você pode definir alertas de monitoramento para a maioria das infraestruturas de nuvem através do próprio fornecedor, e pode também integrar sua aplicação com ótimos serviços de gestão de logs como [NewRelic](#), [Loggly](#) ou [Splunk Storm](#).

#4 Conformidade

Confirmidade é a precisão com a qual o produto ou serviço atende padrões específicos. A nuvem pode lhe ajudar com necessidades de conformidade ou compliance, seja evitando a necessidade de conformidade com um padrão utilizando um provedor de serviço, seja provendo infraestrutura consistente para ajudar a garantir conformidade.

a. Use opções de SaaS para evitar a necessidade de conformidade.

Geralmente, você pode delegar uma funcionalidade auxiliar para um provedor SaaS em conformidade, então você não precisa lidar

com tal requerimento diretamente. Você pode usar a expertise do Mailgun para lidar com o [CAN-SPAM Act](#), para, por exemplo, certificar-se que seus e-mails terão um link para remoção da lista e um endereço físico de correio. Você pode deixar que o PayPal ou [Braintree Payment Gateway](#) lide com a maior parte das suas preocupações relacionadas a compliance PCI-DSS.

Se você precisa de auditoria forte e garantias de segurança, existem serviços como [Dome9](#), que oferecem soluções avançadas de segurança, compliance e auditoria para sua infraestrutura de nuvem.

b. Usar uma camada de caching oferecida pela Rede de Distribuição de Conteúdo (CDN) da nuvem evita problemas com clientes gerados por não conformidade.

Não esqueçamos daqueles testes que não intimidam tanto. Você está em conformidade com IEEE ou padrões W3C? Se você está configurando servidores de cache por conta própria é possível que um pequeno erro de configuração cause erros ou deixe mais lenta sua aplicação para alguns usuários. Se você está usando a camada de caching oferecida pelo CDN da nuvem então é improvável que eles estejam configurados de forma a causar problemas.

Ferramentas como RedBot, que “acham erros comuns de protocolo”, Google Page Speed ou YSlow podem ajudar ainda mais, com verificações no caching, negociação de conteúdo e compressão.

MELHORIA DA QUALIDADE COM A NUVEM - PARTE II



Max Lincoln

Na *primeira parte* do artigo sobre qualidade de processos na nuvem examinamos como a nuvem ajuda com Performance, Funcionalidades, Confiabilidade e Conformidade. Na parte II descreveremos os efeitos positivos da nuvem sobre a Durabilidade, Serviciabilidade, Estética e Qualidade Percebida.

Durabilidade

Durabilidade mede a extensão de vida do produto. Quando um produto pode ser reparado, torna-se mais difícil estimar sua

durabilidade, já que ele será usado até não ter mais retorno econômico para ser operado.

O segredo da durabilidade é usar peças intercambiáveis. Um vela de ignição típica dura de 16.000 a 32.000 quilometros. Um carro durará muito mais - contanto que você substitua suas velas de ignição, óleo, os pneus, os freios e outras peças. Quando a vela de ignição alcança o fim da sua expectativa de vida, ela é substituída e não reparada.

O mesmo deveria acontecer com os servidores. Servidores, especialmente em nuvem, seriam supostamente commodities idênticas. Mesmo existindo alguns tipos de servidores - servidores web, servidores de banco de dados, servidores app X - servidores do mesmo tipo deveriam ser idênticos. Por isso se um servidor para de funcionar corretamente, deveria ser mais simples e fácil substituí-lo do que consertá-lo.

Servidores que são facilmente destruídos e substituídos são chamados de *Servidores Phoenix*. Nós recomendamos que servidores renasçam das cinzas mais frequentemente, ao invés de esperar eles quebrarem. Você não iria esperar seu carro parar de andar para substituir o óleo. Por que deveríamos esperar nosso website cair para substituir o antigo servidor web?

Esse processo não se aplica apenas ao hardware. Ele envolve reverter a “deriva de configurações” (ou *configuration drift*) - mudanças ao longo do tempo que começam a diferenciar servidores que deveriam ser idênticos. Algumas vezes isso acontece mesmo que você esteja usando ferramentas de automação de infraestrutura como Puppet e Chef. Seus servidores antigos têm somente a tech stack atual como os servidores novos ou eles tem algum resquício de software antigo que você esqueceu de desinstalar?

Serviciabilidade

Serviciabilidade (ou Recuperabilidade) é a velocidade com a qual um produto pode ser colocado em serviço quando danificado, bem como a competência e o comportamento do técnico de manutenção.

A nuvem é incrivelmente fácil de ser mantida e nos possibilita construir sistemas e produtos, de TI como serviço, de uma maneira

jamais anteriormente criada. Mais uma vez, a característica de self-service da nuvem é extremamente importante, como um recurso de pooling. Se precisamos executar uma mudança ou manutenção em um serviço, nós podemos simplesmente removê-lo do pool, repará-lo e recolocá-lo em serviço.

1. Reduzir downtime e evitar janelas de manutenção para deploy às 3am

Técnicas como *Blue/Green* deployments são fáceis na nuvem. Essa é uma estratégia de deploy na qual você tem o balanceador de carga com dois pools: azul e verde. Somente um pool está ativo por vez, e você pode fazer deploys com zero downtime utilizando a alternância entre os dois pools. Então, se o pool azul está ativo, você faz o deploy das suas mudanças no pool inativo, nesse caso o verde. Uma vez que você rodou seu smoke test nos servidores verde, você pode desativar o pool azul e ativar o pool verde. Se acontecer qualquer downtime, deverá ser de cerca de um segundo. Você então monitora seus sistemas e o feedback de usuários para verificar quaisquer sinais de problemas. No primeiro sinal de problema, você só precisa trocar novamente para o pool azul. Novamente, isso só durará um segundo. Você não reverteu e perdeu todas as suas mudanças do pool verde - você só está comprando mais tempo para poder investigar os problemas que foram apontados. E se você determinar que é um problema mínimo ou um alarme falso, você pode dar uma segunda chance ao pool verde.

2. ... Ou considere iniciar um substituto

O padrão azul/verde é o mais fácil para visualizar, implementar e zerar o downtime de deploy dos sistemas, mas há outras opções. Algumas organizações aplicam algo parecido com o padrão azul/verde, mas revertem para o seu sistema em um site de

Reparação de Desastre. Um bom benefício dessa abordagem é exercitar alguns dos seus processos de Reparação de Desastre e equipamentos a cada deploy - uma coisa que muitas empresas deixam passar. O padrão de deploy canário é uma técnica que redefine o que “produção” e “versão de produção” significam. Se você tem um grupo de servidores, você poderia fazer o release de uma nova versão em somente um deles. Você verifica (exatamente como no caso dos canários em uma mina de carvão) se esse servidor apresenta qualquer problema, e reverte o release caso necessário. Se não encontrar problemas, você pode continuar um novo release de maneira incremental - 10%, 25%, 50%, 80% e finalmente 100% em seus servidores. Essas estratégias não exigem fluxos de trabalhos manuais complexos e propensos a erros.

3. A nuvem é altamente controlável, com SDKs multi-nuvem com várias linguagens de programação e ferramentas disponíveis de fornecedores populares como PuppetLabs, OpsCode, Ansible e HashiCorp.

A maioria dos provedores de nuvem fornece uma poderosa API de controle para seus recursos. Frequentemente, você pode escolher seu conjunto de linguagens e ferramentas favoritas - a Rackspace, por exemplo, suporta SDKs em Ruby, Java, .Net, Node.js, Python e PHP. Não subestime o poder de fazer deploys usando scripts que seu time de desenvolvimento e operações entendam, e fazendo o deploy as 3pm ao invés das 3am, dessa forma o time que criou as mudanças está por perto para suportar o seu lançamento.

Estética

Estética é a medida subjetiva que indica como um usuário responde a um produto. Ela representa uma preferência pessoal. Concordo que a nuvem pouco faz para embelezar o seu site. Apesar disso, ambientes e serviços na nuvem ainda assim são

ferramentas poderosas para refinar a estética da sua aplicação.

Como é muito fácil colocar para rodar um ambiente na nuvem, você pode facilmente criar um para demonstrar uma proposta de alteração visual ou para melhorar a experiência de usuário.

Se você está construindo um website, especialmente um website para dispositivos móveis, um grande desafio é certificar-se de que ele está esteticamente bom, não só no seu laptop, mas também em cada navegador e dispositivo móvel usado pelos seus usuários. Ferramentas de teste baseadas na nuvem tornam fácil o processo de comparar a aparência da aplicação quando executada nos diversos dispositivos. Se você criar um ambiente para homologação, você pode utilizar serviços cloud do tipo amplo acesso (Broad Network Access) como mogotest, Browsershots, Browserstack ou Cross Browser Testing. Você também pode criar seu próprio ambiente de testes na nuvem, utilizando ferramentas como Selenium, Huxley ou dpxdt.

Qualidade percebida

A percepção da qualidade é uma qualidade atribuída a um bem ou serviço baseada em medidas indiretas como “história do produto”. O seu site pode estar livre de problemas agora, mas se você teve um monte de problemas no passado seus usuários ainda podem estar com a sensação de que seu site é de baixa qualidade. Não é apenas o número de bugs que você teve mas também o quanto rápido você os corrigiu.

O uso da nuvem, combinado com Entrega Contínua, torna mais fáceis e rápidas a reprodução, correção e lançamento da correção de bugs. Torna também mais fácil lançar novas funcionalidades e melhorias baseadas em feedback do usuário.

Esse ciclo rápido de correção de bugs e entrega de melhorias é o que dá ao usuário uma percepção de qualidade. Aumentar a percepção de qualidade é a chave para fazer seus usuários sentirem-se como parceiros e promover o seu produto fazendo indicações.

A nuvem e as oito dimensões

Eu acredito plamente que desenvolvendo, testando e fazendo implantação na nuvem você pode construir produtos de melhor qualidade. A capacidade de construir um produto de melhor qualidade em todas as oito dimensões pode não ser exclusividade da nuvem, mas a nuvem ajuda a remover quaisquer barreiras que estejam impedindo você de atingir estas metas.

PROTRACTOR: TESTANDO APLICAÇÕES ANGULARJS COM UMA SOLUÇÃO INTEGRADORA

Daniel Amorim



Se você está desenvolvendo uma aplicação AngularJS, use Protractor para testá-la. Porquê?

- Protractor é um framework de testes funcionais para aplicações AngularJS e funciona como uma solução integradora combinando poderosas ferramentas e tecnologias tais como NodeJS, Selenium, webDriver, Jasmine, Cucumber e Mocha.
- Existem diversas customizações do Selenium para facilitar a criação de testes para aplicações AngularJS
- Protractor tanto acelera quanto evita a necessidade do uso de “sleeps” e “waits” em seus testes tendo estes otimizados internamente.

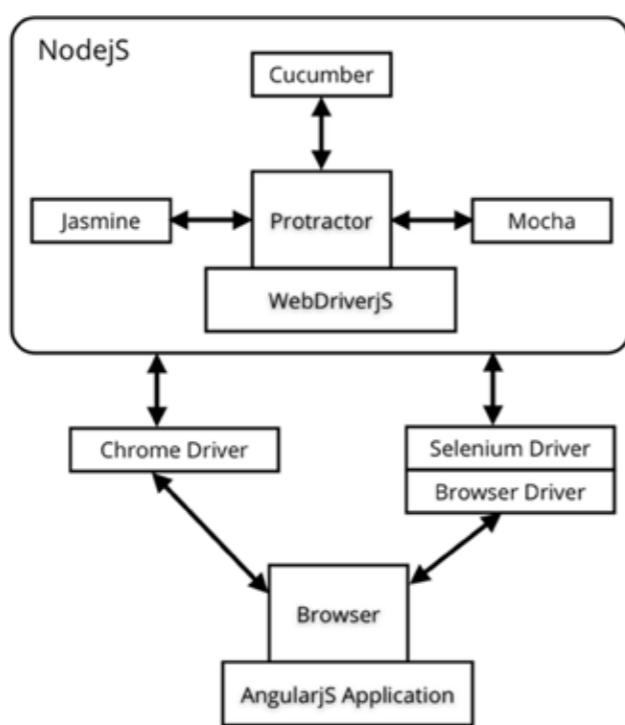
- Como é baseado nos conceitos do AngularJS, é fácil aprender Protractor se você já conhece sobre AngularJS e vice-versa.
- Protractor permite que seus testes sejam organizados baseados no Jasmine, possibilitando que você escreva tanto testes unitários quanto funcionais no Jasmine.
- Executa em navegadores reais e headless.

O que mais você precisa de um framework de testes automatizados?

Entendendo o Protractor mais a fundo

A primeira versão do Protractor foi lançada em julho de 2013, quando o framework era basicamente um protótipo de um framework de testes. Porém o Google, com o apoio da comunidade de testes, está evoluindo o framework para acompanhar a evolução do AngularJS e para satisfazer as necessidades da comunidade que está usando-o.

O projeto do *Protractor* é publico no Github e você pode acompanhar as issues do projeto, adicionar issues que você acha interessantes, comentar nas issues abertas pelos outros, fazer pull requests para colaborar com o projeto, etc. Qualquer um que esteja interessado em colaborar com o crescimento do projeto será bem-vindo!



O Protractor é um framework de automação de testes funcionais, então o intuito dele não é ser a única forma de testar a sua aplicação AngularJS, mas sim cobrir os critérios de aceitação exigidos pelo usuário. Mesmo existindo testes que executem em nível de UI, com o Protractor ainda é necessária a criação de testes unitário e de integração. Esse é um framework para testes funcionais que roda em cima do Selenium, então ele já contém todos os benefícios e facilidades do mesmo além das funcionalidades que ele provê para testar aplicações AngularJS especificamente. Pelo fato do Protractor rodar em cima do Selenium é possível a utilização dos drivers que implementam o WebDriver, tais como ChromeDriver e GhostDriver. No caso do ChromeDriver é possível rodar os testes em cima dele sem a necessidade do Selenium Server. Já para utilizar o GhostDriver é necessária a utilização do PhantomJS, o qual se utiliza do GhostDriver e nesse caso permite rodar os testes em Headless mode.

O framework permite que seja utilizado Jasmine para organizar e criar seus testes e resultados esperados. O Jasmine é compatível com o Protractor pelo fato de que todos os recursos que são extraídos do browser para fazer os testes são promises, e o comando expect do Jasmine trata internamente essas promises e faz parecerem transparentes as validações dos testes.

No início, como o Protractor ainda era recente e ainda estava em fase de maturação, a documentação para usá-lo era restrita e ficava desatualizada rapidamente devido à sua evolução constante. Porém, nos últimos meses a colaboração da comunidade tem crescido bastante e essa documentação tem sido mantida mais atualizada. Isso se deve à facilidade que a comunidade tem de questionar e colaborar através do projeto publico no Github. A única burocracia é que você precisa assinar o contrato digital de colaborador do Google.

PROTRACTOR NA PRÁTICA EM 3 PASSOS

Daniel Amorim



Não é só com teoria que se aprende automação de testes (assim como várias práticas de desenvolvimento). Por isso, nesse artigo vamos utilizar uma abordagem prática de como utilizar *Protractor*. Seguindo os 3 passos abaixo, você já saberá o básico para avançar na criação dos seus testes sozinho.

Passo 1 - Instalando

Um pré-requisito para o uso do Protractor é o Node.js, pois o framework de testes é um pacote node que deve ser instalado através do NPM (Node Package Manager).

Com apenas um simples comando “npm install protractor -g” o Protractor estará instalado em sua máquina e pronto para ser executado através do comando “protractor”. Porém, ao executar esse comando, o protractor vai emitir a mensagem de que exige um parâmetro para a execução do Protractor.

Esse parâmetro exigido pelo Protractor é o caminho do arquivo de configuração do mesmo, pois para executar os testes é necessário ter um arquivo de configuração que vai guiar o Protractor em como ele deve ser executado.

Passo 2 - Configurando

Existem vários parâmetros que permitem a você configurar o Protractor. Abaixo eu descrevo alguns que penso serem os mais importantes.

- **SeleniumAddress:** Permite informar uma URL do Selenium Server que o Protractor usará para executar os testes. Nesse caso o Selenium Server deve estar previamente iniciado antes de rodar os testes no Protractor.
- **SeleniumServerJar:** Permite informar o caminho do arquivo .jar do SeleniumServer. Caso esse parâmetro seja utilizado, o Protractor irá controlar a inicialização e a finalização do Selenium Server, não sendo necessário iniciar o Selenium Server antes de executar o Protractor.
- **SauceUser e SauceKey:** Caso informados o Protractor irá ignorar o parâmetro SeleniumServerJar e irá executar os testes contra um Selenium Server no SauceLabs.
- **Specs:** Um array de arquivos de testes pode ser passado através do parâmetro "specs", o qual o Protractor irá executar. O caminho dos arquivos de teste deve ser relativo ao arquivo de configuração.
- **seleniumArgs:** Permite passar parâmetros para Selenium caso o Protractor initialize-o através do "SeleniumServerJar".
- **capabilities:** Parâmetros também podem ser passados para o WebDriver através do "capabilities", onde é informado o browser contra qual o Protractor vai executar os testes.
- **baseUrl:** Uma URL de acesso padrão pode ser passada para o Protractor através do parâmetro "baseUrl". Com isso toda a chamada feita para um browser será para essa URL especificada.
- **framework:** O framework de testes e de assertions que será utilizado pelo Protractor pode ser determinado pelo parâmetro "framework". Para este existem as seguintes opções: Jasmine, Cucumber e Mocha.

- **allScriptsTimeout:** Para configurar um timeout para cada teste executado, o Protractor provê o parâmetro "allScriptsTimeout", o qual deve receber um valor em milisegundos.

Todos esses parâmetros são encapsulados em um objeto node com nome de config para que o Protractor possa identificar esses parâmetros.

No código abaixo temos um exemplo de arquivo de configuração do Protractor que foi salvo como config.js.

```
exports.config = {  
  seleniumServerJar: './node_modules/protractor/selenium/selenium-server-standalone.jar',  
  specs: [  
    'tests/hello_world.js'  
  ],  
  seleniumArgs: ['-browserTimeout=60'],  
  capabilities: {  
    'browserName': 'chrome'  
  },  
  baseUrl: 'http://localhost:8000',  
  allScriptsTimeout: 30000  
};
```

Nesse exemplo está configurado o parâmetro seleniumServerJar para iniciar o Selenium Server através do Protractor. O arquivo de testes que será executado neste exemplo é o hello_world.js que está dentro da pasta tests. Estes testes serão executados contra o navegador Chrome devido ao parâmetro capabilities estar configurado com o atributo browserName como 'chrome'. O timeout configurado para a execução de cada teste é de 30 segundos.

Após ter o arquivo de configuração pronto, basta ir até a pasta do arquivo e executar o comando protractor config.js e o Protractor irá executar seguindo as intruções passadas no arquivo. Porém, a mensagem abaixo será exibida pelo fato de não termos um arquivo de testes chamado hello_world.js ainda.

```
/usr/local/lib/node_modules/protractor/lib/cli.js:91
    throw new Error('Test file ' + specs[i] + ' did not match any files.');

Error: Test file tests/hello_world.js did not match any files.
    at run (/usr/local/lib/node_modules/protractor/lib/cli.js:91:13)
    at Object. (/usr/local/lib/node_modules/protractor/lib/cli.js:265:1)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
    at Function.Module.runMain (module.js:497:10)
    at startup (node.js:119:16)
    at node.js:901:3
```

Passo 3 - Criando testes

Como já citado nesta [introdução](#), o Protractor roda em cima do Selenium e por isso ele se utiliza de todas as vantagens que o Selenium tem embutidas em si. Porém, o que faz do Protractor o melhor framework de automação de testes para aplicações AngularJS são as facilidades criadas nele especificamente para testar esse tipo de aplicação.

Os comandos customizados pelo Protractor visam capturar os elementos da interface da aplicação através das diretivas do AngularJS. Isso torna o framework interessante pelo fato de que o profissional que aprender a utilizar o Protractor, automaticamente vai aprender sobre o comportamento do AngularJS em relação à renderização de elementos na interface. O inverso também acontece: a partir do momento que se conhece AngularJS, facilmente se aprende a utilizar o Protractor.

O AngularJS utiliza algumas técnicas específicas para manipular o DOM, inserindo ou extraíndo informações do HTML. Exemplos disso são a utilização do ng-model para a entrada de dados, o binding de atributos para exibir dados no DOM e o ng-repeat para a exibição de informações no HTML que estejam contidas em uma lista no javascript.

Para capturar um elemento na interface que contenha um ng-model, por exemplo, o Protractor provê o comando by.model("nome"). Através desse comando o framework retorna o WebElement o qual contém a diretiva ng-model com o valor nome.

```
<div>
  <label>Nome</label>
  <input type="text" ng-model="nome">
  <label>Endereço</label>
  <input type="text" ng-model="endereco">
</div>
```

Dado o exemplo acima, quando executado o comando element(by.model("nome")) o Protractor retorna o WebElement abaixo:

```
<input type="text" ng-model="nome">
```

Para capturar elementos dos quais o AngularJS faz binding de alguma informação, o comando é um pouco diferente, mas a lógica é a mesma do model. O exemplo abaixo demonstra como utilizá-lo.

```
<label>{{nome}}</label>
<label>{{endereco}}</label>
<label>{{CEP}}</label>
</div>
```

Acima temos um código no qual fizemos alguns bindings de informações através do AngularJS. Ao executar o comando element(by.binding("endereco")) pelo Protractor, é retornado o WebElement abaixo:

```
<label>{{endereco}}</label>
```

Na utilização do ng-repeat o Protractor fornece o comando by.repeater("aluno in alunos") o qual retorna um array de elementos.

Este comando permite encadear as chamadas das funções row e column para tornar sua busca mais específica.

Encadeado com o comando row retorna o elemento em que o ng-repeat está, porém com as informações do objeto na posição passada por parâmetro ao comando row. Ex: by.repeater("aluno in alunos").row(0) retorna o primeiro elemento da lista de nomes, pois o javascript inicia a contagem do 0.

Caso queira buscar um elemento específico dentro da estrutura do repeater, é possível utilizar ainda o comando column encadeado com os outros dois anteriores. Neste comando passa-se o nome do atributo no qual está sendo feito binding no HTML, com isso o Protractor retornará o WebElement no qual este binding está sendo feito.

O exemplo abaixo demonstra como funcionam esses comandos.

```
<div ng-repeat="aluno in alunos">
  <span>{{aluno.nome}}</span>
  <span>Nota {{aluno.nota}}</span>
  <input type="text" ng-model="aluno.nota"/>
</div>
```

O código a cima é uma estrutura HTML criada para rodar uma aplicação AngularJS. Dado que a nossa lista de alunos seja um array com os alunos André, Fernando e José, o HTML gerado pelo AngularJS ficaria da seguinte forma.

```
<div ng-repeat="aluno in alunos">
  <span>AndrÃ©</span>
  <span>Nota 8</span>
  <input type="text" ng-model="aluno.nota"/>
</div>
<div ng-repeat="aluno in alunos">
  <span>Fernando</span>
  <span>Nota 9</span>
  <input type="text" ng-model="aluno.nota"/>
</div>
<div ng-repeat="aluno in alunos">
  <span>JosÃ©</span>
  <span>Nota 7</span>
  <input type="text" ng-model="aluno.nota"/>
</div>
```

Dado este exemplo podemos entender como funciona o comando by.repeater do Protractor.

No caso de executado o comando element(by.repeater("aluno in alunos")) o Protractor irá retornar toda a estrutura HTML apresentada acima.

Executando o comando element(by.repeater("aluno in alunos")).row(1) é retornada apenas a estrutura de HTML abaixo:

```
<div ng-repeat="aluno in alunos">
  <span>Fernando</span>
  <span>Nota 9</span>
  <input type="text" ng-model="aluno.nota"/>
</div>
```

E no caso mais específico para buscar um elemento final, executando o comando element(by.repeater("aluno in alunos").row(1).column("nota")) o Protractor retorna apenas a tag span abaixo:

```
<span>Nota 9</span>
```

Agora que já sabemos como usar alguns dos comandos do Protractor, vamos criar alguns testes para demonstrar na prática o uso dos mesmos. Vamos utilizar o exemplos de site do AngularJS (<http://angularjs.org/>) que apresento abaixo.

```
<html ng-app>
  <head>
  </head>
  <body>
    <div>
      <label>Name:</label>
      <input type="text" ng-model="yourName" placeholder="Enter a name here">
      <hr>
      <h1>Hello {{yourName}}!</h1>
    </div>
  </body>
</html>
```

Para essa aplicação podemos criar dois testes como exemplo:

- No primeiro vamos verificar se o texto inicial da tag h1 é "Hello!", pois não temos nenhum valor no atributo "yourName" ainda.
- Em seguida usaremos "Protractor" como entrada de dados para representar o atributo "yourName" e verificar novamente se o texto dentro da tag h1 contém o valor "Hello Protractor!". O comando sendKeys("Protractor") é um método do WebElement para digitar o valor dentro do campo texto.

```
describe('Hello World form', function() {
  it('should display Hello!', function() {
    browser.get('');
    expect(element(by.binding('yourName')).getText()).toEqual("Hello !");
  });

  it('should be able to insert text',function(){
    browser.get('');
    element(by.model('yourName')).sendKeys("Protractor");
    expect(element(by.binding('yourName')).getText()).toEqual("Hello Protractor!");
  });
});
```

Colocando esses testes dentro do arquivo hello_world.js conforme configuramos no capítulo anterior, fará desta a sua atual suite de testes.

O que você precisa fazer agora é rodar novamente o Protractor e agora não será mais uma mensagem de erro que será exibida, mas o status de cada um dos testes executados.

```
Finished in 2.663 seconds
2 tests, 2 assertions, 0 failures
```

Voilà! Seus testes estão funcionando. Você pode incrementar a sua suíte de testes quando for preciso ou ainda melhor, integrá-la a uma ferramenta de integração contínua para mantê-la rodando constantemente.

GATLING: UMA FERRAMENTA DE TESTES DE PERFORMANCE

Rodrigo Toledo



Gatling é uma poderosa ferramenta *open-source* para Teste de Performance, lançada em dezembro de 2011. Ela foi mencionada no *ThoughtWorks Technology Radar* de 2013 e 2014 como uma ferramenta que vale a pena conhecer. Gatling é uma DSL (*domain-specific language*) leve, escrita em *Scala* que vem com uma premissa interessante de sempre tratar seus testes de performance como código de produção.

Um outro ponto interessante sobre o Gatling é que você pode definir e escrever seus cenários de teste de performance da

mesma forma como você está acostumado a fazê-los com outras ferramentas de automação. Assim, você pode escrever código de performance de forma legível e de fácil manutenção, que pode ser manipulado por qualquer sistema de controle de versão.

Gatling integra facilmente com Jenkins através do *jenkins-plugin* e também pode ser integrado com outras ferramentas de integração contínua. Isso é ótimo porque você pode obter feedback constante de seus testes de performance. Outra vantagem é que você pode facilmente executar os testes através do Maven e do Gradle com a

ajuda do *maven-plugin* e do *gradle-plugin*. Gatling também fornece relatórios elegantes e com informações significativas sobre o funcionamento da sua aplicação. A última versão estável (2.1.7) suporta Linux, OS X e Windows.

Portanto, agora que sabemos um pouco mais sobre Gatling, que tal entender melhor tudo isso?

Basicamente, a estrutura do Gatling pode ser definida em três partes:

1. Configuração do protocolo HTTP - Define a URL base que você vai executar seus testes. Além disso, você pode definir outras configurações, como user agent, language header, conexão e assim por diante.
2. Definição de Cenário - É o núcleo do seu teste! Um cenário é um conjunto de ações (GET, POST, etc) que serão executadas, a fim de simular uma interação do usuário com o aplicativo.
3. Definição de Simulação - Define a carga (quantidade de usuários) que irá executar simultaneamente seu cenário por um período de tempo.

Vamos imaginar uma aplicação web simples onde os usuários podem cadastrar modelos de computadores. Essa aplicação fornece feedback para o usuário logo depois que o modelo de computador é inserido corretamente na base de dados.

Em primeiro lugar, vamos adicionar algumas informações básicas e começar definindo a nossa configuração HTTP. Montamos nossa *base URL* para <http://computer-database.gatling.io>. Isto significa que cada vez que executar um request, o Gatling vai utilizar nossa *base URL*.

```
import scala.concurrent.duration._

import io.gatling.core.Predef._
import io.gatling.http.Predef._

class FirstSimulation extends Simulation {

    val httpProtocol = http
        .baseUrl("http://computer-database.gatling.io")
```

Depois disso, vamos definir um cenário chamado "*First Simulation*". Este cenário irá executar um conjunto de ações. A primeira ação será um GET para página inicial. A segunda ação novamente será um GET, dessa vez, acessando o *form* de cadastro de computadores. Note que estamos utilizando um comando interessante aqui: *check()*. Basicamente o que o comando faz é: após enviarmos nossa requisição, usamos *check()* para verificar se a requisição foi efetuada com sucesso.

```
val scn = scenario("First Simulation")
    .exec(http("request_0")
        .get("/"))
    .exec(http("request_1")
        .get("/computers/new")
        .check(status.is(200)))
```

Nossa terceira ação é um POST. Como você pode ver logo abaixo, agora estamos inserindo informações no *form* de cadastro de computadores. Note que estamos realizando essa operação utilizando identificadores únicos para cada elemento dentro do *form*. Além disso, utilizamos *check()* novamente para verificar a nossa resposta. Nesse caso, a resposta deverá conter erros, pois a informação fornecida possui problemas.

```
.exec(http("request_2")
    .post("/computers")
    .formParam("name", "Testing Computer")
    .formParam("introduced", "2015-17-07")
    .formParam("discontinued", "2015-31-12")
    .formParam("company", "6")
    .check(status.is(400)))
```

Na última ação do nosso fluxo, estamos fazendo mais um POST. Dessa vez, iremos fornecer informações válidas para completar o nosso form. A resposta do servidor deverá ser então positiva.

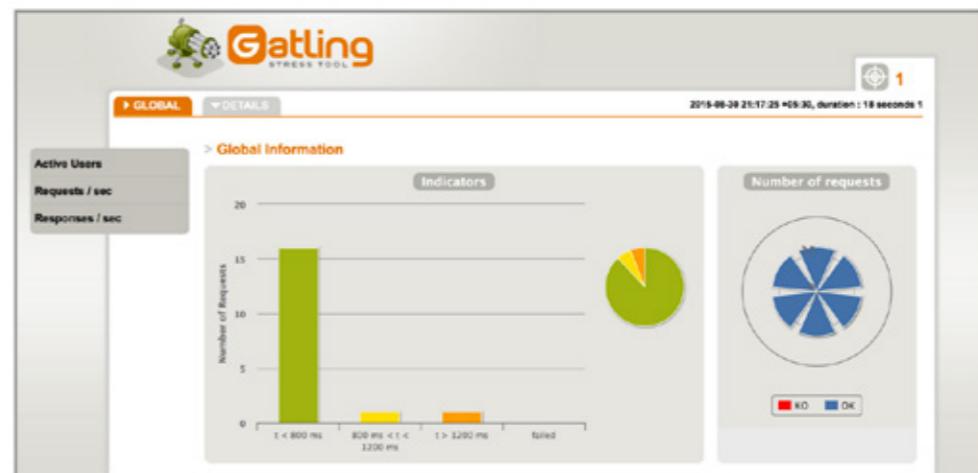
```
.exec(http("request_3")
    .post("/computers")
    .formParam("name", "Testing Computer")
    .formParam("introduced", "2015-07-17")
    .formParam("discontinued", "2015-12-31")
    .formParam("company", "6")
    .check(status.is(200)))
```

Ok, agora temos o nosso fluxo definido: acessamos a nossa *homepage*, tentamos inserir um modelo de computador utilizando dados inválidos e, por fim, utilizamos dados válidos para cadastrar um novo modelo de computador. Mas, o que queremos testar exatamente aqui? Isso é o que nós precisamos dizer ao Gatling através de *setUp()*. Em nosso *setUp()* estamos dizendo ao Gatling: "Ei, por favor simule o meu cenário para 3 usuários". Basicamente, significa que os nossos "usuários" vão começar a interagir com a nossa aplicação ao mesmo tempo.

```
setUp(scn.inject(atOnceUsers(3))).protocols(httpProtocol)
```

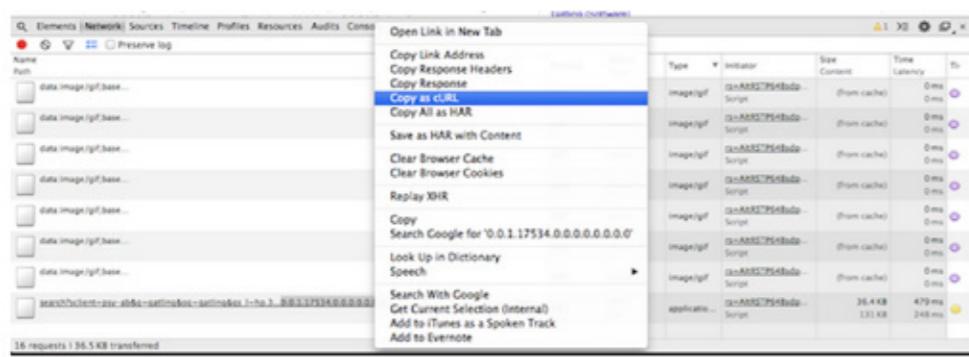
O Gatling pode ser utilizado também para explorar e entender melhor respostas HTTP. Você pode utilizar *Gatling Assertions* para verificar se as suas solicitações (*requests*) retornaram um código 200 ou qualquer resultado esperado, como um valor JSON específico, por exemplo. Gatling permite usar expressões regulares, CSS Selectors, XPath e JSONPath para analisar a sua resposta, extrair a informação que te interessa e até utiliza-la em seus próximos passos!

Para que você não fique com aquela curiosidade sobre como é a página de relatórios do Gatling, aqui vai um exemplo (claro que existem muitos outros modos de visualização):



Vale a pena mencionar que o Chrome Dev Tools pode ajudar na hora de identificar requisições HTTP, mensagens ou seja o que for que você deseja simular. A guia *Network* é extremamente útil nesse caso. Eu também recomendo que você utilize a ferramenta cURL como ferramenta de apoio. Ela pode ajudar a simular facilmente seus pedidos antes de mapeá-los para dentro de seus cenários de teste no Gatling.

Dica extra: No *Chrome Dev Tools*, na guia de Network, você pode clicar com o botão direito sobre a solicitação/request do seu interesse e selecionar a opção "Copy as cURL" ;)



Gatling também fornece uma interface gráfica que permite que você grave seus cenários. Entretanto, o uso dessa interface não é algo necessário devido à simplicidade da DSL.

Então, agora é hora de botar a mão na massa e começar a tratar os seus testes de performance como código de produção!

ALTERNATIVAS AO TESTFLIGHT PARA ANDROID

Rafael Garcia



Como você deve ter ouvido falar, o *Testflight está dando fim ao seu suporte para Android* em 21 de março de 2014, mas não se preocupe, pois mostraremos algumas alternativas ao Testflight para que você possa escolher a ferramenta mais adequada às suas necessidades.

Primeiramente, definiremos alguns critérios para analisar:

- Implantação contínua.
- Possui versão mobile para instalar versões.

- Permissão individual para testadores.
- Permissão via lista de distribuição.
- Custo.

Com os critérios estabelecidos, vamos primeiro olhar o Testflight para ter um ponto de partida e então olharemos outras 5 ferramentas:

- HockeyApp
- Appaloosa

- TestFairy
- Play Store
- Apphance

Testflight (<https://testflightapp.com/>)

Implantação contínua: Upload de novas versões via API e [plugin para Jenkins](#).

```
curl http://testflightapp.com/api/builds.json
  -F file=@testflightapp.ipa
  -F dsym=@testflightapp.app.dSYM.zip
  -F api_token='your_api_token'
  -F team_token='your_team_token'
  -F notes='This build was uploaded via the upload API'
  -F notify=True
  -F distribution_lists='Internal, QA'
```

Versão mobile:

Permissão individual para testadores: Selecione os usuários para os quais quer dar permissão e voilà.

Permissão via lista de distribuição: Crie a lista selecionando os usuários. Digite o nome da lista no parâmetro distribution_list da API.

The screenshot shows the 'Distribution List' creation interface. It has a 'List Name' field (empty), a 'Teammates' section where 'Rafael Garcia' is selected, and a preview area showing the list name and the user selected.

Custo: Free

HockeyApp (<http://hockeyapp.net/>)

É a versão hospedada do projeto open source *HockeyKit*. Um pouco confuso de entender o fluxo de uso no começo, mas tem um bom custo-benefício no quesito funcionalidades.

The screenshot shows the HockeyApp dashboard for the 'example' app. It displays basic information like the app ID and developer details, along with a table of the latest versions and their metrics.

Implantação contínua: Upload de novas versões *via API e plugin para Jenkins*.

```
curl \
  -F "status=2" \
  -F "notify=1" \
  -F "notes='Some new features and fixed bugs.'" \
  -F "notes_type=0" \
  -F "ipa=@hockeyapp.ipa" \
  -F "dsym=@hockeyapp.dSYM.zip" \
-H "X-HockeyAppToken: 4567abcd8901ef234567890abcdef1234567890abcdef/app_versions/upload"
https://rink.hockeyapp.net/api/2/apps/1234567890abcdef1234567890abcdef/app_versions/upload
```

Versão mobile:

The screenshot shows the mobile interface of HockeyApp. On the left is the 'Dashboard' showing two Android apps ('example' and 'My Application 2'). On the right is the 'App' details page for 'example', showing the developer info, download button, and version history.

Permissão individual para testadores: Não possui.

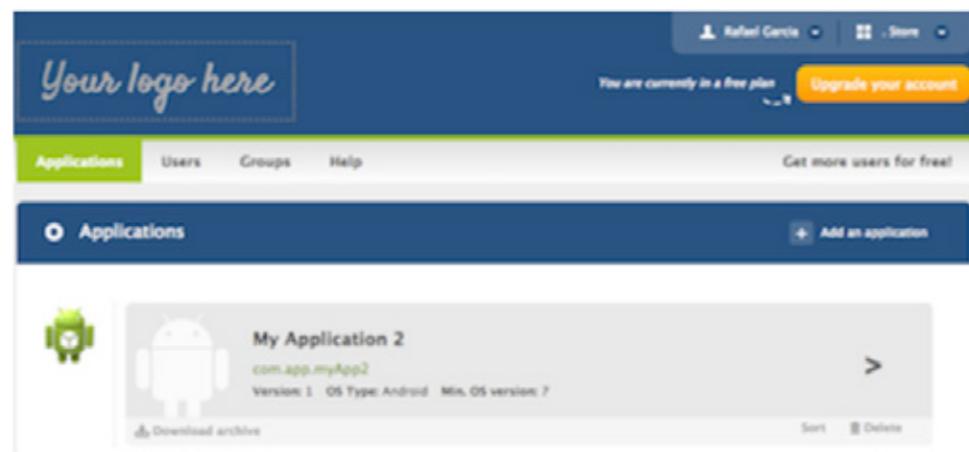
Permissão via lista de distribuição: Crie tags e selecione os usuários. Digite o nome da tag no parâmetro tags da API.

The screenshot shows the 'Enter Tag' interface. It has a 'Tag' input field and a 'Select Users' section where 'Rafael Garcia' and 'Rafa' are listed with their email and roles.

Custo: Primeiro mês é free e o plano é o Básico. Confira os planos [aqui](#).

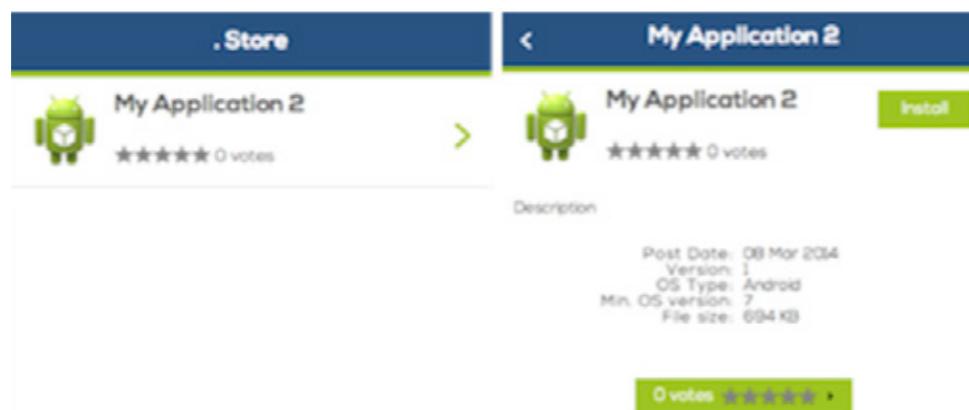
Appaloosa (<http://www.appaloosa-store.com/>)

A usabilidade não é tão legal, porém é bem simples de começar a usar.



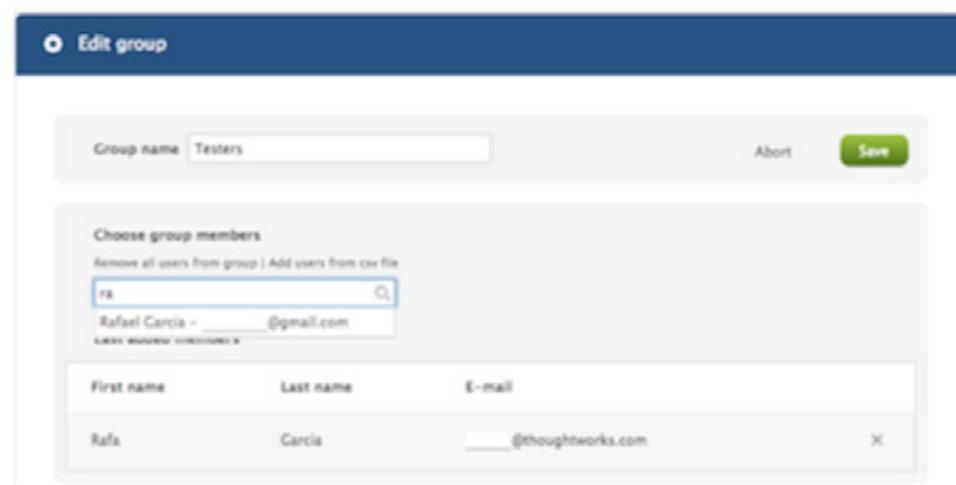
Implantação contínua: *Plugin para o Jenkins*

Versão mobile:



Permissão individual para testadores: Não possui.

Permissão via lista de distribuição: Crie grupos, ligue os usuários ao grupo e informe na app quais grupos tem permissão.



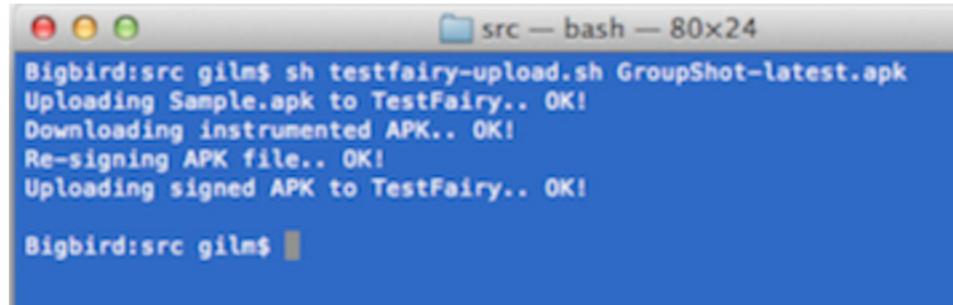
Custo: Free, com limite de 1 app e 10 usuários. Confira os planos [aqui](#).

TestFairy (<http://testfairy.com/>)

Boa interface, simples de usar, *vários relatórios*, dá *suporte às apps com SDK do Testflight*. Peca por não ter uma versão mobile, mas, segundo o suporte do site, estão trabalhando nisso e pretendem lançar uma app nativa em pouco tempo, então fique de olho.



Implantação contínua: Upload de novas versões via API, além de plugin para Gradle.



```
src — bash — 80x24
Bigbird:src gilm$ sh testfairy-upload.sh GroupShot-latest.apk
Uploading Sample.apk to TestFairy.. OK!
Downloading instrumented APK.. OK!
Re-signing APK file.. OK!
Uploading signed APK to TestFairy.. OK!

Bigbird:src gilm$
```

Versão mobile: Ainda não possui.

Permissão individual para testadores: Selecione os usuários que quer convidar para testar sua app



	Email	Status	Installed Version	
	testers			Manage Groups
	_____@thoughtworks.com	Invited		View
	E-mail			

Permissão via lista de distribuição

Crie grupos para os usuários e informe o nome do grupo no parâmetro TESTER_GROUPS no script de deploy. Será enviado um email para os usuários do grupo com o link para download do apk.



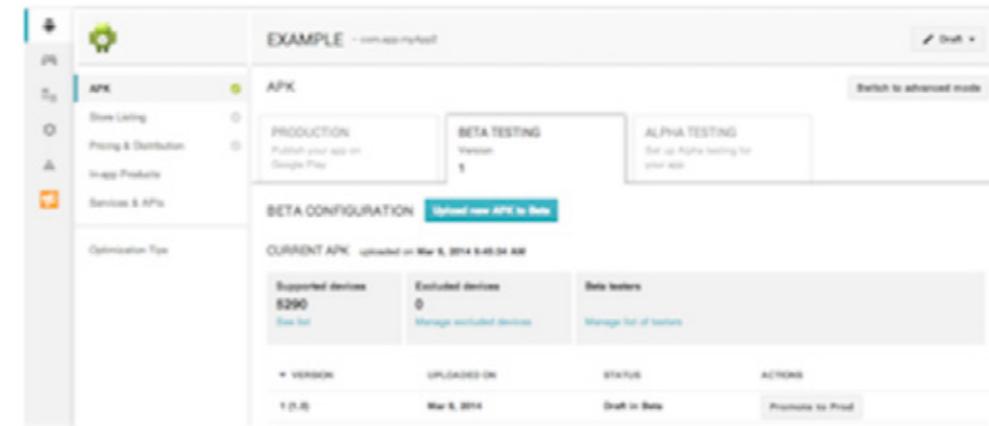
	Testers	Groups	Devices	
	_____@thoughtworks.com	x-testers		View
	E-mail	Add tester		

```
1 #!/bin/sh
2
3 # UPDATE THESE
4 API_KEY=
5 TESTER_GROUPS= testers
6 KEYSTORE=~/.android-studio-keystore/keystore.jks
7 STOREPASS=PUT_YOUR_KEYSTORE_PASSWORD_HERE
8 ALIAS=android
9
```

Custo: Free.

Play Store (<http://developer.android.com/>)

O Google oferece o Developer Console para fazer Beta testing. Um tanto burocrático, é preciso informar várias configurações da app antes de começar a usar de fato o serviço, além disso não gostei de precisa criar grupos de testadores fora da interface do Developer Console.



The screenshot shows the Google Play Developer Console interface for managing an APK. On the left, there's a sidebar with navigation links like 'APK', 'Store Listing', 'Pricing & Distribution', 'In-app Products', 'Services & APIs', and 'Optimization Tools'. The main area is titled 'EXAMPLE - com.example.app' and shows the 'APK' section. It displays three tabs: 'PRODUCTION' (with a button to 'Publish your app on Google Play'), 'BETA TESTING' (with a tab for 'Version 1'), and 'ALPHA TESTING' (with a button to 'Set up Alpha testing for your app'). Below these tabs, there's a 'BETA CONFIGURATION' section with a 'Upload new APK to Beta' button. Further down, it shows 'CURRENT APK' uploaded on 'Mar 8, 2014 9:45:34 AM'. There are sections for 'Supported devices' (5290), 'Excluded devices' (0), and 'Beta testers' (Manage list of testers). At the bottom, there's a table with columns 'VERSION', 'UPLOADED ON', 'STATUS', and 'ACTIONS'.

Implantação contínua: Não possui.

Permissão individual para testadores: Não possui.

Permissão via lista de distribuição: Crie um grupo no Google groups ou uma nova comunidade no Google+.

WHO CAN BETA TEST YOUR APP?

You can add Google Groups or Google+ Communities to be eligible to test your app. Once you have added a group, you need to send your testers the opt-in link below. Once they opt-in they will receive this version through Google Play.

Add Google Groups or Google+ Communities

<https://plus.google.com/communities/109345128321814480386>

Add

Testers opting in

Your app can only be used by your testers when it is published to Google Play. If your app has no production APK then it will only be visible to Alpha and Beta testers. The link that your testers can use to opt-in will be displayed here when you have published your app.

[Close](#)

Custo: É cobrada uma taxa de registro. Veja [como criar seu Google Play Developer Console](#).

Apphance (<http://www.utest.com/apphance>)

O mais simples de todos em funcionalidades e bem fácil de usar.



Implantação contínua: Possuem, ferramenta de automação de builds para projetos mobile.

Versão mobile: Não possui.

Permissão individual para testadores: Selecione os usuários que receberão link de download da app via email.



Permissão via lista de distribuição: Não possui.

Custo: Free com limite de 1 app e 50 dispositivos. Para saber valores é preciso contactá-los.

Aqui está uma tabela comparativa dos serviços:

	TESTFLIGHT	HOCKEY APP	APPALOOSA	TEST FAIRY	PLAY STORE	APPHANCE
<i>Implantação contínua</i>	X	X	X	X		X
<i>Versão mobile</i>	X	X	X		X	
<i>Permissão individual para testadores</i>	X			X		X
<i>Permissão via lista de distribuição</i>	X	X	X	X	X	
Custo	Free	1 mês free	Free, com limite de 1 app e 10 usuários	Free	Cobra taxa de registro	Free com limite de 1 app e 50 dispositivos

ENTRE EM CONTATO

thoughtworks.com



ThoughtWorks®

BELO HORIZONTE | PORTO ALEGRE | RECIFE | SÃO PAULO

