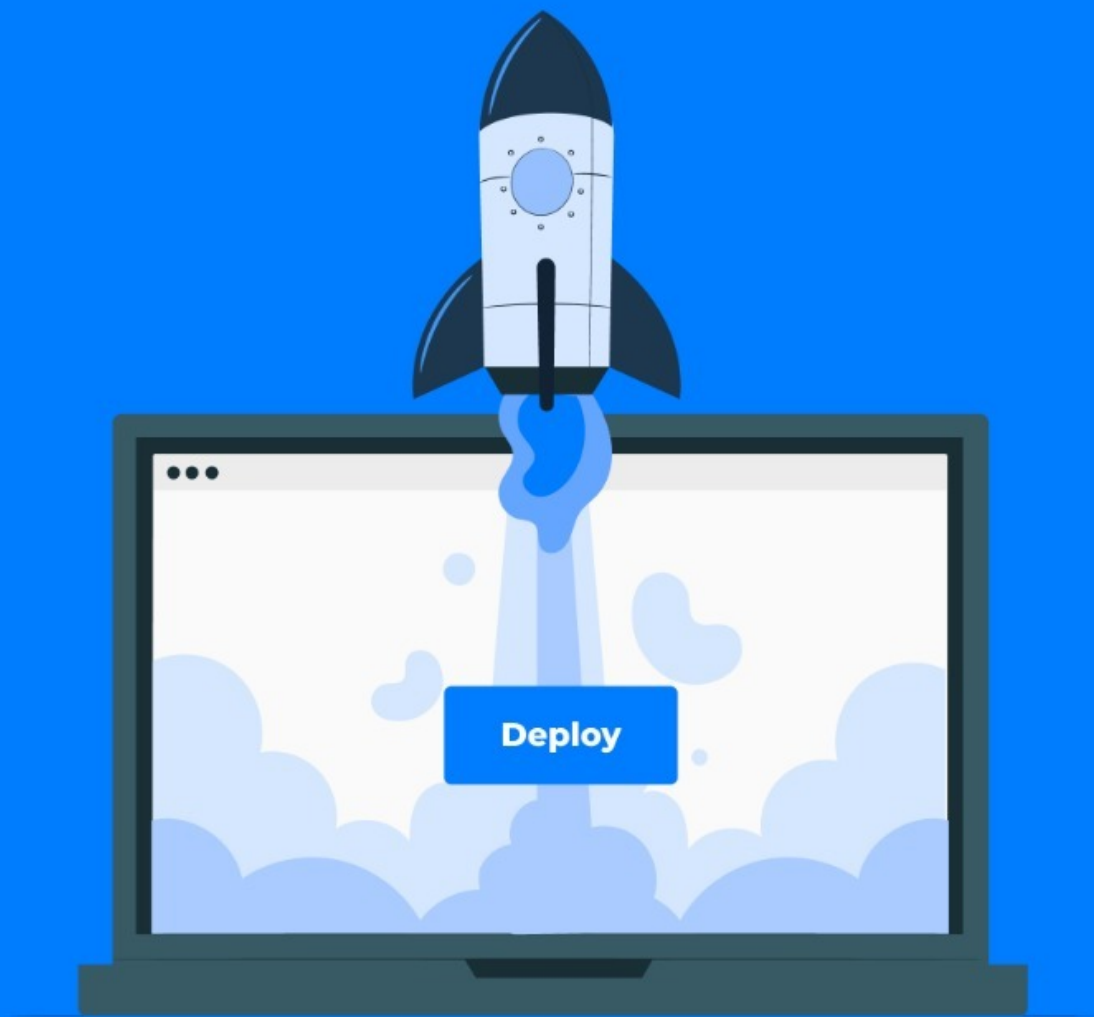


Deploy em Produção para Desenvolvedores



Rafael Gomes

Organizador

Deploy em produção para desenvolvedores

Rafael Gomes

Esse livro está à venda em <http://leanpub.com/deployemprodparadevs>

Essa versão foi publicada em 2021-10-11



Esse é um livro [Leanpub](#). A Leanpub dá poderes aos autores e editores a partir do processo de Publicação Lean. [Publicação Lean](#) é a ação de publicar um ebook em desenvolvimento com ferramentas leves e muitas iterações para conseguir feedbacks dos leitores, pivotar até que você tenha o livro ideal e então conseguir tração.



This work is licensed under a [Creative Commons Attribution 4.0 International License](#)

Conteúdo

Agradecimentos do organizador	1
Colaboradoras	3
Prefácio	4
Introdução	5
Como esse livro está sendo construído?	6
Como ler esse livro	7
Esse livro está “pronto”?	7
Como será dividido esse livro?	7
O que você precisa saber pra ler esse livro?	8
O que é deploy? (Rafael Gomes)	9
Introdução	9
Como funciona o deploy de um produto de software	9
Destinos possíveis de um deploy	9
Na prática, como funciona o deploy?	11
O que é pipeline	12
Introdução	12
Como inicia um pipeline?	12
Etapas do pipeline de software	13
Separação por “job”	13
Onde é executada essa “step”?	14
Quebrando o pipeline	15
O que é Pull Request? (Rafael Gomes)	16
Introdução	16
Como usar Pull Request para o processo de revisão?	17
Como revisar o Pull Request?	20
Recebi uma lista imensa de coisas a corrigir no meu PR, fico triste?	23
Quantas pessoas devem revisar meu código?	24
Conclusão	24

CONTEÚDO

Estratégia de Testes - Construindo Confiança (Samanta Cicilia)	25
Introdução	25
Por que escrever Testes	25
Definindo Testes	26
Desafios	42
Testes Contínuos	43
Como usar a estratégia de testes para tomada de decisão	44
Conclusão	45
Migrações em Banco de Dados Relacionais (Daniane Pereira Gomes)	46
Introdução	46
Boas Práticas em Migrações	48
Conclusão	54
O que deve ter no seu pipeline? (Rafael Gomes)	55
Garantindo a qualidade para sua Infra como código (Rafael Gomes)	57
Introdução	57
O que seria QA para IaC?	57
Testando seu código antes do commit	58
Testes automatizados para Ansible	58
Molecule	60
Conclusão	64
Deploy de Infraestrutura como código (Rafael Gomes)	65
Introdução	65
Exemplo	65
Deploy de infra é sobre infra	66
Tipos de deploy de IaC	66
Conclusão	68
O livro ainda não está finalizado	70

Agradecimentos do organizador

Eu começo agradecendo a pessoa que me deu a chance de estar aqui e poder iniciar esse projeto: minha mãe. A famosa Cigana, ou Dona Arlete, pessoa maravilhosa, que pra mim é um exemplo de ser humano. Passar tudo que passou, com esse sorriso no rosto.

Quero agradecer também a minha segunda mãe, Dona Maria, que tanto cuidou de mim quando eu era criança, enquanto Dona Arlete tomava conta dos outros dois filhos e um sobrinho. Me sinto sortudo por ter duas, enquanto muitos não tem ao menos uma mãe.

Quero agradecer muito minha companheira, Ana Carla, que sempre me deu todo apoio e revisou boa parte dos textos que tem aqui.

Obrigado [Somatório](#)¹ que revisa praticamente **tudo** que eu escrevo. Ele tem sido além de tudo um grande amigo. Um dos melhores presentes que tive do Rio Grande do Sul.

Obrigado a [Daniel Barba](#)² por ter feito essa capa sensacional. Anotem esse nome, pois ele será sensação como desenvolvedor logo.

Obrigado a [Samanta Cicilia](#)³ por ter sido a primeira autora, além de mim, a escrever um capítulo para esse projeto.

Obrigado a [Daniane Pereira Gomes](#)⁴ por ter escrito o capítulo “Migrações em Banco de Dados Relacionais”.

Obrigado a [Guto Carvalho](#)⁵ por ter escrito o prefácio desse livro. Você sempre foi e sempre será uma grande referência nessa área pra mim. Se não fosse o seu material lá no começo, nada disso seria possível.

Obrigado especial a um grupo de pessoas que faz parte de um canal que tenho no telegram para avaliar todo material que eu crio para o livro antes dele sair:

- [Braier Alves](#)⁶
- [Morvana Bonin](#)⁷
- [Giu](#)⁸
- [Victor Martinez](#)⁹

¹<https://twitter.com/somatorio>

²<https://twitter.com/b4rba88>

³<https://twitter.com/samantacicilia>

⁴<https://twitter.com/danianepg>

⁵<https://twitter.com/gutocarvalho>

⁶<https://github.com/braieralves>

⁷<https://twitter.com/morvanabonin>

⁸<https://twitter.com/ReginaSauro>

⁹<https://twitter.com/vcrrmartinez>

Eu preciso agradecer também a [Gleydson Silva](https://twitter.com/gleydsonmazioli)¹⁰ que é o responsável pelo projeto [Guia Foca](https://guiafoca.org/)¹¹ que além de ter sido o lugar onde eu aprendi GNU/Linux, foi também uma grande inspiração para abrir o conhecimento para todas as pessoas.

Não posso deixar de agradecer a [Aaron Swartz](https://pt.wikipedia.org/wiki/Aaron_Swartz)¹², que sempre foi minha inspiração para manter o conhecimento aberto. Que a sua chama se mantenha acesa para sempre.

¹⁰<https://twitter.com/gleydsonmazioli>

¹¹<https://guiafoca.org/>

¹²https://pt.wikipedia.org/wiki/Aaron_Swartz

Colaboradoras

Aqui está a lista de pessoas que colaboraram com o livro com Pull Request (PR). Meu muito obrigado por corrigir, adicionar e assim melhorar o material para toda comunidade:

- [Adail Horst](#)¹³
- [Vinícius Mamoré](#)¹⁴
- [Lays Rodrigues](#)¹⁵
- [Evellyn Lima](#)¹⁶
- [Edson Ferreira](#)¹⁷
- [Kelvin Salton](#)¹⁸
- [Paulo Gonçalves](#)¹⁹

Nesse [link](#)²⁰ você pode acompanhar a colaboração de todas as pessoas, sejam autoras ou colaboradoras.

Meu muito obrigado por corrigir, adicionar e assim melhorar o material para toda comunidade.

¹³<https://github.com/SpawW>

¹⁴<https://github.com/vmamore>

¹⁵<https://github.com/lays147>

¹⁶<https://github.com/evelew>

¹⁷<https://github.com/edsoncelio>

¹⁸<https://github.com/kelvins>

¹⁹<https://github.com/PauloGoncalvesBH>

²⁰<https://github.com/gomex/deploy-em-producao/graphs/contributors>

Prefácio

Na atualidade, podemos dizer que todas as empresas são empresas de tecnologia, em especial aquelas que precisam ter presença na web ou em dispositivos móveis. Neste segmento, a entrega de software é um processo estratégico. Aqueles que conseguem entender as necessidades e processar o feedback de seus usuários – internos ou externos – com assertividade, podem empacotar isso e entregar em tempo hábil de atender sua demanda, mantendo sua base ou até mesmo expandindo esta. Até mesmo empresas em segmentos não ligados diretamente a tecnologia, dependem deste mesmo processo de entrega, seja para operação, gestão ou administração de diversos pontos de seu negócio.

Entregar software hoje se tornou algo imprescindível, e quem entrega mais rápido, com menos erros e falhas, com maior estabilidade, e principalmente controle, normalmente tem uma vantagem competitiva dentro de seu segmento.

A entrega pode estar relacionada a uma correção, uma atualização, a adição de um novo recurso, enfim, existem vários motivos para entregarmos software e várias formas para se fazê-lo.

Este livro disserta sobre este assunto e percorre um caminho que permitirá ao leitor entender os conceitos e princípios da entrega de software, seus benefícios, as formas diversas de realizar tal processo, as boas práticas envolvidas, permitindo ao leitor absorver as experiências aqui compartilhadas, a fim de aplicar e incorporar tais práticas em seu cotidiano de trabalho.

Neste livro Rafael Gomes toma o cuidado de abordar o assunto do ponto de vista de quem ainda não tem muita intimidade com o assunto, utilizando uma linguagem simples, abordando conceitos de forma natural e com exemplos reais, permitindo então uma leitura fluída e de fácil compreensão.

Rafael – @gomex – é um profissional com uma carreira sólida, calcada no compartilhamento de conhecimentos, nos valores universais de vida em comunidade e do código aberto. Há anos lidera grupos open source ligados ao assunto de containers, entrega de software, DevOps e Infra as Code. Escreveu um dos livros referência em Docker no Brasil e continua sua jornada de compartilhar conhecimento neste novo projeto.

Recomendo a leitura, o autor, o conteúdo e referendo seu domínio em todos os assuntos aqui abordados.

Aproveite a leitura :)

Abraços,
Guto Carvalho

Introdução

Ao longo de alguns anos de experiência tenho percebido que muitas pessoas tem dúvidas sobre quais os elementos que podem ser usados para entregar o produto de software em produção.

São inúmeras as possibilidades, tanto para fazer errado, como para realizar corretamente esse processo. O objetivo aqui não é mostrar a melhor possibilidade, mas sim apresentar um caminho, que já foi utilizado por algumas pessoas e normalmente funciona para a maioria dos casos.

O objetivo é mostrar como levar seu código desde sua máquina até os confins mais distantes do então famoso ambiente de produção.

Há muito mistério sobre como trilhar esse caminho, mas há bastante mal entendido também, o que torna a tarefa de uma pessoa iniciante mais difícil.

Uma das perguntas mais famosas feitas por iniciantes é: “Qual ferramenta usar?”. Se você começou a ler esse livro esperando a resposta nas primeiras páginas, infelizmente precisará esperar um pouco, pois aqui serão tratados primeiro os conceitos, lhe dando elementos para que você possa entender os reais motivos de usar a ferramenta A ou B.

A escolha da ferramenta não é o primeiro passo. Imagine você trabalhando no ramo de decoração e precisa pendurar coisas na parede. Se você não entender inicialmente o que deseja colocar na parede, ou seja, o seu formato, peso e tamanho, como conseguirá escolher entre o martelo para colocar o prego ou a furadeira para colocar o parafuso? A situação para colocar um software em produção segue a mesma lógica, isso quer dizer que nesse livro lhe mostraremos os pesos, formatos e tamanhos que você deve levar em conta antes de colocar esse produto em sua parede chamada produção.

A ideia desse livro é oferecer alguns pontos de vista para ajudar você nessa empreitada. Não existe pretensão alguma aqui de fundar nenhum padrão ou ideia nova. O que é apresentado nessas páginas é nada mais do que a soma de experiências de várias pessoas, então não há intenção alguma de tomar todo crédito, afinal toda construção de um novo conteúdo normalmente é 10% experiência própria e 90% de aprendizado prévio.

Apresentaremos nesse livro alguns códigos para servir como exemplo ao que está sendo explicado no capítulo. Se você ainda não conseguir ler o código e entender, não se preocupe, pois a proposta é que o texto seja o suficiente. O código seria apenas um esforço extra pra demonstrar para aquelas pessoas que conseguem ler o código.

Resumindo, a ideia é que esse livro lhe ajude a entender esse assunto, seja você uma pessoa que já sabe sobre o assunto e queira apenas reforçar o que sabe com base em outros pontos de vistas ou se você é uma pessoa iniciante no assunto e quer de fato entender a partir dos fundamentos.

Como esse livro está sendo construído?

Ele é feito de forma colaborativa a partir [desse repositório](https://github.com/gomex/deploy-em-producao)²¹, escrito em markdown e lançado como ebook no [Leanpub](https://leanpub.com)²².

A linha editorial é feita por Rafael Gomes, também conhecido como Gomex, ou seja ele recebe a colaboração de um monte de pessoas e organiza os conteúdos para que o produto final não seja uma “colcha de retalhos”.

Rafael Gomes também aparecerá como autor desse livro, mas seu papel principal aqui será de organizador, que é o título que ele receberá na capa.

Você irá perceber que cada capítulo tem ao seu lado o nome da pessoa que foi a principal autora do mesmo, ou seja, se uma pessoa colaborou com poucas palavras ou pequenas correções no capítulo o nome dessa pessoa aparecerá na página de colaboradores ao final do livro.

Falando em colaboração, se você quiser escrever algo que ainda não está no livro ou apenas corrigir algo que está errado, por favor, submeta um Pull Request(PR). Se quiser apenas reportar um erro, abra um [issue](#)²³. Colabore de alguma forma e todas as pessoas que leem o livro se beneficiarão da sua contribuição.

²¹<https://github.com/gomex/deploy-em-producao>

²²<https://leanpub.com>

²³<https://github.com/gomex/deploy-em-producao/issues>

Como ler esse livro

Ele pode ser lido como a maioria dos livros, que é começando da primeira página até a última, mas você pode pular sem culpa alguns capítulos ou até mesmo ir diretamente no conteúdo que desejar. Ele foi construído de uma forma que os conceitos são apresentados e desenvolvidos a medida que são necessários, isso quer dizer que se você pulou alguns capítulos e encontrou alguma dificuldade para entender, talvez retornar um pouco e ler capítulos anteriores possa auxiliar no entendimento.

O aconselhável é que você leia o livro da forma convencional, pois terá a oportunidade de ver outro ponto de vista sobre os conceitos que você já conhece e assim reforçar esse conhecimento.

Esse livro está “pronto”?

A ideia desse livro é que ele nunca esteja pronto. Isso pode assustar aquelas pessoas que esperam o “final” de uma história, mas aqui a abordagem é diferente.

A ideia do livro é ser um local onde se possa congrega conhecimentos sobre como fazer deploy em produção a medida que o entendimento e experiências de várias pessoas mudem sobre o assunto, ou seja, sempre terá uma coisa nova para ser falar sobre entrega em produção, correto?

Como será dividido esse livro?

A ideia é que ele aborde as seguintes coisas:

1. Conceitos básicos sobre entrega de produtos em produção (Ex. O que é Pipeline, como funciona, o que tem que ter nele e afins). A ideia aqui é falar sobre as melhores práticas de cada etapa da entrega também (ex. testes de integração, migração de banco de dados e etc).
2. Como entregar determinados tipos de software em produção;
 - a. Entregando Django (python) em produção;
 - b. Entregando Ruby on Rails em produção;
 - c. etc.
3. O que fazer após entregar em produção;
 - a. Monitoramento;
 - b. Log;
 - c. etc.

O que você precisa saber pra ler esse livro?

O objetivo desse livro é que ele seja para qualquer pessoa da área de TI, ou seja, para quem está começando também. Alguns termos poderão ser difíceis caso você não seja da área ou tiver pouca experiência, mas em casos como esse, por favor abra um ticket [aqui](https://github.com/gomex/deploy-em-producao/issues)²⁴ com seu feedback, pois apenas assim será possível fazer um livro que sirva para todas as pessoas.

Será necessário muito feedback e colaboração de todas pessoas para entregar um conteúdo que sirva tanto para quem começa, como para quem já está com mais experiência. Esse é o desafio.

Resumindo, você precisa apenas ter um pouco de paciência e vontade de aprender coisas novas. Tenha em mente que se você não entendeu, provavelmente o erro está no livro e não em você.

²⁴<https://github.com/gomex/deploy-em-producao/issues>

O que é deploy? (Rafael Gomes)

Introdução

Há muito tempo o termo deploy é utilizado na área de Tecnologia da Informação (TI) e as pessoas que trabalham nesta área provavelmente já se deparou em algum momento com ele, mesmo que “nunca tenha feito um” provavelmente já tem uma ideia do que ele é.

Vamos então, juntos, estabelecer uma definição do que é na prática o famoso deploy, como ele funciona, e porque é tão importante para área de TI.

Esta palavra tem origem na língua inglesa e sua tradução para o português seria provavelmente: posicionar.

Quando se fala em “fazer deploy”, imagine que isso significa uma forma de posicionar algo, ou seja, é basicamente pegar algo que está em uma posição/localização e colocar em outra.

Isso quer dizer que quando alguém falar que vai “deployar” algo, é basicamente o jeitinho brasileiro de usar uma palavra em inglês e verbaliza-la em português, o que não é de todo ruim, uma vez que quem escuta entende perfeitamente, ou seja, a comunicação funciona e não há problema algum com isso. Chato mesmo é alguém que complica a comunicação para forçar palavras não usuais e assim tentar demonstrar uma certa superioridade linguística. Aconselha-se a leitura [desse livro](#)²⁵ sobre preconceito linguístico.

Como funciona o deploy de um produto de software

O que aqui se chama de “produto de software” é qualquer conjunto de arquivos que tenha como objetivo entregar uma funcionalidade como produto final. Um exemplo seria um site, que pelo conjunto de arquivos HTML, CSS e JavaScript, entrega uma exibição de informações que é traduzida pelo seu navegador e assim você pode ter acesso a informações navegando na internet.

O deploy é o ato de pegar esse conjunto de arquivos e levar até um determinado lugar. Esse lugar é normalmente um servidor, que hospedará esse software e exibirá para o usuário sempre que solicitado.

Destinos possíveis de um deploy

Normalmente um software passa por alguns destinos antes de chegar no seu habitat final, que é o ambiente de produção. O que se chama de produção, dessa forma, é basicamente o lugar oficial de onde os usuários finais deste produto poderão acessá-lo.

²⁵<https://www.amazon.com.br/Preconceito-Lingu%C3%ADstico-Marcos-Bagno/dp/8579340985>

As boas práticas apontam que antes de chegar no ambiente de produção esse software passe por outros ambientes, que normalmente são os seguintes, e muitas vezes segue também nessa ordem:

- Desenvolvimento
- Teste/Staging
- Produção

Desenvolvimento

É o local no qual a pessoa que desenvolve tem acesso direto, é onde se executa rapidamente o código, a fim de verificar se o que está sendo escrito atenderá as expectativas da funcionalidade que está sendo desenvolvida

Normalmente essa é a máquina da pessoa que desenvolve o software, e o verbo “deployar” faz pouco sentido aqui, porque não há uma movimentação de código, uma vez que este será usado na mesma máquina onde foi produzido.

Em alguns casos, a infraestrutura necessária para simular o ambiente de produção é tão complexa que é preciso um ambiente de desenvolvimento fora da máquina local, neste caso, o verbo “deployar” volta a ter seu sentido completo, pois o código será transferido para um outro ambiente, no caso, um de desenvolvimento remoto.

Teste/Staging

Não existe um nome para esse ambiente que possa ser considerado unânime, mas esse é o ambiente no qual se espera que o software esteja mais maduro, isso quer dizer que aqui o código já passou por alguma análise e está pronto para ser validado por outras pessoas.

O que aqui é chamado de **análise** será mais detalhado nos capítulos posteriores mas, por hora, basta saber que esse é o processo usado para avaliar se há algum problema no código, normalmente de forma manual, e feito por uma outra pessoa, que analisa seu código a fim de encontrar possíveis erros.

Esse é, em via de regra, o último local que o código “visitará” antes de ser conduzido para o ambiente que será usado pelos usuários reais do serviço. Ou seja, é aqui o local no qual costumeiramente acontecem os testes mais “pesados”.

Esses testes muitas vezes usam dados mais próximos do que os que seriam usados no ambiente real de forma que validações bem mais elaboradas podem acontecer. Habitualmente os times de software simulam o uso do sistema, de forma automatizada ou não, a fim de encontrar possíveis erros e esses tipos de testes serão assunto de outros capítulos. Por agora basta sabermos que é nesse ambiente que isso habitualmente acontece.

É importante salientar que se você copiar os dados de produção para ajudar na validação dos ambientes de “teste” e/ou desenvolvimento, deverá lembrar de apagar dados pessoais das pessoas que utilizam seu sistema. Imagine se fosse você a pessoa que utiliza um sistema, sabendo que seus dados pessoais estão acessíveis para qualquer membro da equipe de desenvolvimento?

Produção

Aqui é oficial, todo produto agora pode ser utilizado pelos clientes. É onde tudo ocorre “para valer” é normalmente maior (em quantidade de recursos) e mais restrito (no que tange a como quem pode fazer alterações como um deploy). É comum o ambiente de produção ser composto por, no mínimo, duas máquinas configuradas com o mesmo conteúdo. Isso ocorre para criar uma situação chamada de alta-disponibilidade. Neste cenário o serviço é mais resiliente e, caso uma das máquinas seja perdida por falhas inesperadas, o serviço continuará disponível.

Usando o exemplo anterior do site, basicamente, seria a hipótese de se ter duas máquinas hospedando os mesmos arquivos do site e, caso aconteça uma falha elétrica, ou qualquer outro problema em uma das máquinas, a segunda pode assumir o serviço sozinha sem muitos prejuízos à disponibilidade do serviço ofertado, que, neste cenário, significa exibir o site para as pessoas que o acessam.

Na prática, como funciona o deploy?

Frequentemente, o ato de fazer deploy se resume às ações de copiar os arquivos de um lugar - que pode ser o repositório de código ou de artefato - e depositar ele no ambiente de destino.

Seguindo o exemplo anterior do site, o ato de fazer o deploy corresponderia a copiar os arquivos HTML, CSS e JavaScript, que estão no repositório de código, e depositá-los no servidor que hospedará aquele ambiente.

Deploy para testes do site usado como exemplo acima? O ato de fazer deploy seria resumido a copiar os arquivos do repositório de código e depositá-los no servidor que foi designado para ser ambiente de testes.

O que é pipeline

Introdução

O pipeline usado para entregar software segue o mesmo conceito usado normalmente nas indústrias, que é uma esteira metálica que faz o produto “se mover” por dentro da fábrica, e os robôs, que estão parados, montam o produto à medida que ele passa.

Usando o exemplo da fabricação de carros, primeiro a estrutura metálica do carro (chassi) é colocado no pipeline e ela é movimentada pela fábrica, o primeiro robô, que está parafusado no chão, é responsável por pintar o chassi completamente. Sendo assim a cada chassi colocado no pipeline o primeiro robô pintará ele automaticamente.

Quando o chassi chega no primeiro robô, o pipeline para, pois o robô precisa de um tempo no processo de pintura, e somente após terminar o pipeline se movimenta novamente, para que o chassi seguinte seja movido para passar pela etapa de pintura e assim sucessivamente.

O segundo robô nesse pipeline é responsável por colocar as rodas e nesse processo o pipeline também espera ele terminar, e somente quando ele acaba o pipeline se move novamente.

No pipeline de software é bem parecido, sendo que ao invés do chassi, nesse modelo temos o código como “objeto” a ser movido pela esteira.

A grande diferença entre os modelos está na natureza do “objeto” usado para construir o produto. No caso do pipeline de carros, o chassi tenderá a sempre ser o mesmo, mas no caso do código é o exato oposto. O código tenderá a ser sempre diferente. Cada execução do pipeline, o “objeto” normalmente será diferente.

Como inicia um pipeline?

Na entrega de software o código vem do controle de versão, que é o local onde ficará armazenado todo código produzido pelo time de desenvolvimento.

O repositório de código armazenado no controle de versão normalmente tem uma ligação com a ferramenta responsável pelo pipeline de software, sendo assim é correto dizer que, em casos como esses, quando o código é modificado essa ação automaticamente ativa a execução do pipeline, ou seja, uma pessoa adicionou uma linha nova dentro do código? O pipeline será iniciado automaticamente com base nesse fonte atualizado.

Etapas do pipeline de software

O código é colocado na “esteira” e ela caminha de forma parecida com o que foi usado no exemplo da montagem do carro, ou seja, o código chega no primeiro “robô” e ele executa uma função específica e repetitiva. Esse momento é normalmente chamado de “**step**”. A tradução para português seria “**etapa**”, mas usaremos o termo em inglês, pois além de introduzir e fixar um termo tão importante nesse idioma, é esse nome que é usado em boa parte das ferramentas de mercado.

É correto dizer que a execução completa de um pipeline é a “movimentação” do código por múltiplos **steps**, ou seja, o código é depositado na esteira e transferido para o primeiro **step** que fará a primeira intervenção no código, que pode ser uma validação estática do código. Esse mesmo código, que foi validado na primeira **step** passa para uma nova, que pode ser a responsável por transformar esse código em um executável, e na posterior esse artefato é armazenado em repositório.



Pipeline com três steps

Perceba que no exemplo demonstrado na imagem o mesmo código passou por três **steps** diferentes na mesma execução do pipeline.

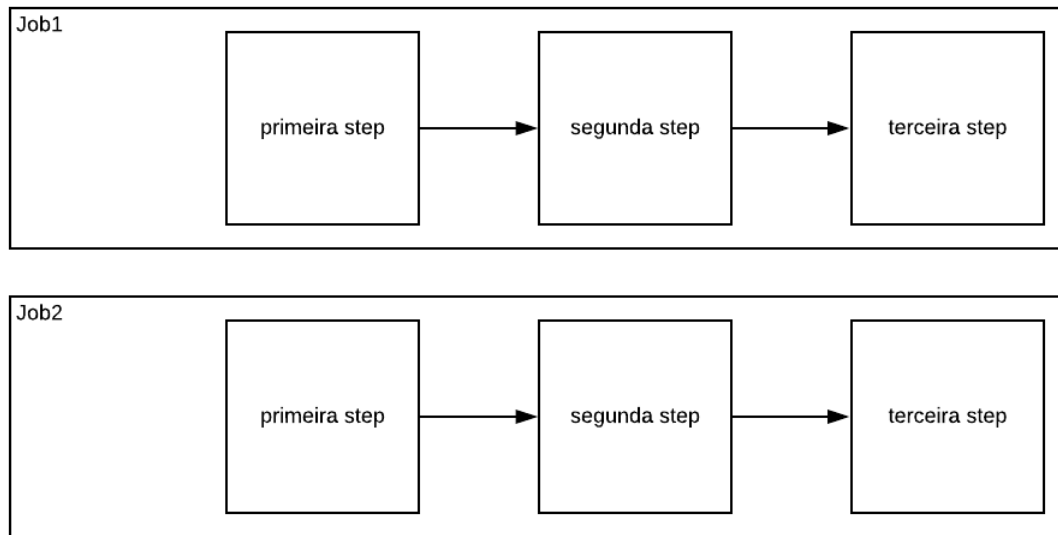
Cada **step** do seu pipeline é um comando, que é executado em uma console. Essa execução normalmente é executada dentro da pasta que tem os arquivos atualizados que foram recém baixados do controle de versão.

No exemplo anterior, a primeira **step** seria um comando para avaliar estaticamente o código fonte que está na sua pasta local, a segunda seria o comando para fazer ****build*** e o terceiro um comando para fazer upload do binário construído na **step** anterior para um repositório de artefatos.

Separação por “job”

Como já sabemos que no pipeline existem várias **steps**, que são responsáveis por executar ações no código a medida que elas avançam na esteira de entrega de software, é importante salientar que existe um outro nível de abstração chamada de **job**. A tradução para português seria “trabalho”, mas usaremos o termo em inglês para facilitar seu uso no futuro, pois esse é o nome que muitas vezes é usado pelas ferramentas de mercado.

O **job** é um conjunto de **steps**, onde essas **steps** normalmente são executadas sequencialmente por padrão, ou seja, a segunda **step** só será executada após terminar a primeira.



Pipeline com dois jobs

Se seu pipeline tiver vários **jobs** geralmente eles serão executados em paralelo, ou seja, se você tiver um **job** que executa seu código em uma plataforma específica, e um outro job que executa em outra. Os dois **jobs** serão iniciados ao mesmo tempo e não haverá nenhuma hierarquia entre eles, a não ser que seja explicitamente descrita.

Obs: Algumas ferramentas de pipeline não tem esse conceito de **jobs** ou utilizam outro nome. Lembre-se que essa explicação tem como objetivo oferecer uma ideia dos fundamentos.

Onde é executada essa “step”?

Como cada **step** tem seu comando, é importante saber onde esse comando é executado, pois quando o pipeline é iniciado, o código é copiado para uma pasta local e os comandos são executados no mesmo lugar.

Normalmente é usado um outro servidor para clonar o código e executar todos os comandos de cada **step**. Essa máquina é comumente chamada de **agente**.

Esse agente é o sistema que será usado para executar os comandos. Ele conecta no servidor de pipeline para buscar as informações necessárias para executar os **steps** e **jobs** da forma que foi configurada.

Esse agente pode ser de fato uma máquina, mas também pode ser um container. Esse modelo é inclusive a melhor forma de utilização de agentes, pois não há necessidade de manutenção de servidores, que consome um tempo de gerência absurdo do time responsável pela infraestrutura.

Imagina ter de instalar uma máquina, especificar todos os pacotes, bibliotecas e afins que é necessário para diferente pipeline que tem dentro de uma organização? E se o time de produto quiser usar uma

linguagem nova? Abre um ticket pro time de infra e espera ele criar um agente novo? Não precisa. Usando o docker você mesmo pode criar sua imagem ou utilizar as milhares que existem prontas nos repositório públicos de imagens que temos hoje na internet.

Vale salientar que mesmo utilizando containers como agente, ainda é aconselhável ter uma outra máquina, pois é nela que será instalado o software responsável pela gerência dos containers e onde será iniciado os containers, ou seja, basicamente os **jobs** serão executadas nesse agente, mas não no sistema operacional padrão e sim naquela que foi iniciado dentro do container.

Se você ainda não sabe como funciona containers, aconselhamos a leitura do livro [Docker Para Desenvolvedores](#)²⁶, mas acredito que o conhecimento de containers não vai afetar a sua capacidade de assimilar esse conceito sobre pipelines. Basta saber que o container é um processo rodando em um sistema operacional isolado, mas que todos os containers do mesmo **host** rodam na mesma máquina.

Quebrando o pipeline

Usualmente as ferramentas de pipeline só permitem que a segunda **step** seja executada se a primeira for finalizada com sucesso. Isso é um comportamento extremamente esperado, porque a ideia do pipeline é justamente garantir que as ações sejam executadas em sequência, pois elas em geral são configuradas de forma gradual, ou seja, as primeiras **steps** fazem as primeiras validações, e as construções e outras intervenções mais críticas e demoradas acontecem depois. Isso quer dizer que se uma validação inicial falhar, e essa validação por via de regra é mais rápida, poupa o tempo de esperar a falha da etapa de construção de artefato, que costumeiramente é mais demorada.

É comum encontrar pessoas afirmando que o objetivo de um pipeline é “quebrar”, pois é nesse processo que se percebe se o código enviado para esteira de fato está preparado para ser entregue ou não.

A automatização das validações e construções são parte central de um processo de entrega de software moderno.

Conclusão

O pipeline é uma abstração, onde temos **jobs** e **steps** compondo as etapas para construção de um produto a ser entregue no final da esteira. Tendo isso em mente, podemos pensar que muito mais do que apenas a ferramenta, o pipeline serve também para criar um fluxo rápido de feedback, onde em caso de quebra podemos entender que aquele código precisa de cuidados até que o problema seja resolvido.

²⁶<https://leanpub.com/dockerparadesenvolvedores>

O que é Pull Request? (Rafael Gomes)

Introdução

Antes de entender o que é um Pull Request é necessário entender os conceitos de Branch, Fork e o básico dos repositórios git. Esse [site²⁷](http://rogerdudler.github.io/git-guide/index.pt_BR.html) pode lhe ajudar nisso.

Uma vez que você já sabe que a Branch é uma ramificação do código, eu acrescento que o Fork é uma cópia inteira do seu repositório. Essa cópia mantém uma ligação simbólica entre o Fork e o repositório de origem. De forma prática, essa ligação não tem grande efeito no uso do dia a dia, ou seja, se alguém fizer fork do seu repositório e fizer mudanças nesse fork o seu repositório não será afetado automaticamente.

O seu repositório original só poderá ser alterado com commits diretos ou através de um Pull Request. Para evitar qualquer confusão, vamos dar nomes aos repositórios. Digamos que nós temos dois repositórios aqui:

- **repositório original**, que é o primeiro repositório, aquele que foi a origem do Fork.
- **repositório fork**, que é uma cópia exata do **repositório original** no momento do Fork.

O Pull Request, é o pedido para que o **repositório original**, ou uma branch do **repositório original**, faça a ação de pull (puxar) as atualizações do **repositório fork** ou de um branch do próprio repositório. Confuso, né? Vamos para um exemplo.

Imagine que você tem um repositório que tem o código para um site, nesse site você recebe como entrada num campo o tempo de vida de um cachorro e você faz a conta para saber qual a “idade de cachorro” dele, depois de um tempo com esse projeto no ar uma pessoa muito interessada no seu projeto propõe colocar uma opção também para gatos.

Só pra alinhar e facilitar o entendimento, o nome do repositório exemplo é **gomex/idade-de-animal**.

Essa pessoa que pretende colaborar, faz um fork do seu repositório e agora ela tem um repositório chamado **colaboradora/idade-de-animal** (imaginando que o usuário dessa pessoa seja “colaboradora”, ok?). Esse novo repositório tem uma ligação simbólica com o **gomex/idade-de-animal**.

²⁷http://rogerdudler.github.io/git-guide/index.pt_BR.html



Repositório Fork

Todas as mudanças feitas no “colaboradora/idade-de-animal” serão visíveis apenas nesse repositório e todas as mudanças feitas no “gomex/idade-de-animal” depois desse fork não serão automaticamente atualizadas no “colaboradora/idade-de-animal”, mas por definição não seria um problema, afinal a funcionalidade que a pessoa está trabalhando deve ser específica, ou seja, o que ela está trabalhando não deveria conflitar com as alterações que acontecem no repositório original, ou seja, não tem mais ninguém além dela trabalhando em “idade de gato”, né? Falaremos sobre resolução de conflitos depois.

Depois que a colaboradora adiciona a funcionalidade de calcular a idade de gato o que ela faz? Ela faz um pedido para que o repositório “gomex/idade-de-animal” puxe (pull em inglês) tudo que “colaboradora/idade-de-animal” tem diferente do seu repositório e agora essa diferença faça parte do repositório oficial. Isso é o Pull Request. Um pedido para que o repositório original se atualize a partir de mudanças feitas no repositório novo criado a partir de um fork.

Você pode estar se perguntando “E se alguém nesse meio tempo adicionou uma funcionalidade nova tipo ‘idade de papagaio’, isso pode afetar o Pull Request do idade de gato?” A resposta é: depende.

Se a funcionalidade for feita no mesmo local de código mas em linhas diferentes, não teremos problemas. Caso as alterações ocorram nos mesmos arquivos e linhas aí sim teremos um conflito e trataremos disso em outro capítulo.

A sugestão é que funcionalidades diferentes sejam tratadas de forma isolada, a fim de não causar conflito algum no processo.

Todo esse processo que descrevi aqui, pode ser feito também baseado em branch, mas a pessoa que colabora precisa ser membro do repositório e não uma pessoa aleatória na internet, pois ela precisa ter permissão para criar branch no repositório. No fim é o mesmo propósito, mas ao invés de repositório inteiro, tudo que expliquei aqui acontece no nível de ramificações.

Por fim, note que em outras plataformas de repositórios online, o conceito de Pull Request pode ter outros nomes como por exemplo Merge Request(Requisição para merge).

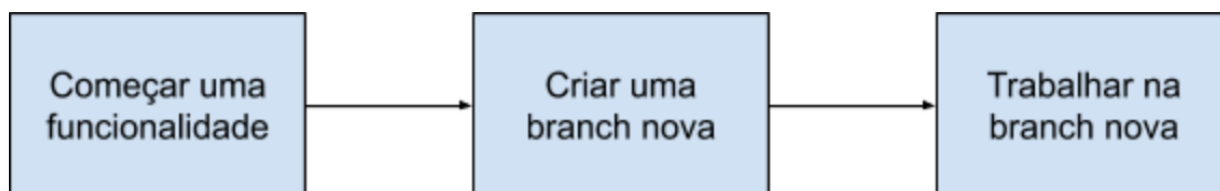
Como usar Pull Request para o processo de revisão?

A maioria das organizações utiliza o Pull Request como mecanismo padrão para revisão de código, pois ele é basicamente a “porta de entrada” para a base “oficial” de código, seja em relação ao repositório ou branch.

Normalmente as branches que serão usadas para construir o artefato final do repositório oficial são protegidas e não podem receber commits diretos, ou seja, tudo que entra nessas branches devem entrar por um PR (Pull Request). Existe a possibilidade do administrador do repositório mandar o código direto, mas isso deve ser apenas uma exceção. Dito isso, eu reforço, **mesmo os administradores do repositório, pessoas desenvolvedoras experientes**, ou até mesmo a **liderança técnica** do time devem mandar suas mudanças por PR e elas devem ser avaliadas por outras pessoas.

Quando começar a trabalhar em uma funcionalidade nova do repositório. Eu faço parte da organização? Tenho acesso a criar uma branch? Caso positivo, eu crio uma branch.

Existe um Padrão para criação de branch? Eu gosto do modelo “feature/nome-da-funcionalidade” assim fica muito claro para todo mundo no que você está trabalhando. Se você usa algum sistema de ticket para gerenciar as tarefas você pode colocar o identificador do ticket também: “feature/nome-da-funcionalidade#435”.

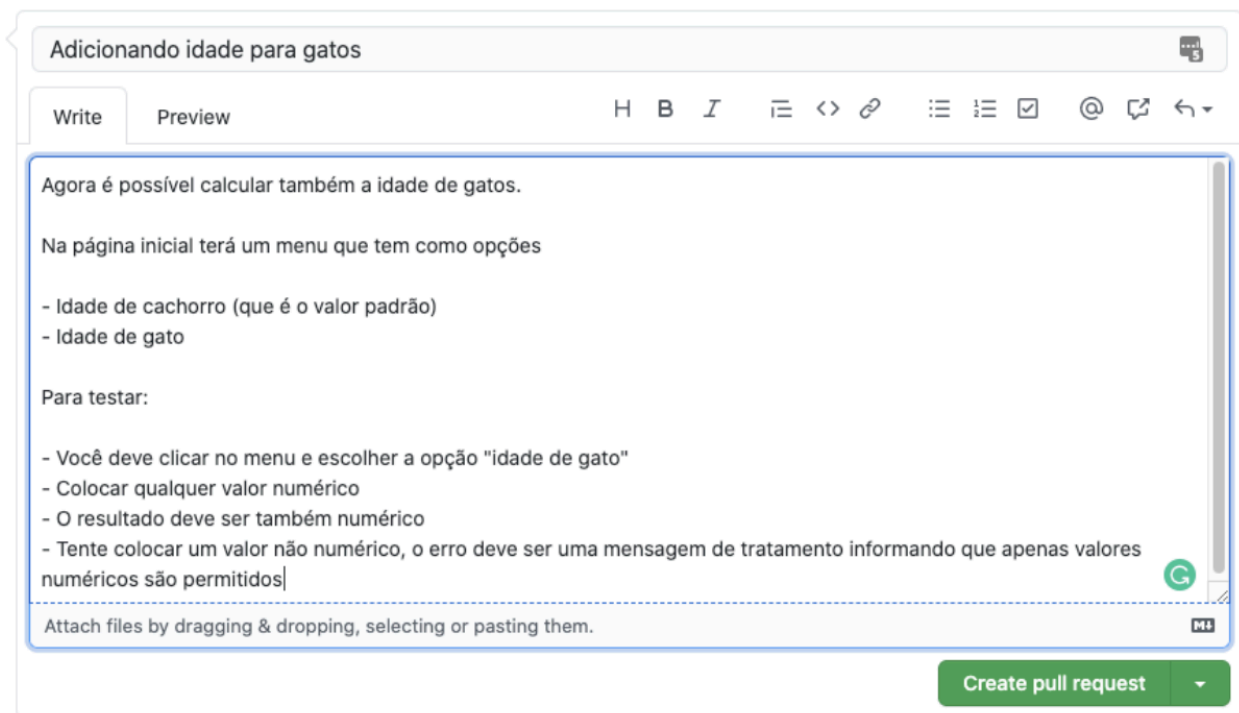


Proposta de fluxo para Pull Request

Lembre-se que sua branch precisa ser bem específica, ou seja, se “aparecer” outra demanda, o aconselhável é abrir outra branch a partir de branch “oficial” (que normalmente é a “master”).

Quando você tiver muita confiança que seu código entrega tudo que a funcionalidade precisa para existir, você deve abrir um PR e na descrição desse PR você deve detalhar qual comportamento esperar dessa mudança que você está propondo.

Segue abaixo um ótimo exemplo:



The screenshot shows a GitHub Pull Request (PR) description template. At the top, there's a title bar with the text "Adicionando idade para gatos" and a small icon. Below the title bar, there are two tabs: "Write" (active) and "Preview". To the right of the tabs is a rich text editor toolbar with icons for bold, italic, link, list, and other formatting options. The main text area contains the following content:

Agora é possível calcular também a idade de gatos.

Na página inicial terá um menu que tem como opções

- Idade de cachorro (que é o valor padrão)
- Idade de gato

Para testar:

- Você deve clicar no menu e escolher a opção "idade de gato"
- Colocar qualquer valor numérico
- O resultado deve ser também numérico
- Tente colocar um valor não numérico, o erro deve ser uma mensagem de tratamento informando que apenas valores numéricos são permitidos

At the bottom of the text area, there's a dashed line and the text "Attach files by dragging & dropping, selecting or pasting them." To the right of the text area is a green button labeled "Create pull request".

Exemplo de Pull Request

O ideal é que o PR tenha o seguinte conteúdo:

- Descrição clara e objetiva do comportamento que será adicionado caso o PR seja aceito;
- Mínimo de detalhe sobre como a nova funcionalidade é usada, talvez um link para uma documentação externa seja uma boa, caso o detalhe seja muito grande;
- Passo a passo sobre como testar, da forma **mais direta e clara** possível e quais comportamentos esperados para os testes executados, ou seja, se for para clicar, diga onde clicar e o que deve acontecer se clicar no lugar informado, se possível diga também o que **não** deve acontecer.

Uma descrição seguindo esse modelo ajudará a pessoa que vai avaliar seu PR e ela talvez não precisará lhe perguntar nada, pois tudo que precisa saber sobre o trabalho e como avaliar ele está descrito lá.

Acredite, cinco ou dez minutos investidos na criação de uma boa descrição de PR pode lhe “salvar” várias interrupções para explicação da sua mudança.

A dificuldade em escrever na descrição do seu PR é um possível indicativo que você não está confiante e não tem uma real noção sobre o que foi entregue. Imagine que talvez esse seja o momento de você organizar mentalmente está entregando realmente.

Algumas pessoas criam uma PR draft (rascunho) para ir atualizando a medida que vão mexendo no código. Eu gosto desse modelo, pois assim nada se perde e você não precisa lembrar de tudo que foi feito em horas de trabalho naqueles últimos minutos de trabalho antes de entregar sua tarefa.

Como revisar o Pull Request?

A pessoa que vai olhar um PR ela precisa ter em mente alguns pontos:

- Qual o objetivo daquele PR? Ele está claro na descrição?
- As mudanças que estão sendo propostas no PR seguem o padrão que é usado nessa organização?
- A forma que a pessoa entregou à funcionalidade é a melhor? Existe maneira mais eficiente de fazer a mesma coisa?
- Os testes descritos no PR são o suficiente?

Qual o objetivo daquele PR? Ele está claro na descrição?

É importante estar **muito** claro sobre o que se trata o PR em questão, pois a sua avaliação será com base nisso, sendo assim, a primeira coisa a analisar é a clareza no que está sendo entregue, se houver qualquer inconsistência nesse momento você deve pontuar e deixar claro baseado em que está fazendo a avaliação.

Um exemplo:

Você abre o PR sobre idade de gatos, lê a descrição e não está claro pra ti se a ideia é criar uma forma separada para calcular idade de outros animais ou apenas colocar uma opção na lógica atual feita para cachorro, sendo assim seu comentário poderia ser:

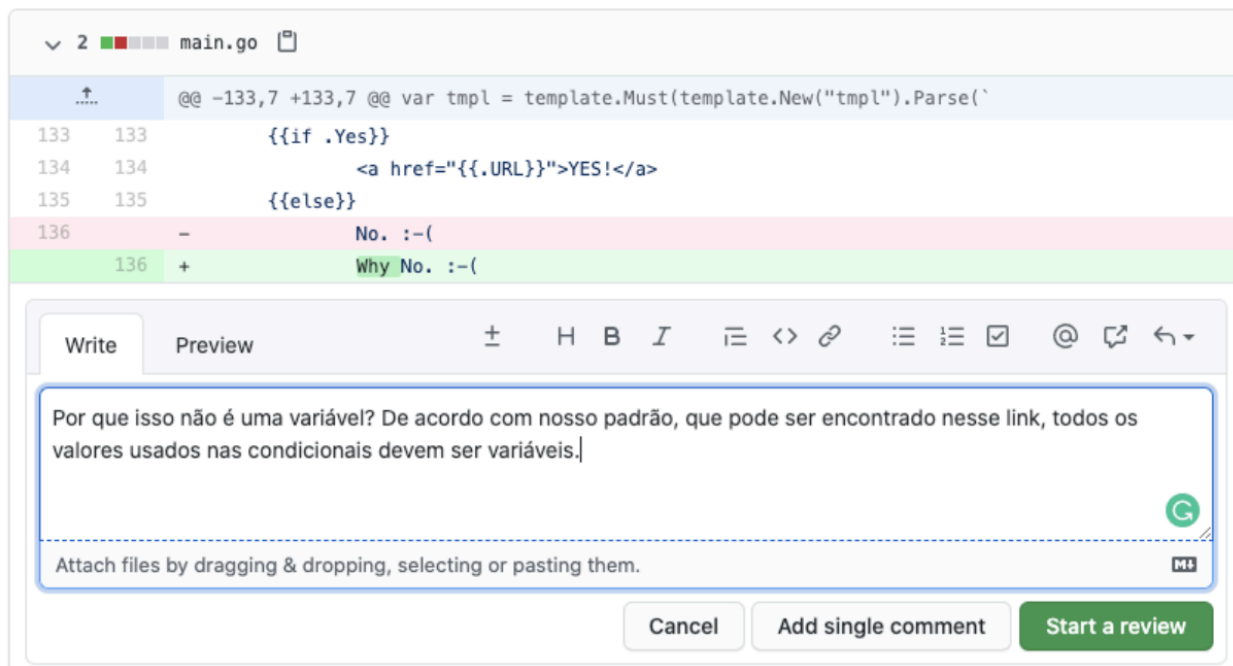
“Não está claro pra mim se você colocou a lógica de calcular idade pra gato separado porque seja de fato a forma que você acha que seja ideal ou se fez isso apenas para não conflitar com o código original por agora e refatorar no futuro. Eu vou analisar seu código atual separado mesmo, mas adianto que mudar para que evite repetição de código seja uma boa no futuro”

Pronto, com isso você está dizendo que sua análise não levará em conta a quantidade de repetição de código que isso possa gerar e que você não concorda, mas que por agora não vê problemas nisso.

As mudanças que estão sendo propostas no PR seguem o padrão que é usado nessa organização?

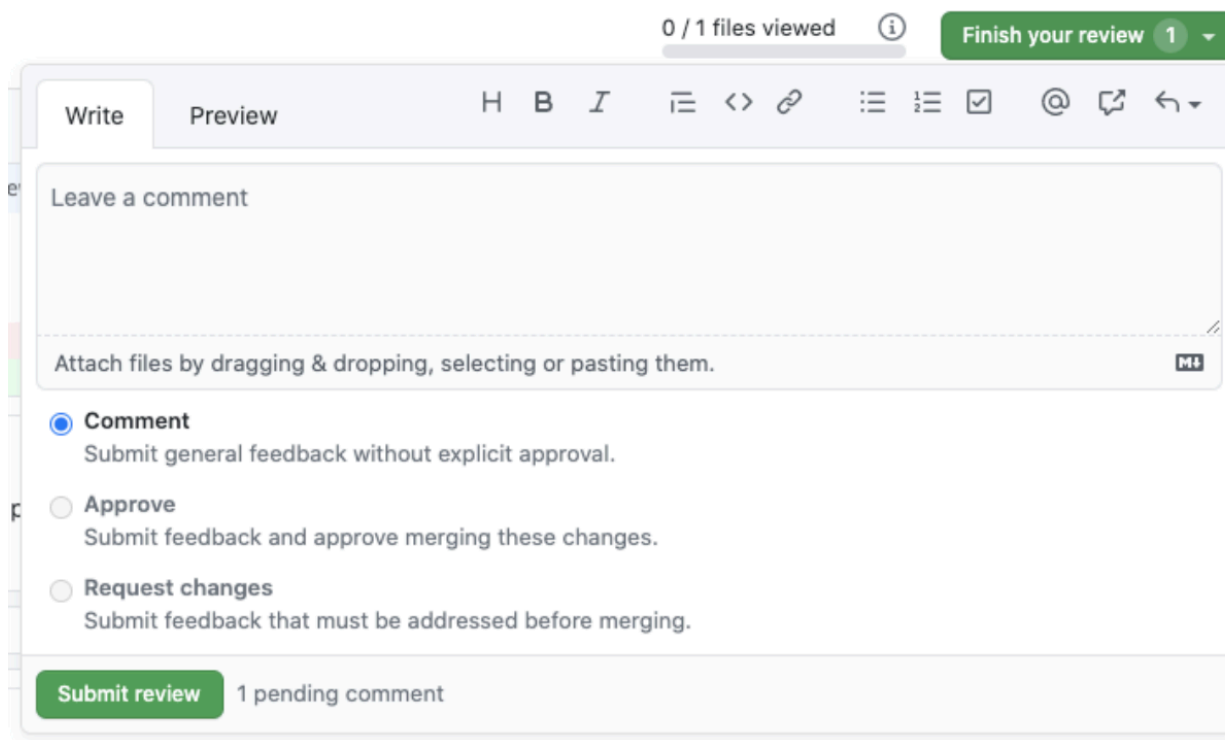
A maioria das organizações seguem alguns padrões para como escrever código, seja em sua formatação (ex. quatro espaço, ponto e vírgula em cima ou embaixo) ou em como organizar funções, métodos e afins.

Esse padrão deve estar claro em algum lugar, e a pessoa que vai colaborar deve ler isso antes, mas nem sempre isso é possível e dessa forma a colaboração pode não seguir esse padrão. Você que está avaliando deve deixar bem claro para esta pessoa qual regra ela está infringindo e qual parte do código isso acontece. O github oferece a funcionalidade de comentar nas linhas do código do PR.



Comentário no review

Depois que comentar todo o PR não esqueça de finalizar sua revisão, caso contrário a pessoa que fez o PR não verá seu comentário.



0 / 1 files viewed ⓘ Finish your review 1 ▾

Write Preview H B I ≡ <> 🔗 ≡ ≡ ☑ @ ↗ ↶

Leave a comment

Attach files by dragging & dropping, selecting or pasting them. 📎

☒ **Comment**
Submit general feedback without explicit approval.

☐ **Approve**
Submit feedback and approve merging these changes.

☐ **Request changes**
Submit feedback that must be addressed before merging.

Submit review 1 pending comment

Revisão de PR

Se precisar que a pessoa atualize algo para que o PR seja aceito escolha a opção de “Request changes” (Solicitar mudanças), caso contrário aprove ou comente sem aprovar, caso precise de mais tempo para decidir sobre aceitar ou não.

A forma que a pessoa entregou à funcionalidade é a melhor? Existe maneira mais eficiente de fazer a mesma coisa?

Esse ponto é um pouco abstrato, pois depende muito da experiência de quem está revisando, mas é talvez a parte **mais importante** desse processo de revisão. Está aqui a grande oportunidade de uma pessoa proporcionar para a outra que mandou o PR as maneiras de deixar o código ainda melhor.

ATENÇÃO!!! Não faça uso desse espaço para diminuir ou ridicularizar a pessoa que mandou o PR pois, caso faça isso, além de perder uma grande oportunidade de melhorar a habilidade de outra que você “julga” inferior, você também perderá a oportunidade de ser uma pessoa melhor. Ajudar as pessoas que trabalham no mesmo projeto que você é uma das coisas mais básicas quando trabalhamos em equipe. Caso tenha problemas em trabalhar dessa forma, aconselho criar um projeto onde você seja a única pessoa a enviar código.

Mais importante de que enviar as sugestões de mudança e apontar os erros do PR é validar se de fato isso é um erro ou uma abordagem diferente, da qual você discorda.

Se sua sugestão melhorar a performance do que será entregue, tente mostrar algum elemento que embase sua sugestão.

Se sua sugestão tem como objetivo seguir uma boa prática, aponte o link para onde a pessoa possa ler mais sobre ela e, se possível, aponte caminhos para que a pessoa possa aplicar aquela melhor prática de uma forma mais fácil. Essa é uma boa oportunidade para exercitar seu uso dessa boa prática também.

Os testes descritos no PR são o suficiente?

É importante avaliar se há testes o suficientes e não importa se os testes podem ser de exploração ou automatizados, você precisa praticar a avaliação disso. Não precisa ser uma pessoa especializada em QA (Quality Assurance) para fazer isso.

Ter uma pessoa QA no seu time é aconselhável, mas não ache que ela será a única a fazer essa análise. Nos primeiros PR você pode pedir a ajuda dela e fazer essa parte da avaliação juntas, mas aconselho que pratique o suficiente para internalizar esse tipo de revisão, pois a necessidade de entender qualidade de código, assim como segurança, “DevOps” ou afins deve ser de interesse de todos. Esses assuntos devem ser uma preocupação do **time** e não apenas de um cargo específico. A pessoa que está nesse cargo deve ser responsável por ajudar o time a evoluir nesse assunto, ajudando como uma espécie de consultor interno. Repito, essa pessoa **não** deve ser a única responsável sobre o assunto que é experiente.

Recebi uma lista imensa de coisas a corrigir no meu PR, fico triste?

Caso a pessoa que comentou no seu PR tenha sido respeitosa e cuidadosa ao criar o review não há motivos para tristeza. Encare essa longa lista de correções como uma boa experiência para melhorar sua habilidade em codificação ou documentação.

Tenha em mente que a pessoa que mandou o PR não é necessariamente melhor do que você. Ela apenas dedicou parte do seu tempo para sugerir melhorias em seu trabalho, teve a atenção e cuidado necessário para ajudar o time como um todo para entregar um código melhor para a organização. Ela provavelmente não tem nada contra você e quanto mais detalhista ela for, não entenda isso como a manifestação de um código de baixa qualidade e sim como um código que pode alcançar outro nível de qualidade, auxiliando você a perceber minúcias do seu trabalho.

Projetos e prazos apertados são duas coisas que normalmente andam juntos e uma longa lista de correções pode ser desanimadora, mas entenda que o problema está no prazo curto que normalmente as empresas trabalham. Nesse caso, o que pode ser feito se divide em duas possibilidades:

- Você pode calcular no futuro o prazo levando em consideração esse nível de exigência na revisão;
- Negociar com a pessoa que revisou partes das críticas, tentando explicar sobre os prazos e afins.

Uma dica aqui é fazer com que seu PR seja o menor possível, assim a possibilidade de retrabalho no retorno da revisão também será menor.

Outro ponto positivo de um PR mínimo está associado a dopamina que é um dos “hormônios da felicidade”, como você já deve ter percebido os jogos que mais jogamos costumam ter fases curtas, mas investimos horas e mais horas com eles em nossas telas, por qual motivo? Dopamina, pequenas recompensas neuronais que nos dizem que conseguimos alcançar um objetivo. Então quando estiver fazendo um PR, ao invés de tentar fazer uma fase longa e complexa, tente fazer “fases menores”, mais atômicas, e corrija coisas pequenas e rápidas (e da forma correta é claro!) para que você e o avaliador dos PRs possam fazer bom uso de pequenas e constantes doses de dopamina ao aprovar ou ser aprovado cada PR.

Quantas pessoas devem revisar meu código?

Algumas empresas colocam no mínimo duas, outra colocam três, mas existem muitas que com apenas uma revisão já torna o PR disponível para ser de fato aceito. Isso depende da empresa.

O ideal seriam duas pessoas, mas se isso atrasar demais o andamento do seu projeto, uma deve ser o suficiente, mas lembre que essa pessoa a revisar terá **muito** mais responsabilidade e normalmente não poderá ser uma pessoa com pouca experiência. Isso quer dizer que você estará usando ainda mais o tempo de pessoas mais experientes para avaliação de código do que de fato produzindo códigos.

Conclusão

O PR é um método ideal, simples, com possibilidade de interação assíncrona através de comentários no código, possibilidade de debate, múltiplas opiniões e uma forma centralizada de entender como seu código avançou ao passar do tempo, quais os motivos que deixaram determinado comportamento entrar no código e quais foram as argumentações que embasaram as decisões.

Uma ferramenta ideal, que se usada sabiamente, pode ser muito poderosa!

Estratégia de Testes - Construindo Confiança (Samanta Cicilia)

Introdução

Testes sempre fizeram parte do processo de desenvolvimento de software. Independente do método utilizado, manual ou automatizado, você provavelmente executa algumas validações para garantir que os requisitos foram cumpridos, essa é uma das formas de ter segurança antes de fazer as entregas em produção.

Garantir a qualidade é um ponto crítico para que tenhamos segurança nos deploys. Além de garantir essa segurança de que as mudanças sendo implantadas não introduzirão nenhum problema no ambiente de produção também é uma forma de ajudar com a produtividade do time. Tendo diferentes tipos de testes automatizados, o time de desenvolvimento percebe os problemas o mais cedo possível e consegue corrigi-los antes que eles impactem os clientes finais.

Nos últimos 20 anos a abordagem de testes evoluiu drasticamente para suportar a criação de sistemas cada vez mais complexos. Com isso a complexidade dos testes também aumentou o que tem exigido cada vez mais processos e ferramentas sofisticadas para que os testes não sejam uma dor de cabeça e sim uma forma de dar confiança para realização de mudanças em produção.

E como se não bastasse o aumento da complexidade dos sistemas, também enfrentamos um mundo onde as empresas precisam iterar e se adaptar o mais rápido possível para atender as expectativas de negócio. Se você tem práticas robustas de teste, você não sente medo de fazer aquele deploy na sexta-feira. :)

Por que escrever Testes

Quando falamos de teste, estamos basicamente falando de uma lista de passos que são executados seja manual ou automaticamente:

1. Em primeiro lugar você tem um comportamento que precisa ser verificado
2. Uma entrada que vai ser passada para o seu sistema
3. Uma saída que pode ser observada

Tudo isso dentro de um ambiente controlado. Quando você executa esse teste você descobre se o sistema se comporta ou não como o esperado. Há algum tempo atrás esse processo era feito

majoritariamente de forma manual, dependendo muito tempo, às vezes meses, para conseguir dizer se uma versão podia ser implantada em produção ou não.

Com o passar do tempo esse trabalho manual foi substituído por testes automatizados que podem ser reaproveitados e executados de forma mais rápida e assertiva em relação aos manuais.

Mas nem tudo são flores nessa história, criar e manter uma suíte de testes automatizados saudável requer esforço. Existem muitos desafios envolvidos que vamos ver no decorrer dos próximos tópicos.

Mesmo com esses desafios é importante entender que escrever testes é parte do processo de entrega de software e ajuda a manter a qualidade das entregas. Isso não pode ser negociado. É um esforço que compensa quando você consegue pegar os bugs antes de chegar em produção ao invés de esperar que algum cliente ligue reclamando.

Além de aumentar a confiança na publicação de novas versões, os testes servem como uma documentação de como o software deveria se comportar e te obriga a pensar também no design do que está sendo desenvolvido, já que isso implica na dificuldade ou facilidade de criar testes.

Definindo Testes

Existem muitas definições na literatura sobre quais testes criar e onde, uma das mais famosas é a pirâmide de testes citada pelo [Martin Fowler](https://martinfowler.com/articles/practical-test-pyramid.html)²⁸. Nessa visão a ideia é que você tenha mais testes na base da pirâmide (testes unitários) que são mais rápidos de executar, um pouco menos no meio (testes integrados) e menos ainda no topo (testes de interface e manuais).

No nosso contexto eu vou usar a categorização feita pelo Google, que pode ser conferida no livro [Software Engineering at Google](https://www.amazon.com.br/Software-Engineering-Google-Titus-Winters/dp/1492082791)²⁹. O Google utiliza dois critérios para categorizar os testes: tamanho e escopo.

Tamanho tem relação com a quantidade de recursos consumidos pelo teste e escopo tem relação com quanto código aquele teste está validando.

Eu acredito que essa é uma abordagem melhor para falar sobre teste já que pode ser aplicada a qualquer que seja o tipo do projeto: monolitos, microserviços, micro frontends, etc, e não fica preso a quantidade de testes que você precisa ter em cada uma das camadas. Utilizando esse conceito você se desprende um pouco daquela pressão de seguir a pirâmide e consegue ter uma outra visão para analisar que testes fazem sentido para você.

São 3 as categorizações:

- Pequeno: são os testes mais contidos, geralmente são utilizados dublês de teste para evitar chamadas externas às funções testadas. São testes rápidos e determinísticos;
- Médio: aqui temos testes que executam múltiplos processos mas ainda assim sem acessar componentes externos, nessa categoria entram os testes que acessam banco de dados por exemplo;

²⁸<https://martinfowler.com/articles/practical-test-pyramid.html>

²⁹<https://www.amazon.com.br/Software-Engineering-Google-Titus-Winters/dp/1492082791>

- Grande: esses testes são os que necessitam de uma maior complexidade para execução, nesse momento os sistemas já estão integrados. São mais lentos e menos determinísticos.

Dentro de cada um desses tipos de teste podemos ainda categorizar quais características de qualidade estão sendo avaliadas em funcionais e não funcionais. Quando falarmos de **testes funcionais** nos referimos à validação de características relacionadas ao comportamento da aplicação durante a utilização do sistema; quando falarmos de **testes não funcionais**, nos referimos à avaliação de aspectos de qualidade como desempenho, usabilidade, compatibilidade e outras “idades” onde o objetivo é validar essas características ao invés da funcionalidade em si.

Testes Pequenos

Testes Unitários

Testes Unitários são aqueles que tem um escopo mais limitado, normalmente uma simples classe ou método. Esses testes são os que vão te ajudar no dia a dia do processo de desenvolvimento já que eles são mais rápidos de executar, devido ao escopo mais contido. Isso ajuda a otimizar a produtividade dado que esses testes podem ser executados antes de fazer o push para o repositório.

Imagine a seguinte função onde, dependendo do idioma informado, você receberá como se escreve Paulo nesse idioma:

```
1  # handler.js
2  function getNameAccordingLanguage(language) {
3      switch(language) {
4          case "en":
5              return "Paul";
6          case "pt":
7              return "Paulo";
8          case "el":
9              return "Πάολο";
10         default:
11             return "";
12     }
13 }
14 module.exports.getNameAccordingLanguage = getNameAccordingLanguage;
```

E aqui temos um teste unitário que valida apenas o escopo dessa função, testando cada um dos idiomas contidos na função anterior en-pt-el, e um teste onde informamos um idioma inexistente para testar o resultado da condição default:

```
1 # __tests__/handler.test.js
2 const handler = require('../handler');
3
4 test('Name is informed based on Language', () => {
5   expect(handler.getNameAccordingLanguage("en")).toBe("Paul");
6   expect(handler.getNameAccordingLanguage("pt")).toBe("Paulo");
7   expect(handler.getNameAccordingLanguage("el")).toBe("Πάολο");
8   expect(handler.getNameAccordingLanguage("bla")).toBe("□□");
9 });
```

Esse é um teste que tem o escopo limitado, nesse caso testar que dada uma entrada, nesse caso o idioma, recebemos como resultado o nome “Paulo” de acordo com cada idioma.

Esses testes ajudam muito na manutenibilidade posto que como exercitam um escopo mais contido, se um deles quebra você consegue rapidamente identificar o ponto de falha, diferente de um caso onde, por exemplo, você tivesse mais elementos envolvidos como um banco de dados, um container da aplicação, etc.

No decorrer desse tópico vamos falar mais sobre isso, mas vale já começar a reforçar que os testes precisam trazer segurança para o time fazer o deploy em produção sem peso na consciência. Se os testes não revelam os bugs ou se quebram demais desnecessariamente o time acaba perdendo a confiança e a crença de que os testes são um elemento importante no processo de desenvolvimento. Por isso você deve tratar testes como trata código de produção: utilizando boas práticas, fazendo refatoração para implementar melhorias e sempre buscando otimização.

Não tem como falar de testes unitários sem tocar no assunto de duplês de teste, que é o que vamos abordar no próximo tópico.

Dublês de Testes

Como vimos a ideia dos testes unitários é ter um escopo mais limitado, mas como fazer isso se normalmente nossas funções acionam outros componentes como o próprio banco de dados, outras funções ou até mesmo sistemas externos?

É aí que entra o conceito de duplês de teste, uma forma de substituir esses componentes que são externos ao objetivo do nosso teste. Outro benefício de utilizar duplês é otimizar o tempo de execução dos testes.

Vamos utilizar o seguinte código como exemplo e o método que será testado é o `getPokemon` que retorna um pokemon cadastrado no banco de dados:


```
1  const Database = {
2    find() {}
3  }
4
5  class PokemonsController {
6    constructor(Database) {
7      this.Database = Database;
8    }
9
10   getPokemon() {
11     return this.Database.find('pokemon');
12   }
13 }
```

Os dublês podem ser categorizados em alguns patterns:

Fake

Os fakes possuem uma resposta fixa, independente de como são chamados, podem ser implementados através de uma classe ou função. Uma vantagem de usar fake é que você não precisa ter nenhuma dependência externa como uma biblioteca, mas por outro lado você só consegue validar a saída e não todo o fluxo de comportamento.

```
1  describe('PokemonsController getPokemon()', () => {
2    it('should return a Pokemon', () => {
3      const databaseResponse = {
4        id: 1,
5        name: 'Pikachu',
6        species: 'mouse',
7        type: 'eletric'
8      };
9
10     const fakeDatabase = {
11       find() {
12         return databaseResponse;
13       }
14     }
15     const pokemonsController = new PokemonsController(fakeDatabase);
16     const response = pokemonsController.getPokemon();
17
18     expect(response).toEqual(databaseResponse);
19   });
20 });
```

No exemplo acima criamos um fake que vai sempre retornar as informações do Pikachu quando for chamado dentro do teste, ou seja, ao invés de realmente acessarmos o banco de dados para pegar essas informações, o fake fará esse papel de fornecer os dados:

```
1  const databaseResponse = {
2    id: 1,
3    name: 'Pikachu',
4    species: 'mouse',
5    type: 'eletric'
6  };
7
8  const fakeDatabase = {
9    find() {
10      return databaseResponse;
11    }
12  }
```

Nesse teste, passamos para o `PokemonsController` o nosso fake, ao invés do Database de verdade, e verificamos que a resposta de “getPokemon” retorna os mesmos dados declarados no “databaseResponse”, que são os dados do Pikachu:

```
1  const pokemonsController = new PokemonsController(fakeDatabase);
2  const response = pokemonsController.getPokemon();
3
4  expect(response).toEqual(databaseResponse);
```

Spy

Os spies possibilitam a “gravação” do comportamento que está sendo espionado, assim podemos testar por exemplo se uma função foi chamada, quantas vezes ela foi chamada e quais os parâmetros. Aqui podemos testar um comportamento interno, o que é uma vantagem, mas não múltiplos comportamentos de uma vez. Para criar spies precisamos da ajuda de bibliotecas da própria linguagem. Dessa vez vamos precisar da ajuda da biblioteca [sinonjs](https://sinonjs.org/)³⁰ para criar o spy.

³⁰<https://sinonjs.org/>

```
1 describe('PokemonsController get()', () => {
2   it('should find a pokemon from database with correct parameters', () => {
3     const find = sinon.spy(Database, 'find');
4
5     const pokemonsController = new PokemonsController(Database);
6     pokemonsController.getPokemon();
7
8     sinon.assert.calledWith(find, 'pokemon');
9     find.restore();
10   });
11 });
```

Aqui adicionamos um spy na função “find” para que o Sinon devolva uma referência a essa função:

```
1 const find = sinon.spy(Database, 'find');
```

No assert verificamos se a função foi chamada com o parâmetro esperado que é “pokemon”, observe que diferente do fake, nesse caso estamos passando o Database que foi “espiado” pelo sinon. No final restauramos a função original utilizando find.restore().

```
1 const pokemonsController = new PokemonsController(Database);
2 pokemonsController.getPokemon();
3
4 sinon.assert.calledWith(find, 'pokemon');
5 find.restore();
```

Stub

Diferentes dos spies, os stubs conseguem mudar comportamentos, dependendo de como forem chamados, permitindo testar mais cenários. Pode ser usado inclusive para testar código assíncrono.

```
1 describe('PokemonsController getPokemon()', () => {
2   it('should return a pokemon info', () => {
3     const databaseResponse = {
4       id: 1,
5       name: 'Pikachu',
6       species: 'mouse',
7       type: 'eletric'
8     };
9
10    const find = sinon.stub(Database, 'find');
11    find.withArgs('pokemon').returns(databaseResponse);
```

```
12
13     const pokemonsController = new PokemonsController(Database);
14     const response = pokemonController.getPokemon();
15
16     sinon.assert.calledWith(find, 'pokemon');
17     expect(response).to.be.eql(databaseResponse);
18     find.restore();
19   });
20 });
```

Nesse exemplo “injetamos” os dados do Pikachu para que a nossa função `getPokemon` retorne esses dados. Lembrando que não estamos acessando o banco de dados de verdade em nenhum momento:

```
1  const databaseResponse = {
2    id: 1,
3    name: 'Pikachu',
4    species: 'mouse',
5    type: 'eletric'
6  };
7
8  const find = sinon.stub(Database, 'find');
9  find.withArgs('pokemon').returns(databaseResponse);
```

Depois verificamos se a função foi chamada da forma correta e se recebemos o resultado esperado. No final restauramos a função original utilizando `find.restore()`:

```
1  sinon.assert.calledWith(find, 'pokemon');
2  expect(response).to.be.eql(databaseResponse);
3  find.restore();
```

Mock

Os mocks são capazes de substituir a dependência permitindo assim verificar vários comportamentos. Você pode utilizar por exemplo para verificar se uma função foi chamada e se ela foi chamada com os argumentos esperados.

Embora mock tenha comportamento parecido com spy e stub é importante não confundir com ambos. Para aprofundar recomendo a leitura do texto [Mocks Aren't Stubs](https://martinfowler.com/articles/mocksArentStubs.html)³¹.

³¹<https://martinfowler.com/articles/mocksArentStubs.html>

```
1 describe('PokemonController get()', () => {
2   it('should call database with correct arguments', () => {
3     const databaseMock = sinon.mock(Database);
4     databaseMock.expects('find').once().withArgs('pokemon');
5
6     const pokemonsController = new PokemonsController(Database);
7     pokemonsController.get();
8
9     databaseMock.verify();
10    databaseMock.restore();
11  });
12 });
```

Primeiro criamos o mock do nosso Database:

```
1 const databaseMock = sinon.mock(Database);
```

Depois temos 2 asserções, a primeira para verificar se o método “find” foi chamado uma vez e na segunda se ele foi chamado com o argumento “pokemon”:

```
1 databaseMock.expects('find').once().withArgs('pokemon');
```

Temos o “verify()” que verifica se as expectativas foram atingidas e no final restauramos a função original utilizando ‘find.restore()’:

```
1 databaseMock.verify();
2 databaseMock.restore();
```

Mockar ou não mockar: eis a questão

Quando se fala da utilização de dublês existe quase uma questão filosófica: **“mockar ou não mockar, eis a questão”**. Existem alguns casos que são inevitáveis, como por exemplo testar funções que disparam email, ou utilizam alguma integração externa, nesses casos os dublês com certeza trazem produtividade ao tornarem a execução dos testes mais rápida e menos intermitente.

Por outro lado precisamos lembrar que não estamos testando o comportamento 100% como vai ser executado em produção. Por isso é importante considerar alguns pontos antes de optar pelo uso de dublês e além disso ter testes de diferentes tipos que ajudem nessas validações.

Se a implementação real permite uma execução rápida, determinística e simples, faz mais sentido utilizar essa implementação nos testes. Por exemplo uma função que valida CPF, datas, endereço, listas, etc.

Se não for esse o caso, você precisa avaliar o tempo de execução, o quanto o teste é determinístico ou não (se você não consegue controlar o teste, as chances de você ter intermitência aumentam absurdamente) e o quanto é fácil ou difícil construir as dependências.

Lembre-se de avaliar seu contexto SEMPRE!! E optar pela solução que traz mais segurança para o seu processo de desenvolvimento.

Para se aprofundar nesse assunto eu indico a leitura do [xUnit Patterns - Test Double](http://xunitpatterns.com/Test%20Double.html)³².

Testes Médios

No tópico anterior abordamos testes que são mais auto-contidos, aqui já começamos a falar de algumas integrações entre componentes para validar os fluxos.

Testes de Integração

Nos testes de integração já começamos por exemplo a fazer testes que exercitam uma instância local de banco de dados.

```
1  import server from '@server/app'
2  import supertest from 'supertest'
3
4  const app = () => supertest(server)
5
6  const user = {
7    name: 'test user',
8    email: 'test@mail.com',
9    password: 'password'
10 }
11
12 describe('The register process', () => {
13
14   it('Should register a new user', async () => {
15     const response = await app().post('/api/v1/auth/register').send(user)
16     expect(response.status).toBe(200)
17     expect(response.body.message).toBe('Account registered.')
18     expect(response.body.data.token).toBeDefined()
19   })
20 })
```

Nesse teste temos o seguinte:

³²<http://xunitpatterns.com/Test%20Double.html>

```
1 import server from '@server/app'
2 import supertest from 'supertest'
3
4 const app = () => supertest(server)
```

Nessa primeira parte temos o import do nosso app `import server from '@server/app'`, que é o arquivo da aplicação node onde estão todas as rotas de uma aplicação [express](#)³³. O import `supertest from 'supertest'` se refere ao import do [SuperTest](#)³⁴ que é a biblioteca que estamos utilizando para fazer os testes aqui e `const app = () => supertest(server)` onde servimos a nossa API para ser possível que tenhamos acesso ao endpoint que será testado.

```
1 const user = {
2   name: 'test user',
3   email: 'test@mail.com',
4   password: 'password'
5 }
```

Aqui temos uma constante chamada *user*, que iremos utilizar no nosso teste.

```
1 describe('The register process', () => {
2
3   it('Should register a new user', async () => {
4     const response = await app().post('/api/v1/auth/register').send(user)
5     expect(response.status).toBe(200)
6     expect(response.body.message).toBe('Account registered.')
7   })
8 })
```

Nessa última parte temos o teste em si, onde estamos acessando o endpoint de registro de usuários `/api/v1/auth/register` e enviando os dados através do nosso *user*. Depois disso temos as famosas asserções, representadas pelo `expect`, para conferir que a resposta foi um HTTP status 200, e a mensagem recebida foi `'Account registered.'`.

Lembrando que conforme incluímos mais componentes nos testes, a tendência é que eles demorem um pouco mais e que tenham mais pontos de falha. É um risco que devemos ter consciência de que assumimos e precisamos aprender a lidar com ele, já que testes isolados não conseguem por si só garantir todos os cenários necessários.

Testes de Contrato

O advento dos microsserviços trouxe esse tipo de teste pra um destaque posto que a comunicação entre esses serviços é um possível ponto de falha.

³³<https://expressjs.com/pt-br/>

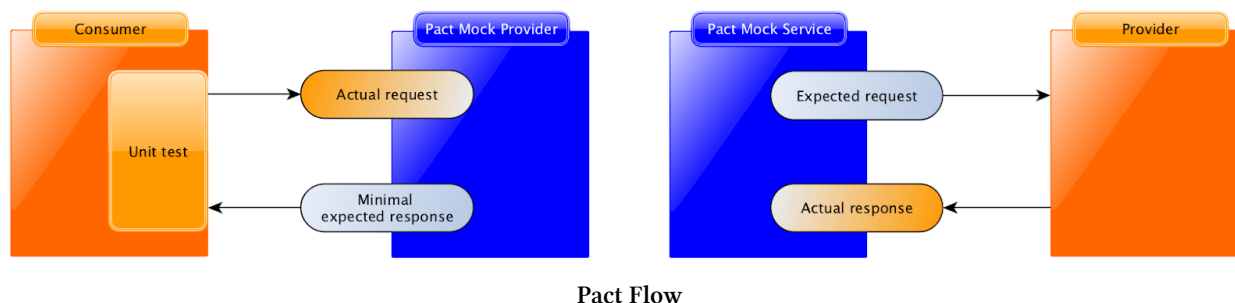
³⁴<https://github.com/visionmedia/supertest>

Imagine que você tem um serviço A que consome recursos de um serviço B. O serviço B tem um atributo chamado `email` que não é obrigatório e como esse atributo não é importante para o modelo de negócio do produto A, ele nunca passou esse atributo e nem pretende fazer isso. De repente o serviço B vê a necessidade de tornar o `email` obrigatório e como ele não tem visibilidade de quem são os seus consumidores, ele simplesmente sobe alteração para produção e a partir daí o serviço A passa a receber um erro 422³⁵ para TODAS as suas chamadas. Daí começa aquela saga que nós conhecemos: abre um incidente em produção, corre para ver o que aconteceu, identifica o problema e com sorte consegue com que o serviço B reverta a alteração até que isso seja melhor alinhado.

É nesse cenário que entram os testes de contrato orientado ao consumidor.

Nesse teste o consumidor, serviço A da nossa história, tem um contrato escrito especificando suas expectativas em relação ao serviço B e esse contrato é executado no momento dos testes. O pulo do gato aqui é que esses contratos também ficam disponíveis para o serviço B baixar e conferir se as suas mudanças não quebraram nenhuma expectativa. Desse jeito o serviço B além de conhecer todos os seus consumidores e como eles se comportam, também tem uma validação automatizada no seu próprio pipeline que vai impedir que novas mudanças sejam promovidas se algum desses contratos foi quebrado.

A ferramenta mais madura atualmente para esse tipo de teste é o Pact³⁶. Na imagem abaixo você consegue ver exatamente esse fluxo descrito:



Seja o seu serviço um consumidor ou provedor, é importante se preocupar com os contratos.

Testes Grandes

Existem alguns pontos não cobertos entre os testes que comentamos anteriormente:

- Se estamos usando dublês por exemplo, quem garante que aqueles dublês são fiéis a implementação real? E se o time esquecer de atualizar um dublê de um comportamento que foi alterado?
- Questões de configuração de ambiente, e se o time esquecer de configurar aquela variável na especificação do container? E se tiver um problema na conexão do container da aplicação com o banco de dados?
- Compatibilidade de plataformas

³⁵<https://httpstatuses.com/422>

³⁶<https://docs.pact.io/>

Apesar desses testes serem complementares aos anteriores, eles normalmente violam algumas propriedades, eles são mais lentos devido ao número de componentes envolvidos, eles costumam ser mais intermitentes e eles são mais difíceis de escalar, nem todo mundo consegue ter um ambiente de sandbox igual ao de produção por exemplo. Vamos então conhecer alguns desses testes.

Testes ponta-a-ponta

Os famosos teste ponta-a-ponta, assim como o nome já diz, são testes onde o comportamento do usuário é simulado o mais próximo possível do mundo real. Então se o seu projeto é uma API, seria um teste executado na infraestrutura o mais próxima de produção, passando por todos os componentes e algumas vezes até utilizando serviços externos.

Se o seu projeto tem interface web, seria um teste simulando todo o fluxo de abrir um browser, realizar ações e depois finalizar. Se é um aplicativo móvel, você precisaria instalar esse aplicativo, abrir, realizar as ações e depois desinstalar, você poderia usar um aparelho real ou um simulador por exemplo.

Além do fluxo propriamente dito que acabamos de falar, existe um passo anterior que é garantir que os dados que você precisa para esse teste realmente existem. Como por exemplo se você precisar estar logado para testar o envio de e-mail, você precisa garantir que existe um usuário com esse permissionamento para ser utilizado no teste, senão o seu teste já começa quebrando.

Apesar desses pontos esses testes também tem sua importância principalmente por serem mais fiéis ao comportamento do usuário final. Por isso é importante avaliar o nível de fidelidade que você quer para o seus testes. Eu sei que quando falamos de testes ponta-a-ponta logo vem a cabeça a imagem da pirâmide de testes invertida, mas esqueça isso por um momento e faça uma análise crítica do quanto esses testes são importantes para o seu produto.

Existem algumas formas de minimizar o impacto desses testes dentro do seu fluxo de entrega em produção, você pode por exemplo executar esses testes em paralelo, isso vai te fazer ganhar um pouco mais de tempo. Outra opção é criar suítes menores com um escopo mais definido, por exemplo se estamos falando de um sistema de pagamentos e temos os fluxos de transações e os fluxos de cadastro de novos clientes, você pode categorizar essas suítes e se você está fazendo uma modificação no fluxo de transações, talvez não seja necessário rodar todos os testes de cadastro.

Aqui você também pode mesclar uma estratégia de dublês, ao invés de fazer chamadas a integrações de terceiros que você não controla, você pode ter uma classe fake respondendo o que você precisa e exercitar os fluxos sem medo de receber uma resposta estranha de uma integração terceira. Você pode executar esses testes em um ambiente compartilhado com outras pessoas ou ser capaz de recriar o ambiente de teste toda vez que uma nova execução for iniciada, tendo assim mais controle desse ambiente e menos chances de enfrentar intermitências.

Uma dica aqui é: avalie os riscos envolvidos e decida a abordagem que traz mais segurança para o seu time!

Testes de Desempenho

Outro tipo de teste que está no grupo dos grandes são os testes de desempenho, esses testes normalmente são executados em um ambiente isolado e exercitam todos os componentes da infraestrutura mas você também pode avaliar a performance de pequenas unidades para identificar se houve degradação de performance entre uma versão e outra.

São testes com um foco em métricas, então não vamos olhar apenas se um registro foi criado e sim que quando eu crio 100 registros por segundo, eu tenho um determinado consumo de recursos ou até mesmo que meus recursos não tem capacidade suficiente para criar esses 100 registros.

Uma ferramenta muito famosa para esses testes é o [JMeter](https://jmeter.apache.org/)³⁷ mas hoje em dia existem várias outras *as a code* que facilitam a criação dos testes e a sua execução dentro de um pipeline.

```
1  scenarios:
2    - name: "Perform a search at Google"
3      flow:
4        - function: "generatingRandomSearchQuery"
5        - post:
6          headers:
7            X-RapidAPI-Host: "google-search3.p.rapidapi.com"
8            X-RapidAPI-Key: "{{ $processEnvironment.RAPID_API_KEY }}"
9          url: "/search"
10         json:
11           country: "US"
12           get_total: false
13           hl: "us"
14           language: "lang_en"
15           max_results: 100
16           q: "{{ random }}"
17           uule: ""
18         expect:
19           - statusCode: 200
20           - contentType: json
```

Esse é um exemplo de um script de teste do [Artillery](https://artillery.io/)³⁸. No meu [blog pessoal](https://medium.com/assertqualityassurance/testes-de-performance-com-artillery-e-datadog-2f2265134202)³⁹ tem um post contando como utilizá-lo integrado com o DataDog para captura das métricas.

Esse e outros testes não funcionais vão te ajudar muito a validar questões de configuração e infraestrutura, calibrar seu auto-scaling e encontrar gargalos antes que eles te surpreendam em produção.

³⁷<https://jmeter.apache.org/>

³⁸<https://artillery.io/>

³⁹<https://medium.com/assertqualityassurance/testes-de-performance-com-artillery-e-datadog-2f2265134202>

Testes de Compatibilidade

Quando falamos de aplicações que possuem a chamada interface de usuário, ou seja, uma aplicação web que acessamos via browser ou um aplicativo utilizado via smartphone, os testes de compatibilidade se tornam extremamente importantes já que conforme sua base de clientes vai crescendo fica impossível acompanhar manualmente os testes das diferentes versões em diferentes plataformas. No caso de aplicações web por exemplo, podemos acessar utilizando o Safari, Chrome, Brave, Firefox, Internet Explorer, Edge e quando falamos de mobile temos uma infinidade de marcas de aparelho como Samsung, Apple, LG, Nokia, Xiaomi, entre outras, além das versões de iOS e Android. Compatibilidade é garantir que sua aplicação funciona nos diferentes dispositivos que o seu usuário pode estar utilizando para acessá-la.

Existem serviços onde você consegue executar seus testes em diferentes browsers, sistemas operacionais e resoluções por exemplo, como a [Saucelabs](https://saucelabs.com/)⁴⁰ e o [BrowserStack](https://www.browserstack.com/)⁴¹. O mesmo pode ser feito para aplicativos Android e iOS, considerando diferentes versões e modelos de aparelhos.

Um outro tipo de teste onde você consegue garantir a compatibilidade é o teste de regressão visual, ferramentas como o [BackstopJs](https://backstopjs.com/)⁴² proporcionam isso.

Outras Verificações

Aqui temos algumas verificações bônus que vão te ajudar a elevar a barra de qualidade do seu projeto e garantir que as entregas em produção estão tinindo.

Análise Estática

A análise estática é uma prática que verifica a qualidade do seu código fonte. Essas verificações podem ser executadas antes mesmo do push através de um hook fazendo com que antes mesmo de enviar suas alterações você já fica sabendo se ofendeu alguma regra de estilo de código ou teve algum problema com a cobertura dos testes.

Uma das ferramentas mais famosas é o [SonarQube](https://sonarqube.org/)⁴³ que tem uma versão on-premise e cloud, nele você consegue observar algumas métricas que te ajudam inclusive a corrigir bugs e encontrar falhas de segurança antes que as alterações cheguem no cliente. Ele já tem alguns templates de boas práticas baseadas na linguagem, mas você pode configurar e incluir outras verificações como por exemplo [regras de segurança baseadas na OWASP](https://docs.sonarqube.org/latest/user-guide/security-rules/)⁴⁴.

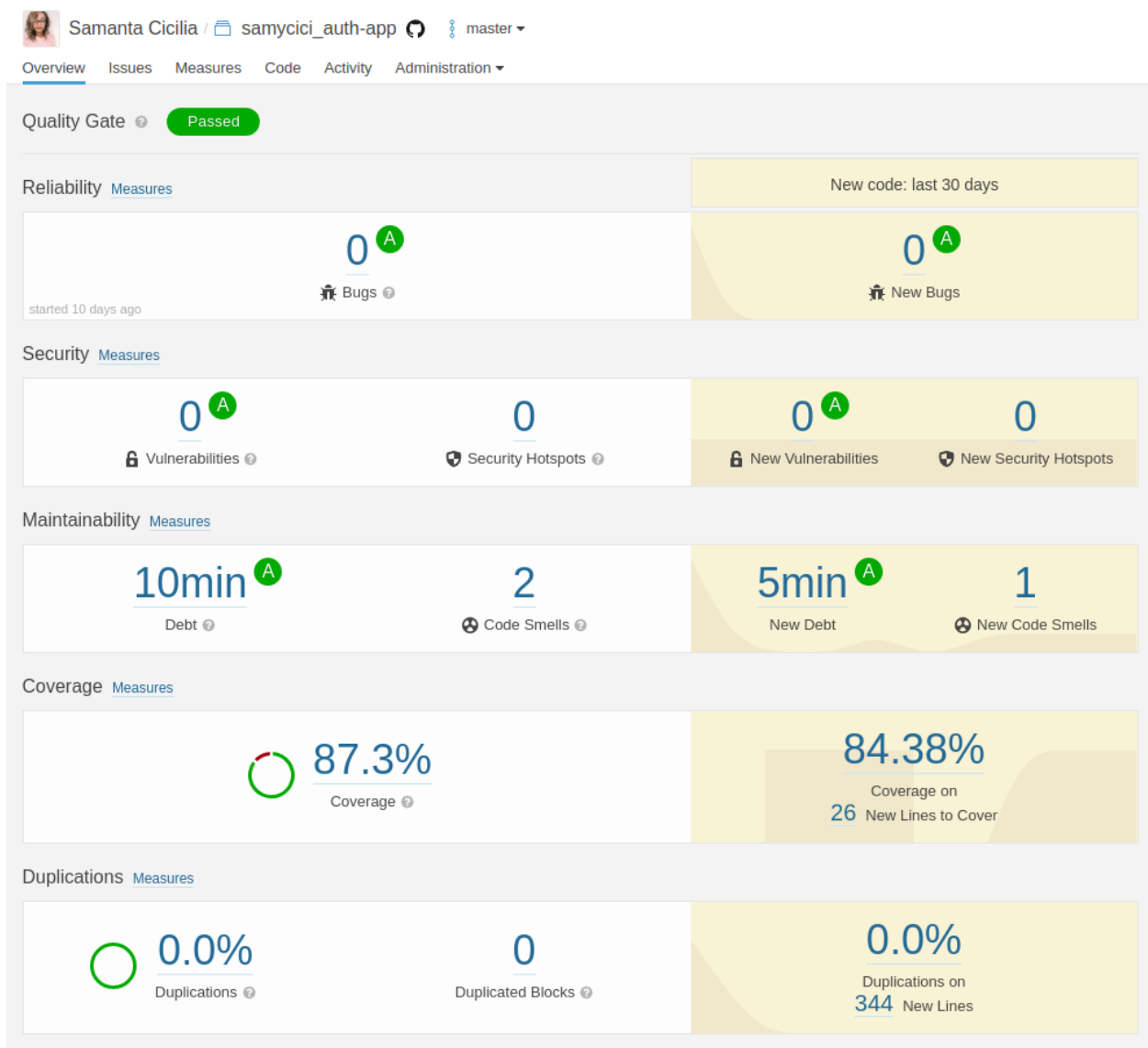
⁴⁰<https://saucelabs.com/>

⁴¹<https://www.browserstack.com/>

⁴²<https://github.com/garris/BackstopJS>

⁴³<https://www.sonarqube.org/>

⁴⁴<https://docs.sonarqube.org/latest/user-guide/security-rules/>



Sonar

A gente volta nesse assunto quando formos falar de Testes Contínuos e onde encaixar cada uma das verificações que falamos aqui. :)

Testes de Mutação

Por fim temos os testes de mutação. A ideia desse tipo de teste é validar a efetividade dos seus testes. A métrica de cobertura de testes por si só pode ser um número enganoso já que basta que algum teste exercite aquela linha de código que ela já é considerada coberta por testes, mesmo que não tenha nenhuma asserção.

Nos testes de mutação, alguns mutantes são inseridos em tempo de execução no código da aplicação e toda vez que uma alteração é realizada os testes são executados para verificar se vão quebrar. Se o

teste quebrar, significa que ele realmente está sendo efetivo posto que uma alteração foi introduzida no código, como por exemplo alterar uma condicional de `!request.authUser.emailConfirmedAt` para `request.authUser.emailConfirmedAt`, e o teste detectou isso como uma anomalia. Agora, se o código for alterado e o teste não quebrar, significa que tem um ponto cego ali que não está sendo testado.

Esses testes requerem bastante recurso computacional, o que inviabiliza executá-los em um pipeline de dia a dia, o que eu tenho feito é sempre executar esses testes na master e nas releases para validar a efetividade dos testes.

O exemplo abaixo é do [Stryker Mutator](https://stryker-mutator.io/)⁴⁵, uma ferramenta para execução de testes de mutação, ele também provê um dashboard para publicação dos resultados.

All files - auth-app/master - Stryker Dashboard

All files										
File / Directory	Mutation score	# Killed	# Survived	# Timeout	# No coverage	# Ignored	# Runtime errors	# Compile errors	Total detected	Total undetected
All files	69.12%	125	67	25	0	0	8	0	150	67
js config/index.js	23.41%	9	24	1	0	0	7	0	10	24
js controllers/auth.controller.js	70.83%	27	14	7	0	0	1	0	34	14
js middleware/auth.js	100.00%	11	0	0	0	0	0	0	11	0
models	60.87%	11	9	3	0	0	0	0	14	9
routes	50.00%	6	7	1	0	0	0	0	7	7
js validation-schemas/index.js	80.00%	4	1	0	0	0	0	0	4	1
validators	91.89%	55	6	13	0	0	0	0	68	6
js app.js	40.00%	2	3	0	0	0	0	0	2	3
js index.js	0.00%	0	3	0	0	0	0	0	0	3

Testes de Mutação

Importante lembrar que no caso dos testes de mutação não basta apenas executar, depois da execução é necessário analisar os resultados e planejar como aumentar a cobertura.

Caso adote o Stryker Mutator algumas estratégias adicionais podem ser utilizadas para reduzir o impacto do consumo de recurso computacional, como:

- Gerar os mutantes apenas nas linhas que possuem cobertura de código, ao invés de em todo o código.
- Executar esses testes em Pull Request apenas para os arquivos alterados. Em JS é possível com a biblioteca [Stryker-diff-runner](https://github.com/tverhoken/stryker-diff-runner)⁴⁶.

Conclusão

Testes são uma parte crucial do processo de entrega de software, para conseguir fazer entregas de qualidade em produção é imprescindível ter testes que tragam segurança para o time que caso exista algum problema nas alterações realizadas ao longo do ciclo de vida da aplicação, eles vão ser alertados o mais cedo possível e conseguir corrigir antes que esses problemas impactem um cliente.

⁴⁵<https://stryker-mutator.io/>

⁴⁶<https://github.com/tverhoken/stryker-diff-runner>

Reforçando a reflexão do início desse tópico sobre a questão da pirâmide de testes, é importante ler diferentes visões e entender o quê dessas visões se encaixa com a sua realidade. Isso se aplica a tudo que você leu aqui!! A parte de dublês de teste é um bom exemplo, existem vertentes onde as pessoas defendem com unhas e dentes a utilização deles e outras vertentes que acreditam que os testes tem que reproduzir o comportamento mais próximo da realidade possível. Não existe uma abordagem certa ou errada, existe uma abordagem que atende às necessidades da sua empresa, seu time, seu projeto e qualquer que seja a escolha tem que estar claro as consequências que vem com ela. Você pode optar por uma abordagem mockista e enfrentar problemas em pontos cegos de integração ou uma abordagem free mocks e acabar com uma suíte de testes que leva horas para executar e apresenta um comportamento instável.

Tudo tem ônus e bônus.

Agora que já definimos os testes, vamos entender como encaixá-los no seu processo automatizado de entrega.

Desafios

Agora que já entendemos a importância de ter testes para garantir nossas entregas em produção é importante falarmos um pouco sobre os desafios que ter esses testes automatizados podem trazer e algumas sugestões sobre como lidar com eles.

Testes Intermitentes

Como vimos anteriormente, temos diferentes tipos de teste e cada um deles abraça diferentes níveis dentro da nossa aplicação. Alguns estão apenas no escopo reduzido de uma função, enquanto outros exercitam diferentes componentes incluindo serviços externos e infraestrutura. Quanto mais integrados os testes são maior a chance de serem intermitentes. Testes intermitentes são aqueles que uma hora passam e na outra falham sem nenhum motivo aparente. É inevitável ter algum nível de intermitência posto que trabalhamos cada vez mais com aplicações complexas e cheias de integração, a diferença está na estratégia que adotamos para mitigar esse risco de intermitência. A primeira dica aqui é não testar serviços de terceiros dentro do seu fluxo de desenvolvimento, prefira mockar esses componentes porque caso esse serviço externo esteja indisponível em ambiente de teste, seu fluxo não será prejudicado. Testes intermitentes podem ser causados por componentes internos também, por exemplo se o seu teste utiliza dados de um teste anterior para seguir e esse teste anterior falhar ou demorar mais do que o normal, seu teste irá falhar por um motivo externo a ele mesmo.

Ao se deparar com um teste intermitente, a primeira coisa que você deve fazer é movê-lo para uma outra suíte, que normalmente chamamos de quarentena e ali analisar esse teste de forma isolada e sem atrapalhar o dia a dia do time.

Consumo de Recursos

Outro ponto importante quando falamos de testes médios e grandes é o consumo de recursos. Nem sempre você terá disponível um ambiente cópia de produção para executar os seus testes, seja pela

dificuldade de reprodução desse ambiente ou até mesmo pelo custo de mantê-lo. Para isso você pode adotar algumas estratégias que te auxiliem a ter um ambiente que atenda a todas as necessidades. Para componentes externos você pode utilizar um serviço como [WireMock](http://wiremock.org/)⁴⁷ para ser o servidor dublê desse componente externo. Você pode também utilizar um ambiente compartilhado com outros times (aqui tome cuidado porque essa decisão pode aumentar sua intermitência posto que existem várias pessoas manipulando o mesmo ambiente).

Gestão das falhas

Por último, é muito importante fazer a gestão dos testes que falharem e para isso é necessário ter visibilidade do que está acontecendo através de relatórios de execução de testes e fluxos automáticos que parem a “linha de produção” quando algum teste falha. Isso vai te ajudar a entender melhor qual foi o motivo da falha do teste, se é um caso de intermitência ou um bug mesmo.

Testes Contínuos

Depois de entender quais testes fazem sentido dentro do seu projeto, é hora de encaixar todos esses testes dentro da esteira de desenvolvimento sempre pensando em como viabilizar a produtividade, por isso executamos os testes o mais cedo possível dentro do ciclo para encontrar os problemas e já corrigi-los antes que impactem o cliente final. Esse é o famoso conceito de *shift-left* que ouvimos por aí, trazer o feedback que os testes nos dão para o nosso dia a dia.

Pré-Submit

Pré-submit é o momento antes de enviar as alterações para o repositório. Nesse momento devemos executar os testes que são mais rápidos e confiáveis, como os testes pequenos que vimos na seção anterior. Eles podem ser executados em segundos ou no máximo minutos e já vão informar se houver algum problema antes mesmo que isso vá para o repositório. Aqui podemos usar ferramentas específicas de cada linguagem para criar esse tipo de regra, se você está trabalhando com JavaScript, por exemplo, existe o [Husky](https://www.npmjs.com/package/husky)⁴⁸ que você pode configurar no seu package.json com os comandos que serão executados automaticamente quando você fizer o `git push`. No caso do JavaScript por exemplo, você configuraria para executar `npm run test`.

⁴⁷<http://wiremock.org/>

⁴⁸<https://www.npmjs.com/package/husky>

```
1 "husky": {  
2   "hooks": {  
3     "pre-push": "npm run test",  
4     "...": "..."  
5   }  
6 },
```

Você pode conferir o exemplo completo neste [repositório](#)⁴⁹.

Pós-Submit

Depois que o código é enviado para o repositório, partimos para o processo de integrar esse novo código no código já existente e garantir que tudo continua funcionando, aqui já executamos os testes médios. Se tudo correr bem, temos uma versão candidata que pode ser lançada em ambiente de teste. Esse é o primeiro passo do seu pipeline.

Versão Candidata

Como resultado do passo anterior você terá sua versão candidata, ou seja, um pacote com o código já alterado e que passou por todos os testes. Depois que temos uma versão candidata já publicada em algum ambiente de testes é a hora de executar os testes grandes para validar que os fluxos de negócio estão funcionando de forma integrada com todos os componentes envolvidos. Aqui você pode ter de 1 a N ambientes de teste, algumas empresas tem o ambiente de staging, pré-prod, homolog, os nomes são variados, mas o importante é ter ambientes de teste onde a sua versão será publicada e testada antes de prosseguir para produção.

Depois dessa validação finalmente podemos enviar a nossa mudança para ambiente de produção. ☒

Produção

Chegando em produção nós podemos reaproveitar os testes grandes e executar os que sejam mais relevantes para garantir que os fluxos principais continuam funcionando, além de se aproveitar das informações que recebemos dos monitoramentos (teremos uma seção dedicada a falar sobre isso nos próximos capítulos) para continuar observando que a mudança introduzida não causou nenhum problema.

Como usar a estratégia de testes para tomada de decisão

Sua estratégia de testes vai ser a grande aliada na tomada de decisão desde a chegada das demandas até o resultado final na mão do cliente. Com a análise estática e os testes de mutação você terá

⁴⁹<https://github.com/samycici/auth-app/blob/master/package.json#L59-L64>

uma visibilidade melhor da qualidade das entregas e isso poderá ser usado de entrada para criação de tarefas que tenham por objetivo aumentar a confiança nos testes e garantir que as partes mais críticas estão sendo bem testadas.

Durante o fluxo de desenvolvimento, o resultado dos testes vai te ajudar a entender se você pode prosseguir e lançar uma nova versão por exemplo, ou se existe algum problema que necessita de ação imediata. E após entregar uma nova funcionalidade para o cliente, você será capaz de avaliar se realmente a sua estratégia está trazendo resultados (software de qualidade em produção) ou se os clientes continuam enfrentando problemas e você precisa entender quais os pontos dentro do processo que precisam ser modificados para endereçar isso.

Conclusão

Uma estratégia de testes que traga confiança para que você introduza mudanças no código requer a utilização de diferentes tipos de teste, testes pequenos, testes médios e testes grandes, onde cada um vai ser responsável pela cobertura de diferentes componentes da sua aplicação, desde as pequenas funções até o fluxo completo de uma compra on-line por exemplo. Quanto mais complexos os testes forem em termos de componentes envolvidos, mais recursos serão necessários para executá-los e mais tempo também.

Apesar dos desafios envolvidos é um fato que testes são indispensáveis em desenvolvimento de software moderno e são a peça chave para fazer entregas em produção com segurança e qualidade.

Prefira ter uma cobertura maior em testes pequenos e médios onde a complexidade de componentes é menor e seus testes serão mais assertivos e menos intermitentes. Mas isso não significa não ter testes grandes, eles também tem sua importância em cobrir pontos que não são abraçados pelos tipos anteriores, aqui prefira validar os comportamentos mais focados na jornada do usuário.

E de nada adianta ter todos esses testes mas não executá-los de forma automatizada no seu pipeline. Quanto mais cedo eles forem executados mais cedo eles te darão o feedback de como a mudança feita afetou o código existente e mais cedo você consegue investigar e resolver os problemas.

A última dica é se preocupar com o design dos seus testes e não apenas em automatizá-los, os testes devem ser úteis e encontrar problemas caso eles apareçam, pensar em testes desde o início do projeto faz com que a aplicação já nasça com cultura de testes.

Migrações em Banco de Dados Relacionais (Daniane Pereira Gomes)

Introdução

Frequentemente o deploy em produção de uma aplicação envolve não só a publicação dos arquivos que compõem a aplicação (HTML, CSS, JavaScript e outros), mas também alterações na estrutura de banco de dados relacionais.

Quando tais alterações envolvem comandos DDL (Data Definition Language), por exemplo, criação, alteração e remoção de tabelas ou de campos em tabelas tais mudanças podem resultar em tempo de indisponibilidade para a aplicação.

A situação torna-se ainda mais crítica em arquiteturas distribuídas ou de micro-serviços. Imagine a seguinte situação: “*Pedido*” e “*Produto*” são sistemas separados e o serviço “*Pedido*” acessa o serviço “*Produto*” para ler o atributo `quantidade_em_estoque` da tabela `produto`. O diagrama a seguir ilustra a comunicação entre os sistemas de “*Pedido*” e “*Produto*”.

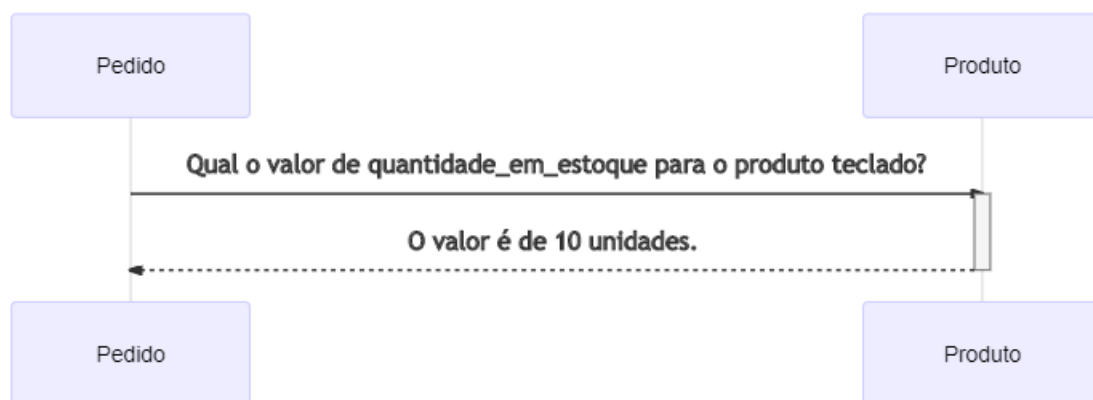


Diagrama de sequência para demonstrar a comunicação entre “*Pedido*” e “*Produto*”.

Consideraremos a tabela com a seguinte estrutura:

```

1 CREATE TABLE `produto` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
3   `nome` varchar(100) DEFAULT NULL,
4   `descricao` varchar(2000) DEFAULT NULL,
5   `quantidade_em_estoque` int(11) NOT NULL,
6   PRIMARY KEY (`id`)
7 );

```

O código acima é responsável por criar um tabela de banco de dados, com quatro colunas, sendo a com o nome `id`, que não pode receber valores nulos e tem os valores incrementados automaticamente pelo banco de dados. As colunas `nome` e `descricao` podem receber valores alfanuméricos e valores nulos num máximo de 100 e 2000 caracteres respectivamente. A coluna `quantidade_em_estoque` pode receber valores numéricos e inteiros até o máximo de 11 algarismos e não aceita valores nulos. A instrução `PRIMARY KEY (`id`)` define que a coluna `id` é a chave primária da tabela, ou seja, um campo único para cada registro que permite identificá-lo e diferenciá-lo dos demais.

A seguir é possível conferir uma representação visual da tabela `produto` seguindo a notação dos diagramas de Entidade Relacionamento.

PRODUTO	
PK	<u>id: int(11) NOT NULL AUTO_INCREMENT</u>
	nome: varchar(100) DEFAULT NULL descricao: varchar(2000) DEFAULT NULL quantidade_em_estoque: int(11) NOT NULL

Representação visual da tabela de produtos em notação de diagrama Entidade Relacionamento

Agora imaginamos que a equipe decidiu que não faz sentido a existência do atributo `quantidade_em_estoque` em `produto` e deseja movê-lo para outra tabela. O atributo é apagado da base de dados e o deploy do serviço “*Produto*” é realizado com sucesso.

Entretanto, a equipe que mantém o serviço “*Pedido*” ainda não fez a alteração na leitura e deploy em produção. Vamos supor que a equipe só conseguirá completar as alterações no código no mês seguinte e nesse momento continua a tentar ler o atributo `quantidade_em_estoque`. Qual o resultado disso para o processo como um todo? Erros e indisponibilidade acontecerão no serviços de “*Pedido*”.

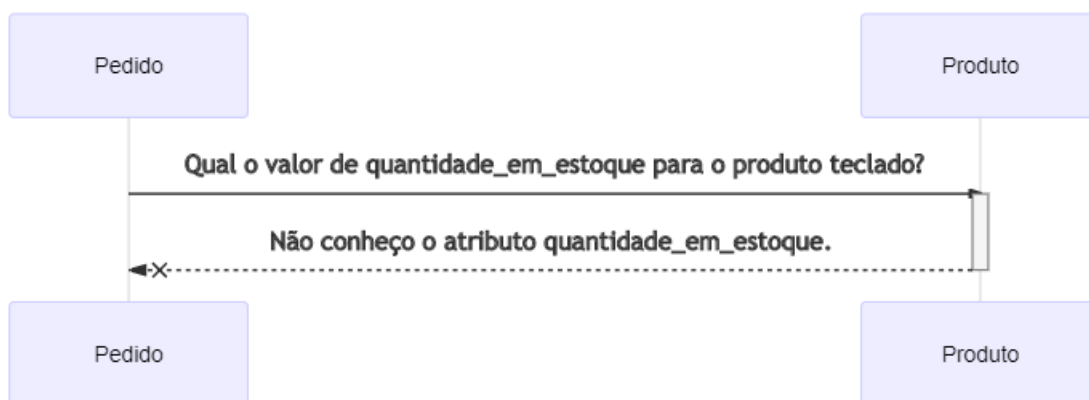


Diagrama de sequência para demonstrar a falha na comunicação entre “*Pedido*” e “*Produto*” quando o campo “*quantidade_em_estoque*” não existe.

O diagrama acima demonstra a interação entre “*Pedido*” e “*Produto*”, onde “*Pedido*” solicita o valor de *quantidade_em_estoque* mas “*Produto*” retorna um erro pois desconhece tal campo.

Imagine também que o campo *descricao* da tabela *produto* deve tornar-se obrigatório. A pessoa responsável pela alteração escreve o *script* com alteração da estrutura da tabela modificando o campo para *not null*. Se tudo correr bem a pipeline identifica a mudança, aplica o *script* automaticamente e o *schema* é alterado.

Quais seriam os possíveis problemas nessa abordagem? Se a tabela já possuir registros e valores nulos, como é o caso do nosso sistema de “*Produto*”, tal comando apresentará erros.

Os exemplos citados tratam-se de **mudanças destrutivas** e devem ser tratados cuidadosamente principalmente em ambientes de produção.

Mas então nunca deve-se fazer alterações em banco de dados de sistemas em produção? E se for realmente necessário apagar colunas de tabelas?

É óbvio que um sistema precisa de mudanças e evoluções e deixar de alterar tabelas no banco de dados não é uma opção. Para tratar disso com segurança, contamos com orientações e práticas de mercado como, por exemplo, as sugeridas por [Pramod Sadalage](http://www.sadalage.com/)⁵⁰ e [Martin Fowler](https://martinfowler.com/)⁵¹ no artigo “*Evolutionary Database Design*”⁵².

Boas Práticas em Migrações

Toda alteração de banco de dados é tratada como uma migração e significa que foram desenvolvidos um ou mais *scripts* que efetuarão alterações no banco de dados, o que resultará na alteração do *schema* da aplicação.

O *schema* é espaço lógico dentro de um banco de dados que agrupa objetos como tabelas, chaves primárias, chaves estrangeiras, views, etc. É uma estrutura criada para guardar e resgatar dados de

⁵⁰<http://www.sadalage.com/>

⁵¹<https://martinfowler.com/>

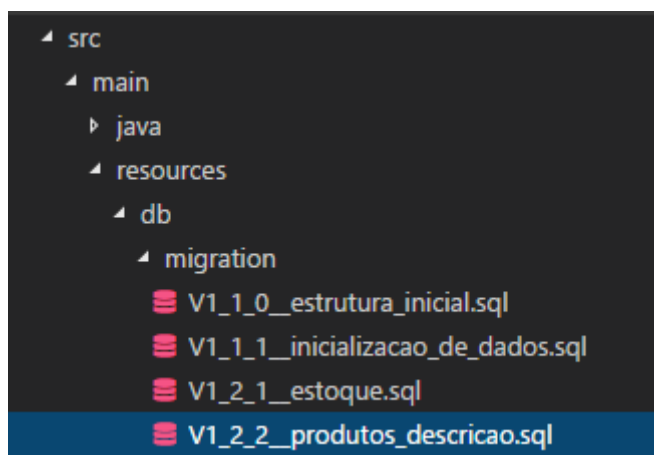
⁵²<https://martinfowler.com/articles/evodb.html>

forma mais eficaz. Tal estrutura permite a organização e agrupamento das informações de forma que estas façam sentido para a aplicação. O *schema* define uma espécie de “molde” que será preenchido com dados. Um mesmo banco de dados pode conter vários *schemas*

Para que as alterações ocorram sem impactos destrutivos, algumas boas práticas serão abordadas a seguir.

Versionamento de *schema*

O *schema* da aplicação deve estar junto do código fonte e demais artefatos de *software*, como testes e outros. As alterações devem fazer uso de alguma ferramenta de controle de versões como o [Git](#)⁵³, para que se mantenha o rastreio de banco de dados e sua relação com o código fonte ao longo do tempo.



Exemplo pasta com migrações

Colaboração e revisão entre a equipe

Para minimizar a possibilidade de erros, é importante que a equipe mantenha a colaboração. Ao fazer uso do Git, as alterações são submetidas à avaliação de mais membros do time antes de ser efetivamente incorporada ao banco de dados. Nesse estágio também pode haver a revisão dos scripts pela *DBA* (Database Administrator) caso exista alguém a desempenhar esse papel. Alterações destrutivas podem ser detectadas e impedidas de acontecer. O capítulo “O que é Pull Request? (Rafael Gomes)⁵⁴” explica detalhadamente o processo de revisão.

Cópias locais e reintegrações

Cada pessoa desenvolvedora deve fazer uso de uma instância de banco de dados próprio, o qual deve ser constantemente reintegrado com a versão oficial. O uso de cópias locais permite que seja feito o desenvolvimento de requisitos sem a necessidade de atualização um servidor compartilhado.

⁵³<https://git-scm.com/>

⁵⁴[o_que_e_pr.md](#)

Imagine que você deseja criar uma nova coluna *categoria* na tabela *produto* que não permite nulos. Se o seu código fonte que trata dessa nova coluna ainda não foi enviado para uma *branch* compartilhada, todas as demais usuárias desse banco de dados terão erros ao tentar inserir novos produtos.

Devido a cópias locais não demandarem que cada alteração seja enviada para um servidor compartilhado, a prática evita que alterações ainda não completamente testadas impactem e quebrem o ambiente de um time inteiro.

Tamanho de migrações

Assim como é aconselhado que os “*Pull Requests*” sejam pequenos, as migrações também devem ser. Pequenas e constantes integrações tendem a causar menos problemas do que alterações grandes e esporádicas.

Vamos considerar o *script* a seguir como um arquivo de migração nomeado “*V1_2_3__alteracoes_produto_e_estoque.sql*”.

```
1 ALTER TABLE produto MODIFY descricao NOT NULL DEFAULT 'Aguardando descrição';
2
3 ALTER TABLE estoque MODIFY quantidade float(10,2) DEFAULT NUL;
```

O código acima altera a coluna *descricao* da tabela *produto* para não permitir valor nulos e usar o valor padrão “Aguardando descrição”. Também altera o campo *quantidade* da tabela *estoque* para aceitar valores decimais com o máximo de 10 algarismos antes da vírgula e 2 algarismos depois da vírgula.

Imagine agora que esta migração apresentou erros ao ser executada no banco de dados. Nesse caso, seria necessário a intervenção humana para depurar o *script* e verificar qual instrução precisamente não pode ser executada. O exemplo citado possui somente duas instruções e neste caso pode até ser simples encontrar qual delas é problemática. Porém, quanto maior a migração mais difícil de depurar o problema e revertê-lo.

O arquivo “*V1_2_3__alteracoes_produto_e_estoque.sql*” então poderia ser dividido em duas partes, como exemplificado a seguir.

```
1 ALTER TABLE produto MODIFY descricao NOT NULL DEFAULT 'Aguardando descrição';
```

Arquivo “*V1_2_3__produto_descricao_not_null.sql*” trata somente das alterações na tabela de *produto*.

```
1 ALTER TABLE estoque MODIFY quantidade float(10,2) DEFAULT NUL;
```

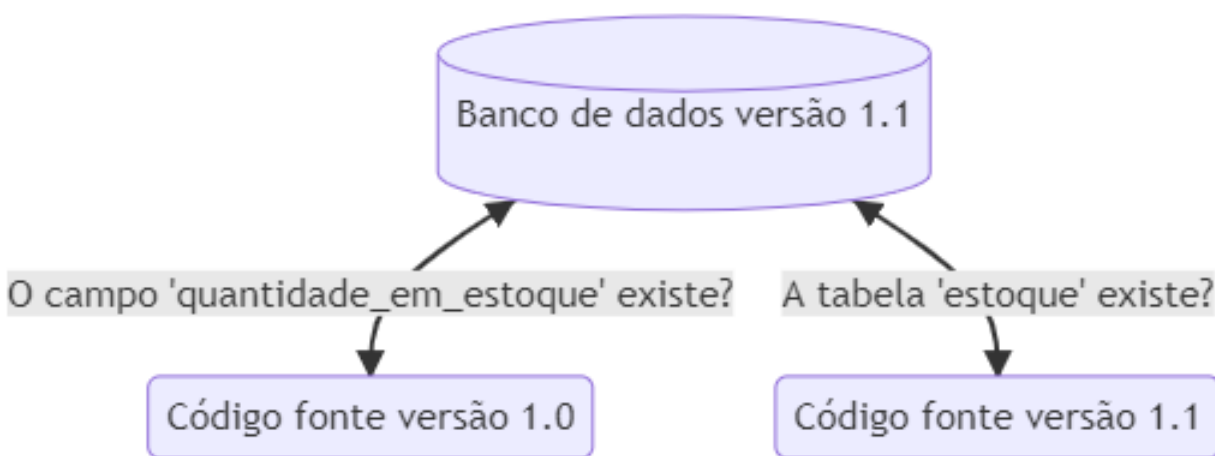
Arquivo “*V1_2_4__estoque_quantidade_float.sql*” trata somente das alterações na tabela de *estoque*.

Compatibilidade retroativa

E como evitar migrações destrutivas como as citadas nos exemplos de “*Pedido*” e “*Produto*”?

Para que uma migração não seja destrutiva é necessário que ela possua compatibilidade retroativa, ou seja: a alteração feita na versão atual não pode fazer a versão anterior do código fonte deixar de funcionar.

Se o *script* for aplicado no banco de dados de produção e o código fonte dos serviços que o acessam não for alterado, essa mudança não pode fazer com que o processo de ponta a ponta apresente erros.



Exemplo de interação entre os códigos fonte nas versões 1.0 e 1.1 com a versão 1.1 do banco de dados.

O diagrama acima ilustra o processo. Podemos notar que o banco de dados está na versão 1.1. Se iniciarmos um serviço com o código fonte da versão 1.0 com o banco de dados na versão 1.1, o sistema irá buscar pelo atributo `quantidade_em_estoque` e deve receber uma resposta de sucesso. Já ao iniciar um serviço com o código fonte na versão 1.1, o sistema irá buscar pela tabela `estoque` na versão 1.1 do banco de dados e o processo deve igualmente acontecer sem problemas. Isso se dá devido à compatibilidade retroativa de versões.

Além de permitir que sistemas externos continuem a funcionar e tenham tempo para fazer alterações, a compatibilidade retroativa é útil se for necessário executar o *rollback* de algum deploy.

Por exemplo: numa sexta-feira às 17 horas a versão 1.1 de “*Produto*” foi liberada em produção e as migrações no banco de dados foram aplicadas. Porém, após alguns minutos de uso, os usuários começam a reportar que não conseguem criar novos produtos. A equipe decide então voltar o sistema para a versão 1.0, que não apresentava erros e assim todos podem ir para casa e retomar a investigação na segunda-feira com mais tempo.

O banco de dados já está na versão 1.1, pois já foram executados os *scripts* de migração. Agora imagine que a migração era destrutiva e simplesmente apagava o atributo `quantidade_em_estoque`. O que vai acontecer ao “subir” o serviço com o código fonte da versão 1.0? O serviço simplesmente não irá funcionar e a responsável pelo deploy precisará verificar logs, etc e investigar o porquê do problema. Se detectar a causa, ainda precisará escrever e aplicar manualmente um novo *script* para

recriação do atributo que foi apagado. Além de alguém perder seu tempo de descanso para verificar algo que muito provavelmente não foi ela que causou, o sistema terá grande *downtime*, coisa que algumas aplicações não podem ter.

O cuidado com a compatibilidade retroativa previne stress, *downtime* e perdas irreversíveis de dados.

Remover campo em tabela

Como ficaria então nosso exemplo de remoção do atributo `quantidade_em_estoque`?

Tal alteração necessitaria passar por 3 etapas:

1) Início das alterações

É feito o *script* inicial com a criação da nova tabela. Exemplo:

```
1 CREATE TABLE `estoque` (  
2   `id` int(11) NOT NULL AUTO_INCREMENT,  
3   `produto_id` int(11) DEFAULT NULL,  
4   `quantidade` int(11) DEFAULT NULL,  
5   PRIMARY KEY (`id`)  
6 );  
7  
8 ALTER TABLE estoque ADD CONSTRAINT fk_estoque_produto  
9   FOREIGN KEY (produto_id) REFERENCES produto(id);
```

O código acima cria a tabela com nome `estoque` com três colunas. A primeira, nomeada `id`, aceita números inteiros até o máximo de 11 algarismos, não aceita valores nulos e é auto incrementada pelo banco de dados. As colunas `produto_id` e `quantidade`, aceitam números inteiros até o máximo de 11 algarismos e valores nulos.

A instrução `PRIMARY KEY (`id`)` define que a coluna `id` é a chave primária da tabela, ou seja, um campo único para cada registro que permite identificá-lo e diferenciá-lo dos demais.

O comando iniciado com `ALTER TABLE` altera a tabela recém criada e adiciona uma restrição (`ADD CONSTRAINT`) como nome `fk_estoque_produto`. Trata-se de uma chave estrangeira `FOREIGN KEY` para a coluna `produto_id` que faz referência à tabela `produto`, coluna `id` (`REFERENCES produto(id)`). Tal ligação fará com que só seja possível informar em `produto_id` valores existentes em `id` na tabela `produto`.

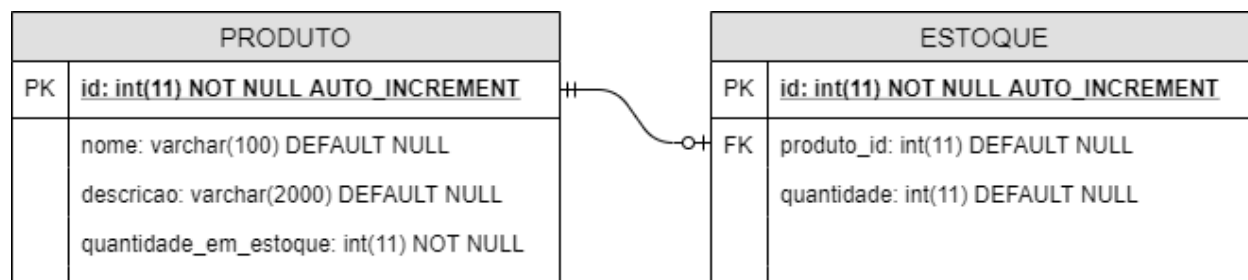


Diagrama Entidade Relacionamento entre as tabelas de produto e de estoque

O diagrama de Entidade Relacionamento acima demonstra a interação das tabelas *produto* e *estoque* no banco de dados. O *script* que acabamos de analisar demonstra a tabela *estoque* representada pelo retângulo da direita. A chave estrangeira é demonstrada através da linha de ligação entre a tabela pré-existente *produto*, representada pelo retângulo da esquerda.

2) Período de transição

Durante o período de transição, tanto o campo *quantidade_em_estoque* quanto as informações em *estoque* devem ser mantidas e alimentadas. Ou seja, se o produto “Teclado” tinha 10 unidades e 1 foi vendido, *quantidade_em_estoque* e o teclado correspondente na tabela *estoque* deverão ser atualizados para o valor 9.

Da mesma forma, quando o sistema de “*Pedido*” (ou outro sistema externo) fizer uma chamada a “*Produto*” para saber o número de teclados disponível, a resposta deve ser enviada sem erros tanto quando a requisição for feita para o campo *quantidade_em_estoque* quanto para a nova tabela *estoque*.

A definição do tempo de duração da transição deve ser definida pela equipe. Quanto tempo é necessário para que todos os sistemas que “conversam” com *Produto* sejam alterados, testados e liberados em produção? Essa pergunta pode ser um bom ponto de partida para a definição da duração do período transitório.

3) Finalização

Passado o período de transição, o campo *quantidade_em_estoque* pode ser finalmente removido, bem como o código que o mantinha.

O mesmo processo pode ser adotado para demais alterações destrutivas. Parece muito trabalho para uma alteração simples? Realmente, esse processo torna as coisas mais complexas, mas não segui-lo pode causar tempo de indisponibilidade do sistema.

O seu cliente pode ter o sistema parado? Quantos pedidos deixarão de ser feitos e quanto dinheiro será perdido caso a API falhar? Esses são pontos a serem considerados.

Alterar campo para `not null`

A instrução `not null` simplesmente define que determinado campo nunca pode ficar sem valores informados, ou seja, não pode aceitar nulos. Uma alteração que parece tão simples precisa mesmo passar por todo três estágios de início, transição e fim?

Para algumas alterações, como essa por exemplo, a equipe pode decidir não aplicar os três estágios e fazer a alteração em um único *script* de migração. Porém para que este não seja destrutivo, é importante lembrar de aplicar “*update*” com valor padrão para possíveis nulos e só depois alterar a estrutura tabela. Deve-se ainda informar um valor padrão (“*default*”) no banco de dados para que a migração não seja destrutiva. O *script* é exemplificado a seguir.

```
1 UPDATE produto SET descricao = 'Aguardando descrição' WHERE descricao IS NULL;  
2  
3 ALTER TABLE produto MODIFY descricao NOT NULL DEFAULT 'Aguardando descrição';
```

O código acima altera todos os registros da tabela `produto`, preenchendo o campo `descricao` com o valor “*Aguardando descrição*” em todos os registros que possuem `descricao` nula.

Após, altera a estrutura da tabela `produto`, modificando o campo `descricao` para não aceitar valores nulos. Quando a descrição não for informada, o banco de dados preencherá automaticamente o campo com a informação “*Aguardando descrição*”.

Conclusão

O processo descrito gerará uma quantidade significativa de trabalho ao desenvolvimento. Apesar de ser uma forma segura de tratar a evolução de bancos de dados relacionais, a equipe deve discutir e entrar em um acordo se essa carga extra faz realmente sentido no contexto do negócio.

Talvez para monólitos, aplicações pequenas ou que podem enfrentar períodos de indisponibilidade as etapas de transição não façam sentido. Não há certo e errado no processo de desenvolvimento. Existem sim boas práticas e orientações de mercado sobre como lidar com problemas conhecidos, mas a adoção parcial, completa ou a não adoção fica a cargo do time.

O que deve ter no seu pipeline?

(Rafael Gomes)

Introdução

Antes de “colocar a mão na massa” e iniciar o processo de construção do seu pipeline, você precisa entender qual problema você está tentando resolver, pois toda intervenção na computação tem (ou deveria ter) como objetivo resolver algum problema, correto? Mesmo que o problema seja otimização, por conta de performance, ou trabalho proativo para que não exista problema no futuro.

Quando você inicia a construção de um pipeline, normalmente, seu objetivo é entregar um produto. Seja ele de software ou infraestrutura.

Se a solução do problema aqui é entregar o produto de forma automatizada, você precisa entender quais são os passos que seu produto precisa seguir para ser colocado em produção.

A ordem importa?

Antes de apresentar os passos, precisamos primeiro entender que a ordem das etapas do pipeline importam **e muito**, sendo assim apresentarei as etapas aqui na ordem que elas devem estar no seu pipeline.

Por que a ordem importa?

Uma das vantagens de usar pipeline no processo de entregar de software é a ideia dele “economizar” tempo das pessoas que estão produzindo código, então o ideal é que as tarefas que demoram menos, entregam algum valor e não dependem de outros passos posteriores sejam as primeiras no seu pipeline.

Vamos usar um exemplo abstrato. No processo de entrega de um software hipotético, temos os seguintes passos:

- Build do artefato
- Teste unitário
- Provisionamento da infra pré-produção
- Teste de integração
- Teste de aceitação
- Provisionamento de infra produção
- Deploy de pré-produção
- Deploy de produção

Na sua opinião, qual seria o primeiro? Vamos analisar alguns dos candidatos:

Build do artefato depende de outro passo? Não. Ele entrega valor? Entrega sim, pois se o build quebrar, a pessoa que está desenvolvendo saberá que tem problemas para fazer build, mas esse processo de build costuma demorar demasiadamente e isso pode fazer com que o feedback seja demorado. Vamos imaginar juntos: A pessoa manda o commit para o repositório, o pipeline automaticamente é executado e depois de alguns longos minutos a pessoa que mandou o commit poderá descobrir que errou, pois a etapa de build vai executar a construção do artefato e assim pegará qualquer problema que apareça nesse processo. Muitas vezes um detalhe bobo pode levar a quebrar o pipeline nessa etapa.

E se pensarmos no **teste unitário**? Depende de outro passo? Não. Ele entrega valor? Entrega sim, e aqui temos um detalhe diferente do **build do artefato**, pois o retorno é mais rápido, uma vez que, normalmente, nada precisa ser realmente construído. Por padrão os testes unitários demoram menos do que o build dos artefatos. Voltando ao processo de imaginação: A pessoa manda o commit para o repositório, o pipeline automaticamente é executado e depois de **segundos** ela já terá um feedback que um determinado teste não está passando. Tudo por culpa daquele “detalhe” bobo que falamos anteriormente.

Seguindo essa lógica, o primeiro passo desse pipeline seria o **teste unitário**, pois não há nada que demore menos e ainda assim não dependa de outro passo. Vejam que são sempre múltiplos fatores para determinar a ordem e em minha opinião são normalmente esses:

- Dependência de outro passo
- Entrega de valor
- Tempo de feedback

Quando falamos de entrega de valor, a preocupação é com o processo de desenvolvimento e não apenas com o produto finalizado. Para o produto finalizando o build talvez seja mais importante do que o teste unitário, pois levando em consideração o processo de desenvolvimento eles tem importâncias bem próximas.

Garantindo a qualidade para sua Infra como código (Rafael Gomes)

Introdução

A infra como código ainda é um assunto em desenvolvimento, e ao contrário da engenharia de software no desenvolvimento convencional de código, que está bastante estabelecida, na automação de infra as melhores práticas ainda estão em amplo debate e pouco adotadas pela maioria das equipes que fazem esse tipo de implementação.

É muito comum em infra as code (IaC) não existir muita preocupação com a qualidade do código escrito, seja porque, normalmente, eles são feitos por pessoas com pouca experiência em desenvolvimento em geral, ou talvez por existir poucos materiais falando sobre isso.

Enviar o código sem uma validação para que o pipeline “tente” aplicar seu código de infra pode ser uma atitude catastrófica. O entendimento da infraestrutura é quase sempre limitado e raramente temos uma compreensão completa do que está sendo executado em nosso ambiente de produção. Tudo bem assumir isso, ok?

O que leva a pessoa que trabalha com infra a cometer esse tipo de atitude? Será que existe outra possibilidade?

Mostraremos alguns caminhos do que pode ser feito para evitar esse tipo de situação, afinal não é uma boa prática essa atitude de “tentativa e erro”. Como estamos falando de infraestrutura, uma tentativa muito equivocada pode causar danos com grandes chances de lhe oferecer novos desafios por algumas horas ou até mesmo dias.

O que seria QA para IaC?

Quality Assurance (QA) é a disciplina que trata sobre prevenir erros ou falhas na entrega de software. É o estudo e prática do que deve ser feito para oferecer alguma garantia antes do código ser compartilhado com todo time, pois todo mundo erra, mas o mais importante é que esse erro seja visualizado o mais rápido possível, evitando assim que ele cause problemas para outros membros da sua equipe ou até mesmo para a pessoa que usará seu software após entregue em produção.

QA pode e deve ser usado para códigos de infra, mas temos pouca literatura sugerindo esse tipo de prática, infelizmente.

Esse texto terá um enfoque especial na ferramenta de gestão de configuração chamado [Ansible](https://docs.ansible.com/ansible/latest/index.html)⁵⁵, pois é um dos softwares de IaC mais usado na comunidade.

⁵⁵<https://docs.ansible.com/ansible/latest/index.html>

Será apresentado formas de como validar o seu código antes dele ser compartilhado com outras pessoas do seu time e como garantir que o pipeline identifique rapidamente o erro no código que você ou outra pessoa do seu time compartilhou sem perceber que tinha problemas.

Testando seu código antes do commit

Pensando no Ansible, o teste não se reduz ao comando **ansible-playbook** executar com sucesso. A execução correta do **ansible-playbook** é um dos passos necessários, mas não o único.

A primeira checagem que seu código precisa passar é o que chamamos de **lint**.

Lint é uma checagem simples e estática do seu código, que é um binário que acessa todos seus arquivos IaC e procura por falhas na escrita do código, como uma indentação incorreta, um espaço em branco desnecessário, uma linha que não precisa ou um ponto e vírgula necessário. Ele busca erros simples e executa muito rápido. Esse é talvez o teste mais rápido que você poderá executar no seu código IaC.

Na checagem estática o código não precisa ser implementado em nenhum servidor, pois ele só olha o conteúdo do código e não seu resultado.

Para o Ansible, temos o **yamllint**⁵⁶ que pode ser instalado com o comando abaixo:

```
1 pip3 install yamllint
```

Para testar usando essas ferramentas é muito simples:

```
1 yamllint .
```

Eu aconselho muito a criação do arquivo **.yamllint**⁵⁷ na raiz do seu repositório para configurar o uso correto do lint.

Testes automatizados para Ansible

Agora que você já sabe validar estaticamente o conteúdo do seu código Ansible, podemos passar para validação automatizada do comportamento do seu código, que é a escrita de um código que tem como objetivo validar se o produto final entregue pelo seu IaC de fato faz o que você espera que ele faça.

Usando um exemplo mais prático, imagine que você tem uma role Ansible que tem como objetivo instalar e configurar um servidor web. Considere que, alguém precisou alterar o código para adicionar a feature de configurar certificados para oferecer uma conexão HTTPS. Porém, a alteração

⁵⁶<https://yamllint.readthedocs.io/en/stable>

⁵⁷<https://yamllint.readthedocs.io/en/stable/configuration.html>

de código quebrou o arquivo de configuração e, seu servidor web, uma vez configurado com essa role, não se sustentará executando no servidor. Pois, o mesmo está com erro no arquivo de configuração.

A execução do Ansible, nesse caso, não falhará, muito menos o lint apontará qualquer problema. Você só perceberá qualquer problema após aplicar o código Ansible em um servidor e então acessar o servidor manualmente tanto na porta HTTP como HTTPS.

Para evitar situações como essas, deve ser usado testes automatizados. Uma boa ferramenta para testar códigos de IaC é o [Testinfra](#)⁵⁸.

Para o utilizar o Testinfra, primeiro você deve executar o seguinte comando para instalar:

```
1 pip install pytest-testinfra
```

Após instalado, você deve criar um arquivo de teste.

Esse arquivo deve ser python, e o nome desse arquivo deve começar com “test_”, ou seja, seu arquivo pode ser “test_http.py” por exemplo.

Para o exemplo citado acima, podemos fazer o seguinte teste:

```
1 def test_http_is_listening(host):
2     http = host.socket("tcp://0.0.0.0:80")
3     assert http.is_listening
```

No arquivo acima, estou fazendo a seguinte pergunta: “Servidor, você tem o endereço socket “tcp://0.0.0.0:80” escutando? Se a resposta for sim, isso indica que meu código não afetou esse comportamento, pois esse teste só será executado após a aplicação do Ansible.

A valor de **host** é inserido pelo Testinfra e por padrão ele usará a máquina onde está rodando o código.

Complicações ao utilizar o Testinfra

Na forma como foi apresentado até aqui, o Testinfra tem uma série de obstáculos para seu uso, que tornam o processo de teste um pouco complexo.

No código apresentado anteriormente, o pacote **pytest-testinfra** e o arquivo **test_http.py** devem estar no servidor destino, o servidor que será configurado com o Ansible. Isso quer dizer que você deve, de alguma forma, fazer isso na máquina destino e ela, possivelmente, ficará com esse “lixo” depois de testada, a não ser que você retire após executar o teste.

É possível utilizar o Testinfra de forma remota, usando o conector [ssh](#)⁵⁹ ou [Ansible](#)⁶⁰, mas isso tornará o processo ainda mais complicado.

⁵⁸<https://testinfra.readthedocs.io/en/latest/>

⁵⁹<https://testinfra.readthedocs.io/en/latest/backends.html#ssh>

⁶⁰<https://testinfra.readthedocs.io/en/latest/backends.html#ansible>

Isso, sem falar, na própria necessidade de ter um servidor destino para testar seu código Ansible, pois imagina toda complicação de ter que destruir o servidor e reinstalar toda vez que seu código Ansible inviabilize o uso daquela máquina?

Qualquer atividade de QA deve ser automatizada ao máximo, ou terá grandes chances de não ser executada pelo time.

Molecule

Para resolver esses problemas, eu te apresento o [Molecule](https://molecule.readthedocs.io/en/latest/)⁶¹, que é uma ferramenta de apoio para testes automatizados no Ansible.

O Molecule visa facilitar a execução de uma rotina completa de QA, pois ela também pode aplicar o lint, realizar os testes, fazer verificação de idempotência e se o código consegue lidar com efeitos colaterais.

Para instalar é muito simples:

```
1 pip install "molecule[ansible]"
```

Para criar uma role já seguindo os padrões do Molecule é muito fácil também:

```
1 molecule init role sua-role-nova --driver-name docker
```

Com o comando acima, será criada uma role com toda estrutura esperada pelo Ansible, mas também com uma pasta chamada **molecule** e dentro dela todo padrão que você pode utilizar para seus testes.

Caso sua role já exista, poderá utilizar o seguinte comando:

```
1 cd sua-role
2 molecule init scenario default --role-name sua-role-nova --driver-name docker
```

Como eu utilizo Ubuntu e **Testinfra** como verificador, meu arquivo **molecule/default/molecule.yml** ficaria da seguinte maneira:

⁶¹<https://molecule.readthedocs.io/en/latest/>


```

1  ---
2  dependency:
3    name: galaxy
4    options:
5      ignore-certs: True
6      ignore-errors: True
7      role-file: requirements.yml
8  driver:
9    name: docker
10 platforms:
11   - name: ubuntu-18.04
12     image: "geerlingguy/docker-${MOLECULE_DISTRO:-ubuntu1804}-ansible:latest"
13     command: ${MOLECULE_DOCKER_COMMAND:-""}
14     published_ports:
15       - 80:80/tcp
16     volumes:
17       - /sys/fs/cgroup:/sys/fs/cgroup:ro
18     privileged: true
19     pre_build_image: true
20 provisioner:
21   name: ansible
22   playbooks:
23     prepare: prepare.yml
24     converge: converge.yml
25 verifier:
26   name: testinfra
27 env:
28   PYTHONWARNINGS: "ignore:.*U.*mode is deprecated:DeprecationWarning"

```

Vamos explicar parte a parte.

```

1  ---
2  dependency:
3    name: galaxy
4    options:
5      role-file: requirements.yml

```

Essa parte é responsável por informar qual gerenciador de dependência será usado e, até o momento, não conheço algo melhor do que o [galaxy](https://docs.ansible.com/ansible/latest/cli/ansible-galaxy.html)⁶². Nas opções, informamos que o arquivo **requirements.yml** conterá a lista das roles que precisa ser baixada para que esteja disponível para ser usada. Perceba que aqui não diz que ela será usada, apenas que fará o download das roles mencionadas.

⁶²<https://docs.ansible.com/ansible/latest/cli/ansible-galaxy.html>

```
1 driver:
2   name: docker
3 platforms:
4   - name: ubuntu-18.04
5     image: "geerlingguy/docker-${MOLECULE_DISTRO:-ubuntu1804}-ansible:latest"
6     command: ${MOLECULE_DOCKER_COMMAND:-""}
7     published_ports:
8       - 80:80/tcp
9     volumes:
10      - /sys/fs/cgroup:/sys/fs/cgroup:ro
11     privileged: true
12     pre_build_image: true
```

Nas seções informadas acima, definimos que usaremos docker para emular a máquina, e a plataforma usada nessa “máquina” será o Ubuntu 18.04. Usaremos uma imagem do [geerlingguy](#)⁶³ nesse exemplo, pois ela tem tudo que precisamos. O **command** é padrão e raramente precisará ser modificado. O **published_ports** define qual porta esse container vai expor e você só precisa colocar caso você queria testar da sua máquina, pois todo teste que falaremos aqui será executado de dentro da “máquina” e não de onde é executado o molecule. Quanto aos valores de **volumes**, **privileged** e **pre_build_image** você também, possivelmente, não precisará mudar.

```
1 provisioner:
2   name: ansible
3   playbooks:
4     converge: converge.yml
```

Na seção acima, é informado que usaremos o Ansible como ferramenta de IaC, definimos também o playbook que será usado na etapa do **converge** do Molecule. Para explicação de cada etapa do Molecule, acesse [essa parte da documentação](#)⁶⁴. É importante salientar que o Molecule aceita apenas o Ansible como provisioner.

```
1 verifier:
2   name: testinfra
```

Nessa última seção, configuramos que será usado o Testinfra como verificador automatizado de código.

O seu teste deve ser colocado dentro da pasta **molecule/default/tests**, com seu nome seguindo o mesmo padrão apresentado anteriormente (começando com “test_”), porém o conteúdo do arquivo deve seguir a estrutura:

⁶³<https://twitter.com/geerlingguy>

⁶⁴<https://molecule.readthedocs.io/en/latest/usage.html>

```
1 import os
2
3 import testinfra.utils.ansible_runner
4
5 testinfra_hosts = testinfra.utils.ansible_runner.AnsibleRunner(
6     os.environ['MOLECULE_INVENTORY_FILE']).get_hosts('all')
7
8
9 def test_http_is_listening(host):
10     http = host.socket("tcp://0.0.0.0:80")
11     assert http.is_listening
```

A seção nova que aparece nesse arquivo é responsável por configurar o Testinfra para utilizar a “máquina” entregue pelo Molecule.

Após tudo configurado, só nos resta executar o teste:

```
1 molecule test
```

Esse comando será responsável por:

- dependency
- lint
- cleanup
- destroy
- syntax
- create
- prepare
- converge
- idempotence
- side_effect
- verify
- cleanup
- destroy

Os nomes de cada etapa são auto-explicativos, mas se tiver dúvidas, basta ler a [documentação oficial](https://molecule.readthedocs.io/en/latest/usage.html)⁶⁵.

Caso você queria testar seu código à medida que o cria, você pode apenas executar os seguintes comandos:

⁶⁵<https://molecule.readthedocs.io/en/latest/usage.html>

```
1 molecule create
2 molecule converge
```

O **create** será responsável por criar o container que atuará como “máquina” e o **converge** aplicará o código Ansible nesse “servidor” temporário, que na verdade é um container docker.

Após aplicar com sucesso seu código Ansible no **converge**, você deve executar o comando abaixo:

```
1 molecule verify
```

Ele será responsável por executar o **Testinfra** nessa “máquina” temporária e assim validar o comportamento após execução do **converge**.

Meu conselho, antes de fazer commit e push do seu código IaC é que você execute o **molecule test**. Ele executa todos os passos em um servidor recém-criado. Além disso, ele também executa o **idempotence** que é responsável por avaliar se alguma tarefa que você definiu no Ansible está idempotente. Em outras palavras, significa que a segunda aplicação do Ansible na mesma máquina, sem modificar nenhum código Ansible, deve ter um resultado sem modificação alguma do servidor.

A idempotência é uma melhor prática que deve ser buscada sempre, pois não faz sentido algum, uma mesma task sendo aplicada no mesmo servidor duas vezes fazer modificações no destino. Isso indica que seu código tem problemas e pode estar atuando na máquina de forma equivocada, o que pode ocasionar em duplicação de arquivos ou até mesmo configuração com problemas em arquivos.

Conclusão

Automação de QA para IaC não é fácil e não temos muito materiais, mas o pouco que temos já consegue oferecer alguma segurança e devemos utilizá-las em nosso processo de entrega automatizada de IaC.

Deploy de Infraestrutura como código (Rafael Gomes)

Introdução

Explicamos aqui no capítulo “O que é deploy?”, mas aqui nesse capítulo falaremos sobre um tipo especial de deploy. Esse que lida com a infraestrutura.

Infraestrutura como código (IaC) é um grande domínio de conhecimento. Falaremos especificamente de infraestrutura de servidores, onde é criado e/ou mantido uma plataforma que receberá código posteriormente.

Existem algumas formas de manter uma infraestrutura mantida de forma automatizada. Quando se fala sobre “deploy de infraestrutura como código (Deploy de IaC)” estamos falando de criar e/ou fazer modificação em uma infraestrutura com base em um código *infra as code* (IaC) de um repositório.

Para facilitar o entendimento, ignore a existência de container (docker, podman ou afins), pois isso seria uma outra camada, que explicaremos em uma outra oportunidade.

Exemplo

Imagine a necessidade de criar e manter um servidor web para servir um código PHP.

Nesse servidor precisa ser instalado alguns softwares principais:

- [FPM](https://www.php.net/manual/pt_BR/install.fpm.php)⁶⁶: que é responsável pela tradução do código PHP.
- [NGINX](https://www.nginx.com/)⁶⁷: que é responsável por fazer o proxy da requisição do usuário para o FPM.

Esses softwares precisam ser devidamente instalados e configurados de acordo com definições do seu uso.

Quando se fala em fazer deploy de um servidor web para PHP, estamos falando da existência de um código IaC que será responsável por todo processo de instalar e configurar tudo que precisa para que exista a infraestrutura pronta para receber o código PHP e servir o site que você deseja.

⁶⁶https://www.php.net/manual/pt_BR/install.fpm.php

⁶⁷<https://www.nginx.com/>

Deploy de infra é sobre infra

Muitas pessoas têm dificuldade para entender deploy de infraestrutura, pois há uma confusão sobre deploy do produto em si e de sua infra.

Em todo momento que falarmos sobre deploy neste capítulo será apenas sobre infraestrutura. Que a depender da situação, pode ser chamado também de plataforma.

Em alguns casos o código pode ser “deployado” ao mesmo tempo, mas isso não é uma regra.

No exemplo citado anteriormente de um serviço PHP, não estamos falando do código PHP, apenas os softwares que deixaremos pronto para receber o deploy do código posteriormente.

Tipos de deploy de IaC

Quando se fala de deploy de plataformas, especificamente aquelas que envolvem servidores, temos dois caminhos possíveis:

- Gerência de configuração
- Infraestrutura imutável

Gerência de configuração

Deploy de IaC por gerência de configuração é quando você usa uma ferramenta que conecta em um servidor existente e faz intervenções nele. Nesse modelo o mesmo servidor sofre alterações com o passar do tempo.

Sempre que for preciso modificar o comportamento da plataforma, você precisará rodar a ferramenta de gerência de configuração escolhida no mesmo servidor, e ela será responsável por modificar o servidor com base na mudança que você informou no código. Seja por mudança do código de IaC ou suas variáveis.

Essa comunicação entre a ferramenta de gerência de configuração e o servidor normalmente acontece via SSH (Para servidores GNU/Linux) e winrm (Para servidores Microsoft Windows). Esse software de IaC conecta na máquina e faz intervenções baseado no que está no código e suas variáveis.

É importante salientar que as ferramentas de gerência de configuração farão de tudo para garantir que o que foi definido no código, pois estes softwares garantem mudanças de estado atual para o estado desejado definido como IaC, desde que este processo não falhe por algum motivo específico.

Exemplo:

Se o IaC define que um determinado serviço deveria estar em execução, como podemos ver no código abaixo:

```
1 - name: Iniciar o nginx
2   ansible.builtin.service:
3     name: nginx
4     state: started
```

No código acima temos as seguintes instruções detalhadas:

```
1 - name: Iniciar o nginx
```

Descreve o nome da task em questão.

```
1 ansible.builtin.service:
```

Descreve o nome do módulo ansible, que nesse caso é o `service`⁶⁸ que é responsável por gerenciar serviços.

```
1 name: nginx
2 state: started
```

Descreve o nome do serviço que deseja gerenciar e qual estado esperado dele, ou seja, caso o serviço esteja desligado, ele vai iniciar. Caso esteja iniciado, não faz nada.

Nesse exemplo, o código garante que o serviço iniciará caso esteja parado, mas se o serviço cair logo após o ansible inicia-lo, o ansible não saberá disso, a menos que exista outra task que faça essa checagem posteriormente.

Infraestrutura imutável

Antes de falar o que é Infraestrutura **imutável**, vamos explicar o que é Infraestrutura **mutável**.

A infraestrutura mutável é exatamente o que é feito na gerência de configuração. Uma máquina é criada com uma determinada configuração e a **mesma máquina** sofre alterações para que um novo estado do serviço seja alcançado.

No exemplo demonstrado anteriormente usando o ansible, ele se conecta a um servidor existente. Tudo é feito na mesma instância, a não ser que alguém solicite a recriação dessa máquina.

Exemplo

Ao instalar o serviço nginx pela primeira vez, é escolhida a máquina que tem o nome de **bakunin** e o ansible conecta em bakunin por ssh e faz a instalação do nginx.

Se for necessário modificar a versão do nginx ou instalar um outro software, o ansible conectará em **bakunin** via ssh novamente e aplicará o que de novo foi adicionado no código ansible.

Infraestrutura mutável é aquela que pode mudar dentro da mesma instância. Ela “nasce” em um estado de configuração e um processo externo pode fazer esse estado mudar.

⁶⁸https://docs.ansible.com/ansible/latest/collections/ansible/builtin/service_module.html

E a Infraestrutura imutável?

Infraestrutura imutável é aquela que não muda, certo? Sim, mas pra explicar isso melhor vamos ampliar a visão sobre essa questão.

Quando falamos sobre infra que muda ou não, falamos do servidor em específico, no exemplo apresentado anteriormente, uma vez que o servidor **bakunin** fez boot pela primeira vez ele não seria modificado até o dia que fosse desligado.

O serviço que **bakunin** hospeda poderá continuar em outra máquina, mas a instância **bakunin** não mudará no modelo de Infraestrutura imutável.

Não há nenhuma conexão ssh, ou outro método, para instalar ou configurar nada. A instância “nasce” com tudo que precisa para hospedar o serviço a que foi designada.

Na infraestrutura imutável é necessário criar uma imagem, uma espécie de fotografia da instância em um determinado momento, para que quando necessário criar a instância, tudo seja iniciado sem precisar de gerência de configuração após ligar a máquina.

Fotografia ou Snapshot, é o termo utilizado para referenciar um recurso que não pode ser modificado, igualmente a fotografia, podemos escrever(desenhar) por cima mas seu conteúdo base é impossível de ser modificado por este processo.

Exemplo

Usando o mesmo exemplo de uma máquina com nginx, ao invés de iniciar uma instância genérica, sem nginx, e após seu boot conectar via ssh e completar sua configuração, é criada uma máquina temporária, onde a instalação e configuração do nginx é feita através de uma ferramenta de configuração, e o processo final resulta em uma imagem (snapshot da instância)

A imagem será usada quando for necessário fazer o deploy da infraestrutura imutável, pois nesse caso iniciaremos a máquina com essa imagem. Dessa forma a instância inicia já com nginx e nenhuma configuração posterior será necessária. E toda vez que for necessário trocar algo na instância, ao invés de conectar na mesma, é criada uma máquina nova a partir de uma imagem nova. Você pode inclusive manter as duas instâncias imutáveis executando ao mesmo tempo e desligar a antiga apenas quando tiver certeza que a nova está funcionando como esperado.

Importante notar que nem todos os serviços suportam uma infraestrutura imutável, mas falaremos disso melhor em outra oportunidade.

Conclusão

Existem dois tipos de deploy de IaC e o que diferencia eles é como eles podem ser modificados, na gerência de configuração a modificação é feita na mesma máquina e na infra imutável ela é feita com a criação de outra máquina e modificando o apontamento do serviço para essa máquina nova.

Em ambos casos o serviço é modificado, mas apenas na gerência de configuração é utilizada a mesma máquina.

O modelo de infra imutável é o ideal, mas nem sempre é possível. Aprender a conviver com a gerência de configuração, da melhor forma possível, é uma habilidade necessária para se desenvolver, caso seu objetivo seja trabalhar com IaC em ambientes reais.

O livro ainda não está finalizado

Esse livro está em construção e ainda não terminamos a sua versão 1.0. Estamos trabalhando muito duro para garantir que os conhecimentos necessários serão encontrados na ordem correta e com aprofundamento necessário para que todas as pessoas possam aprender.

Você receberá uma notificação a cada atualização do livro.

Se você tem alguma sugestão para capítulos, por favor acesse [esse link](#)⁶⁹ e cadastre uma *issue*.

A *issue* é um registro que pode ser entendido de diversas formas, mas nesse que você fará, ele será entendido como um pedido de melhoria. Você estará colaborando com o conteúdo do livro. Expondo o que deseja que ele tenha.

Se você achar alguma falha, você pode abrir um *issue* também, ou corrigir (se possível) e enviar um *pull request*.

Muito obrigado a todas as pessoas que colaboraram e ainda colaboram com esse projeto.

⁶⁹<https://github.com/gomex/deploy-em-producao/issues>