Aprenda com quem faz

A Primeira Maneira: Os Princípios do Fluxo

Bárbara Cabral da Conceição

2022

SUMÁRIO

> Capítulo 1

Capítulo 1. Princípios da Primeira Maneira

A primeira maneira consiste em acelerar o fluxo de valor de entrega para o cliente da esquerda (desde a concepção) para a direita (uso do software pelo cliente).

1.1. Os princípios do Fluxo

Dentre os princípios da primeira maneira estão:

1. Tornar o trabalho visível;
2. Reduzir o tamanho dos lotes de trabalho;
3. Aplicar a teoria das restrições e otimizar o fluxo;
4. Remover desperdícios e fazer com que o foco seja o cliente;
5. Reduzir o tamanho das transferências (handof);
6. Incorporar qualidade na origem (shift left);
7. Limitar o trabalho em andamento / WIP (work in progress);
8. Infraestrutura como código / Infrastructure as a Service (IaaS);
9. Integração Contínua, Entrega e Implantação Contínua;
10. Automatizar etapas manuais (Testes Automatizados, TDD etc.);
11. Arquitetura e Releases de Baixo Risco.

As vantagens principais ou resultados de melhorar o fluxo são:

- Não passa um defeito conhecido ao *downstream,* ou seja, defeitos são conhecidos em ciclos curtos de *feedback*;
- Não permite a otimização local para degradação global;

- Sempre busca o aumento do fluxo, seja em velocidade de entrega ou em aumento de número de estórias a serem entregues por ciclo;
- Sempre busca obter um conhecimento profundo do sistema.

1.2. Metodologias que ajudam a reduzir desperdício

Algumas metodologias ajudam a reduzir desperdícios, a principal delas é baseada no método Lean Manufacturing, que foi o Sistema Toyota de Produção (TPS). Ele foi desenvolvido por Taiichi Ohno no Japão logo após a Segunda Guerra Mundial, e foi baseado no modelo de linha de produção de Henry Ford e Frederick Taylor. O conceito principal é a linha de produção enxuta. Os mesmos princípios do Lean podem ser aplicados também para o desenvolvimento do software, são eles:

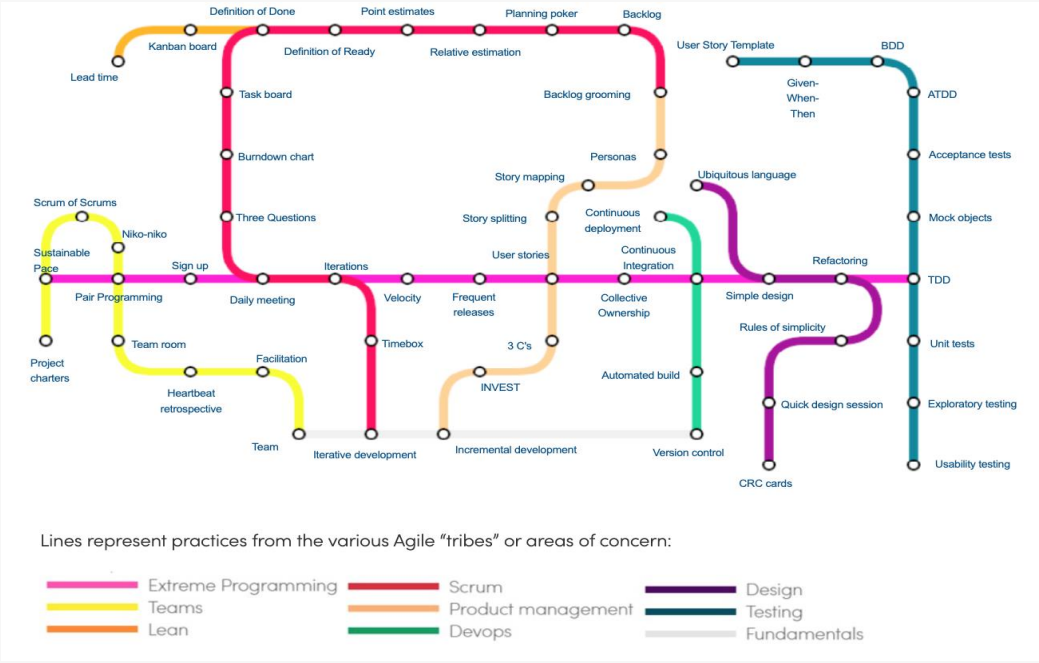
- Eliminar desperdício (Eliminate waste).
- Construir Qualidade Interna (Build quality in).
- Criar conhecimento (Create knowledge).
- Adiar compromisso de coisas não definidas (Defer commitment).
- Entregar rápido (Deliver fast).
- Respeitar as pessoas (Respect people).
- Otimizar o todo (Optimize the whole).

Figura 1 – Lean aplicado ao Desenvolvimento de Software.

Fonte: Hangout Agile https://hangoutagile.com/lean-software-development-wave-ii/.

Baseados no Lean, nasceram várias metodologias ágeis, a mais famosa delas foi o Kanban. E com certeza a Primeira Maneira, Fluxo, também é muito baseada no Lean e no Kanban. Dentro da metodologia Ágil, também temos o Scrum, além de outras práticas ágeis que podemos observar no Mapa Abaixo:

Figura 2 – Subway Map to Agile Practices.



Fonte: https://www.agilealliance.org/agile101/subway-map-to-agile-practices/.

> Capítulo 2

Capítulo 2. Pipelines de Implantação

Mik Kersten define um fluxo de valor como:

"O conjunto de atividades de ponta a ponta executadas para entregar valor a um cliente por meio de um produto ou serviço".

2.1. Fluxo de Valor e Elementos de um Pipeline

O mapeamento do fluxo de valor é uma técnica Lean para visualizar, caracterizar e melhorar continuamente o fluxo de valor em todo esse conjunto de atividades de ponta a ponta, eliminando barreiras, sejam elas processuais, culturais, tecnológicas ou organizacionais. Na literatura Lean, essas barreiras são chamadas de “desperdício” (waste). Exemplos comuns de desperdício em organizações de desenvolvimento de software incluem espera desnecessária ou troca de tarefas, adição de recursos extras que geram inchaço de software e duplicação de código em violação do princípio “não se repita” (DRY).

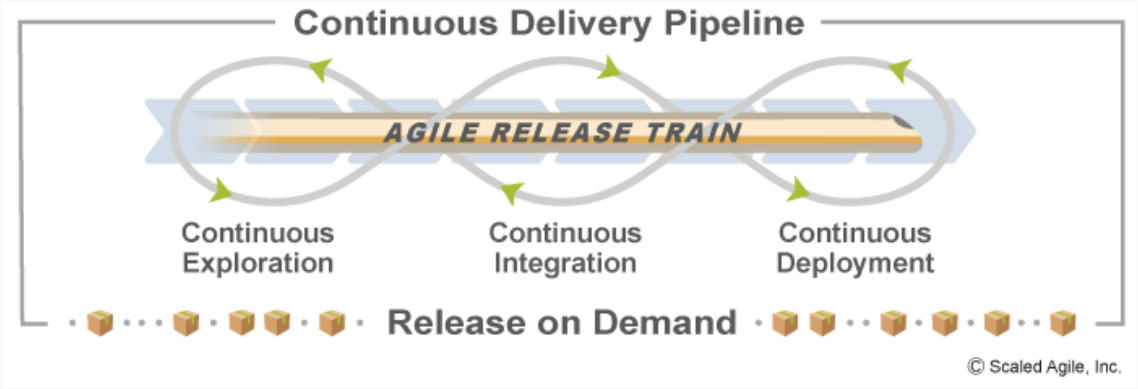
Mapear um fluxo de valor de desenvolvimento de software pode revelar processos desnecessários, etapas de homologação excessivamente complexas, longos atrasos nos testes devido a processos manuais etc.

O Continuous Delivery Pipeline (CDP) representa os fluxos de trabalho, as atividades e a automação necessários para conduzir uma nova funcionalidade, desde a concepção até a liberação de valor sob demanda para o usuário final. Conforme ilustrado na Figura 3, o pipeline consiste em quatro aspectos:

- Exploração Contínua / Continuous Exploration (CE);
- Integração Contínua / Continuous Integration (CI);
- Implantação Contínua / Continuous Deployment (CD);

- Liberação sob Demanda / Entrega Contínua / Continuous Delivery (CD).

Figura 3 – The SAFe Continuous Delivery Pipeline © Scaled Agile, Inc.



Fonte: https://www.scaledagileframework.com/continuous-delivery-pipeline/.

Construir e manter um pipeline de entrega contínua fornece a cada arte a capacidade de fornecer novas funcionalidades aos usuários com muito mais frequência do que com os processos tradicionais. Para alguns, “contínuo” pode significar lançamentos diários ou até mesmo lançamentos várias vezes por dia. Para outros, contínuo pode significar lançamentos semanais ou mensais – o que satisfaça as demandas do mercado e os objetivos da empresa. As práticas tradicionais tendem a perceber os lançamentos como grandes blocos monolíticos. No entanto, a realidade é que a liberação de valor não precisa se traduzir em uma abordagem de “tudo ou nada”.

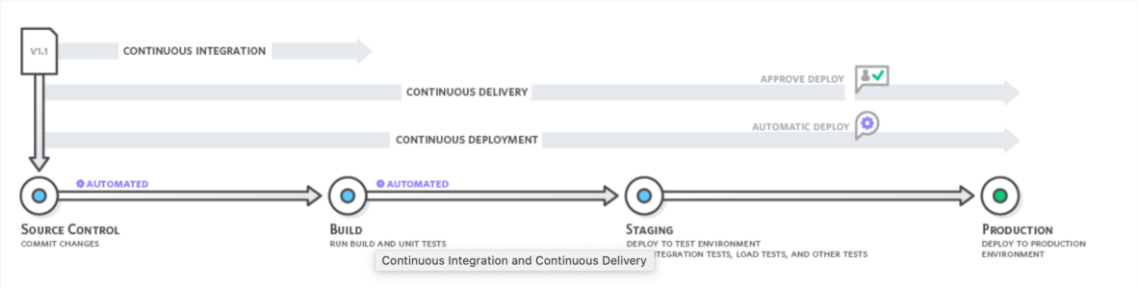
A dissociação de funcionalidades em pequenos blocos de entrega possibilita a entrega com maior frequência e menor risco. Por esse motivo, atualmente prefere-se uma arquitetura de microsserviços em muitos casos.

2.2. Integração Contínua / Continuous Integration

A integração contínua é uma prática de desenvolvimento de software DevOps em que os desenvolvedores mesclam (merge) regularmente suas alterações de código em um repositório central, após o qual compilações e testes automatizados são executados. A integração

contínua geralmente se refere ao estágio de construção (build) ou integração do processo de lançamento de software e envolve tanto um componente de automação (p. ex., um CI ou serviço de build) quanto um componente cultural (p. ex., aprender a integrar com frequência). Os principais objetivos da integração contínua são encontrar e solucionar bugs mais rapidamente, melhorar a qualidade do software e reduzir o tempo necessário para validar e lançar novas atualizações de software.

Figura 4 - Estágios de Integração Contínua até a entrega.



Fonte: https://aws.amazon.com/devops/continuous-integration/.

Benefícios da Integração contínua:

- Melhore a produtividade do desenvolvedor: A integração contínua ajuda sua equipe a ser mais produtiva, liberando os desenvolvedores de tarefas manuais e incentivando comportamentos que ajudam a reduzir o número de erros e bugs liberados para os clientes.

- Encontre e resolva bugs mais rapidamente: com testes mais frequentes, sua equipe pode descobrir e resolver bugs mais cedo, antes que eles se transformem em problemas maiores posteriormente.

- Entregue atualizações mais rapidamente: a integração contínua ajuda sua equipe a fornecer atualizações aos clientes com mais rapidez e frequência.

2.3. Implantação Contínua / Continuous Deployment (CD)

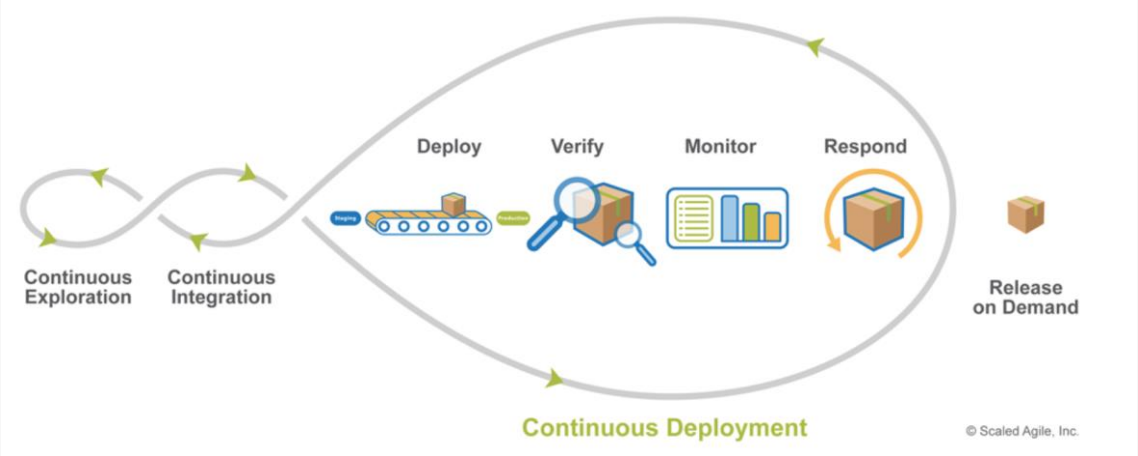
A Implantação Contínua se concentra – assim como o nome indica – na implantação; a instalação e distribuição reais do software. No sentido tradicional, a implantação contínua se concentra na automação para implantação em ambientes ou clusters. Ela se concentra no caminho de menor resistência para colocar o software no(s) ambiente(s) necessário(s).

SAFe descreve 4 atividades de implantação contínua:

- Deploy para produção: descreve as práticas necessárias para implantar uma solução em um ambiente de produção;
- Verifique: se a solução descreve as práticas necessárias para garantir que as alterações funcionem na produção conforme pretendido antes de serem liberados para os clientes;
- Monitore: problemas descreve as práticas para monitorar e relatar quaisquer problemas que possam surgir na produção;
- Responda: e recuperar descreve as práticas para resolver rapidamente quaisquer problemas que ocorram durante a implantação.

Com plataformas modernas, como Kubernetes, a separação de ambientes pode não ser física quando comparada a plataformas legadas ou tradicionais baseadas em máquina. Um namespace (separação de software) pode ser tudo o que separa o desenvolvimento da produção, embora bons princípios de sistemas distribuídos ainda se aplicam, independentemente da plataforma escolhida. Com sistemas distribuídos, as topologias para as quais as alterações precisam se propagar podem ser grandes, mesmo em ambientes de pré-produção.

Figura 5 - Fases da Implantação Contínua.



Fonte: https://www.scaledagileframework.com/continuous-deployment/.

2.4. Entrega Contínua / Continuous Delivery(CD)

A tecnologia é falível porque os humanos constroem tecnologia. A Entrega Contínua é a automação de etapas para obter mudanças com segurança na produção. Onde a Implantação Contínua se concentra na implantação real, a Entrega Contínua se concentra no lançamento e na estratégia de lançamento. Um objetivo indescritível seria um “apertar de um botão” para colocar as alterações em produção. Esse “apertar de um botão” é a Entrega Contínua.

Com a entrega contínua, temos a capacidade de lançar alterações de todos os tipos sob demanda de maneira rápida, segura e sustentável. As equipes que treinam bem a entrega contínua podem lançar softwares e fazer alterações na produção a qualquer momento, correndo poucos riscos, inclusive durante o horário comercial, sem afetar os usuários. É possível aplicar os princípios e as práticas de entrega contínua a qualquer contexto de software, incluindo:

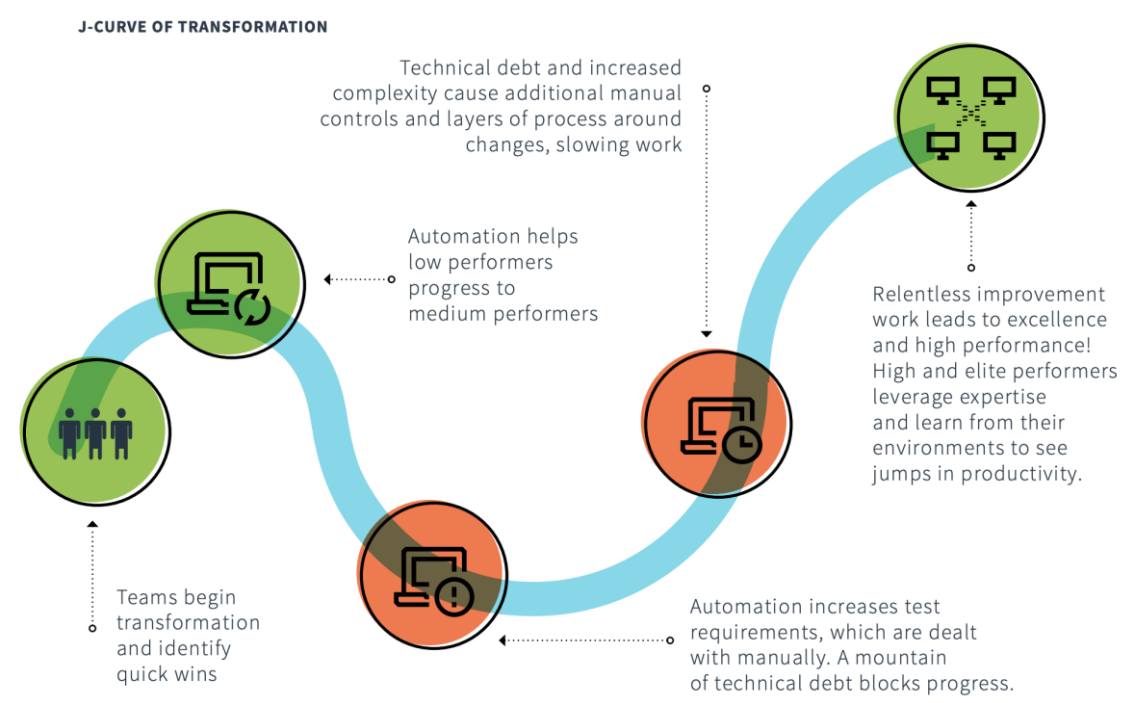
- Atualizar serviços em um sistema distribuído complexo;
- Fazer upgrade do software de mainframe;
- Fazer alterações na configuração da infraestrutura;

- Fazer alterações no esquema do banco de dados;
- Atualização automática do firmware;
- Lançar novas versões de um app para dispositivos móveis.

Durante o processo de encontrar a curva de transformação para a entrega contínua, geralmente os times encontram alguns obstáculos que podem ser observados de acordo com a pesquisadora DORA:

- No início da curva, as equipes começam a transformação e identificam conquistas rápidas.
- Em uma melhoria inicial, a automação ajuda os usuários com baixo desempenho a progredir para os de médio desempenho.
- Em uma redução da eficiência (a parte inferior da curva J), a automação aumenta os requisitos de teste, que são tratados manualmente. Uma grande quantidade de dívida técnica bloqueia o progresso.
- À medida que as equipes começam a sair da curva, a dívida técnica e o aumento da complexidade causam mais controles manuais e camadas de processo em torno das mudanças, diminuindo o trabalho.
- Na parte superior da curva, o trabalho de aprimoramento contínuo leva a um desempenho excelente. Profissionais de alto nível aproveitam a experiência e aprendem com os ambientes para aumentar a produtividade.

Figura 6 - Armadilhas comuns da implementação da entrega contínua.



Fonte: https://cloud.google.com/architecture/devops/devops-tech-continuous-delivery.

Por fim, o objetivo da entrega contínua é garantir que os lançamentos sejam executados com baixo risco durante o horário comercial normal. O objetivo é que ninguém precise trabalhar fora do horário comercial normal para realizar implantações ou lançamentos. Portanto, isso é algo que é importante avaliar.

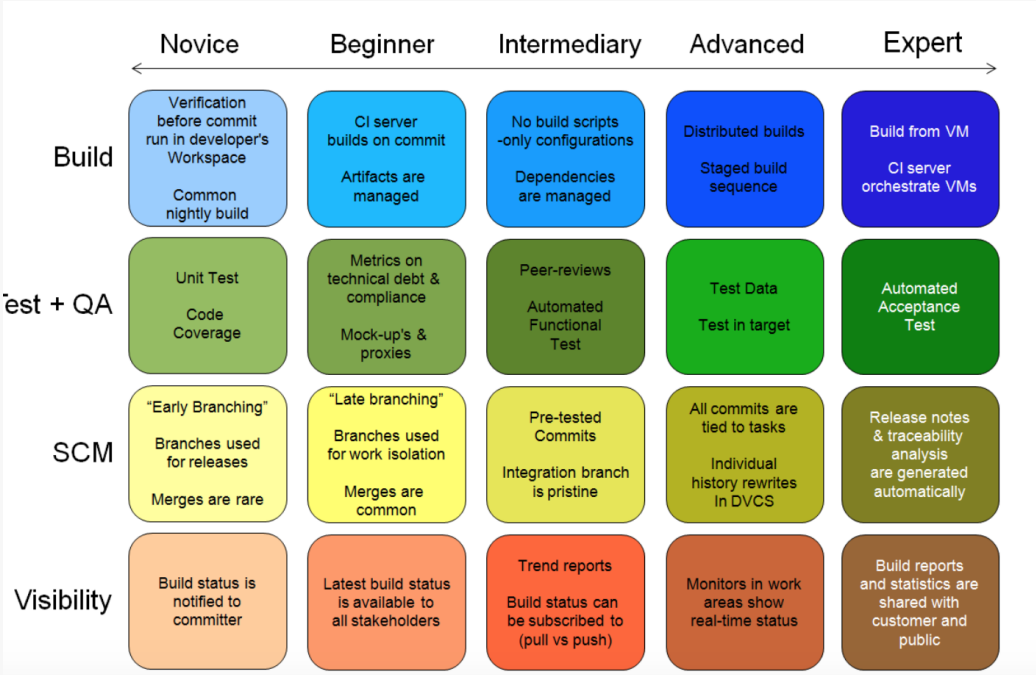
O desempenho na entrega contínua é refletido nos resultados alcançados. É possível fazer a verificação rápida de DevOps da DORA (em inglês) para ver o desempenho com as principais métricas de entrega contínua.

Continuous Delivery checklist

Os princípios de Martin Fowler são um ótimo ponto de partida para pensar na melhor configuração do seu processo de desenvolvimento de software. Jez Humble e David Farley também apontam em seu livro

Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation um checklist que você pode se basear.

Figura 7 - Continuous Delivery Maturity Matrix by Chris Shayan.



Fonte: https://www.cloudbees.com/continuous-delivery/continuous-integration.

> Capítulo 3

Capítulo 3. Controles de Versão

O controle de versão, também conhecido como controle de fonte, é a prática de rastrear e gerenciar as alterações em um código de software. Os sistemas de controle de versão são ferramentas de software que ajudam as equipes de software a gerenciar as alterações ao código-fonte ao longo do tempo. Como os ambientes de desenvolvimento aceleraram, os sistemas de controle de versão ajudam as equipes de software a trabalhar de forma mais rápida e inteligente. Eles são ainda mais úteis para as equipes de DevOps, pois as auxiliam a reduzir o tempo de desenvolvimento e aumentar as implementações bem-sucedidas.

O repositório é o local onde o sistema de controle de versão mantém o rastreamento de todas as alterações realizadas no projeto. A maioria dos VCSs apenas armazenam o estado atual do código, além de quando essa alteração foi realizada, quem a fez, e uma mensagem de texto em um log que explica o porquê da alteração ter sido realizada.

3.1. Conceitos Básicos

Histórico de alterações completo e a longo prazo de todos os arquivos:

Isso significa todas as alterações feitas por muitas pessoas ao longo dos anos. As alterações incluem a criação e exclusão de arquivos, assim como as edições em seus conteúdos. Diferentes ferramentas de versionamento diferem na maneira de lidar com a renomeação e a movimentação de arquivos, se melhor ou pior. Esse histórico também deve incluir o autor, a data e as notas escritas sobre o objetivo de cada alteração. Ter o histórico completo permite voltar às versões anteriores para ajudar na análise da causa raiz de bugs e é crucial para corrigir problemas nas versões

mais antigas do software. Se o software estiver sempre sendo trabalhado, quase tudo poderá ser considerado uma "versão mais antiga" do software.

Ramificação e mescla:

O trabalho simultâneo da equipe é certo, mas mesmo os indivíduos que trabalham sozinhos podem se beneficiar da capacidade de trabalhar em fluxos independentes de mudanças. Criar uma "ramificação" nas ferramentas de versionamento mantém vários fluxos de trabalho independentes uns dos outros, além de oferecer a facilidade de mesclar esse trabalho de novo, permitindo que os desenvolvedores verifiquem se as alterações em cada ramificação não estão em conflito. Muitas equipes de software adotam a prática de ramificação para cada funcionalidade, ou talvez ramificação para cada versão, ou ambos. As equipes podem escolher entre vários fluxos de trabalho diferentes quando decidem como usar os recursos de ramificação e fusão.

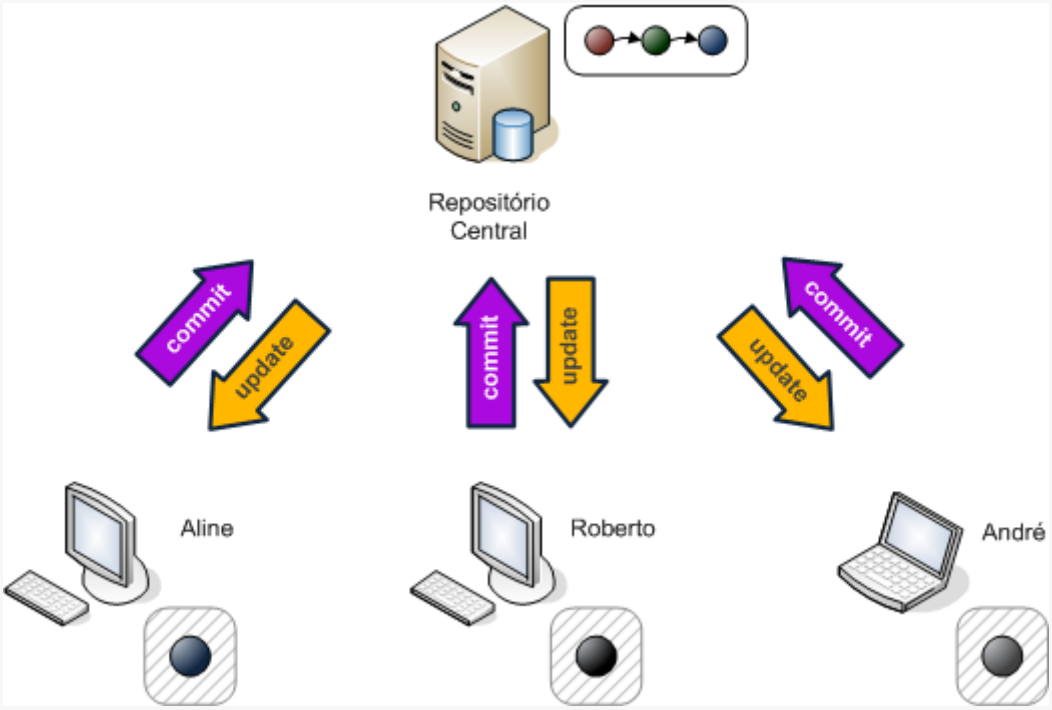
Rastreabilidade:

Ser capaz de rastrear cada alteração feita no software e conectar ao software de gestão de projetos e rastreamento de bugs, como o Jira, e ser capaz de anotar cada alteração com uma mensagem descrevendo o objetivo da mudança ajuda não só com a análise de causa raiz e outras análises forenses. Ter o histórico anotado do código na ponta dos dedos quando você estiver lendo o código, tentando entender o que está fazendo e por que ele foi projetado assim pode permitir que os desenvolvedores façam alterações corretas e harmoniosas que estejam de acordo com o design do sistema pensado para o longo prazo. Em especial, isso pode ser importante para o trabalho efetivo com o código legado e é crucial para permitir que os desenvolvedores calculem o trabalho futuro com precisão.

Controle de Versão Centralizado:

O controle de versão centralizado segue a topologia em estrela, havendo apenas um único repositório central, mas várias cópias de trabalho, uma para cada desenvolvedor. A comunicação entre uma área de trabalho e outra passa obrigatoriamente pelo repositório central. No controle de versão centralizado há um único repositório e várias cópias de trabalho que se comunicam apenas através do repositório central.

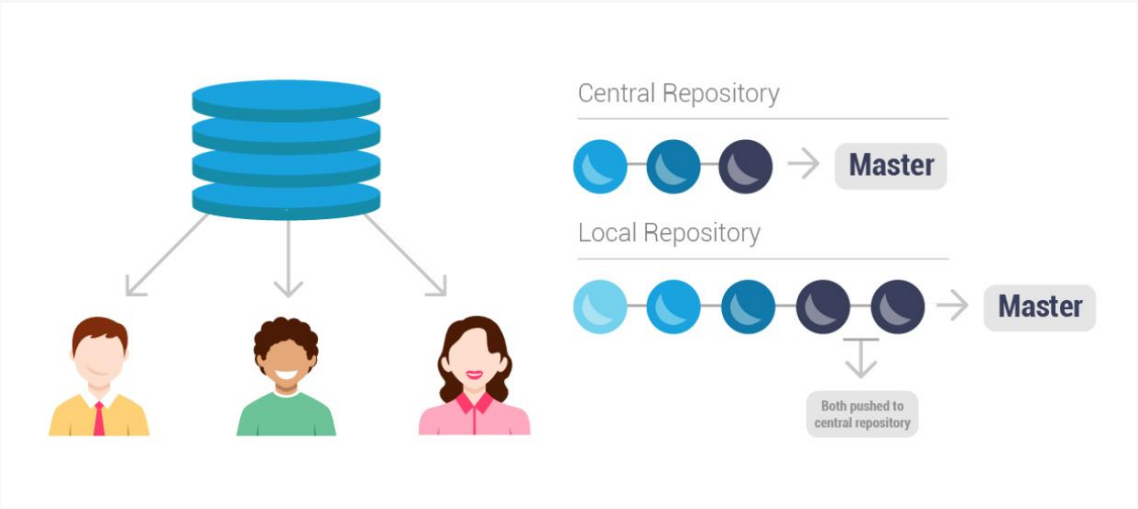
Figura 8 - Topologia em Estrela.



Fonte: https://blog.pronus.io/posts/controle-de-versao/conceitos-basicos-de-controle-de-versao-de-software-centralizado-e-distribuido/.

Como vantagem, esse fluxo de trabalho não necessita que a equipe tenha que gerenciar vários ramos de código. Como desvantagem, aumenta a probabilidade de existirem conflitos no código, já que todos os desenvolvedores estão utilizando o mesmo ramo de código, o que pode inserir bugs no sistema.

Figura 9 - Desenvolvedores trabalham diretamente no único ramo existente.



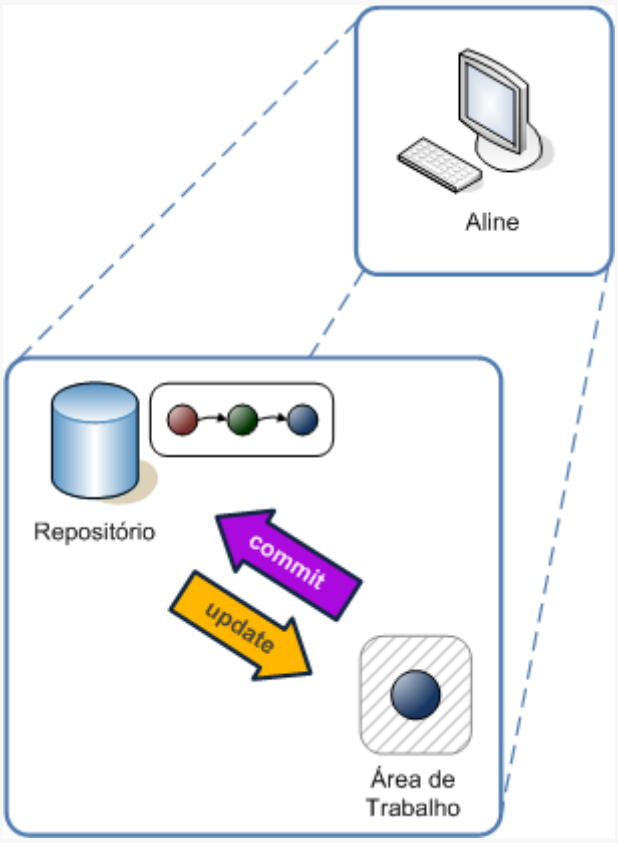
Fonte: https://blog.enacom.com.br/2019/01/07/controle-de-versao-e-gerenciamento-de-codigo/.

Outra desvantagem do fluxo de trabalho centralizado é observada quando há a necessidade de criar uma nova release em produção e existem novas features no repositório que ainda não foram finalizadas e não irão compor a release. Nesse caso, será necessário adotar medidas para impedir que a feature não finalizada não faça parte da release.

Controle de Versão Distribuído:

São vários repositórios autônomos e independentes, um para cada desenvolvedor. Cada repositório possui uma área de trabalho acoplada e as operações commit e update acontecem localmente entre os dois. No controle de versão distribuído, cada desenvolvedor possui um repositório próprio acoplado a uma área de trabalho. A comunicação entre eles continua sendo através de commit e update.

Figura 10 - Controle de versão distribuído.



Fonte: https://blog.pronus.io/images/controle_versao/conceitos_basicos/repositorio_desenvolvedor.png.

3.2. Manipulando arquivos e sincronizando

De forma geral, o repositório precisa ter tudo que é essencial para o projeto, tanto para codificar, modificar quanto para construir as versões. Esses artefatos armazenados podem incluir Makefiles, Rakefiles, código-fonte, documentos do projeto, build.xml do Ant, imagens que são usadas pela aplicação, testes de unidade, entre outros.

Um repositório pode se comunicar com qualquer outro através das operações básicas pull e push:

- pull (Puxar). Atualiza o repositório local (destino) com todas as alterações feitas em outro repositório (origem).

- push (Empurrar). Envia as alterações do repositório local (origem) para um outro repositório (destino).

A sincronização entre os desenvolvedores acontece de repositório a repositório e não existe, em princípio, um repositório mais importante que o outro, embora o papel de um repositório central possa ser usado para convencionar o fluxo de trabalho.

Resumo das Operações Básicas dos Controles de Versão Centralizado e Distribuído:

Centralizado	Distribuído	Descrição
checkout	clone	criação da cópia de trabalho/repositório
commit	commit	envia alterações para o repositório, criando uma revisão
update	update	atualiza a cópia/área de trabalho em uma revisão
	pull	importa revisões feita em outro repositório
	push	envia revisões locais para outro repositório

Working Trees

Um Working Tree é onde as alterações são realizadas. Assim, o working tree é a visão atual do repositório e pode incluir código-fonte, documentos, arquivos build, testes de unidades etc.

Para iniciar um working tree, basta inicializar um repositório no Git ou ainda pode-se clonar um repositório existente. O clone faz uma cópia de outro repositório e então faz um checkout do branch master (o master é a

linha principal de desenvolvimento). O checking out é o processo que o Git usa para alterar o working tree para um certo ponto do repositório.

O Git rastreia os arquivos armazenados no repositório como “conteúdo”. Esta abordagem é diferente da maioria dos sistemas de controle de versão que rastreavam arquivos. Assim, ao invés de rastrear um arquivo como "modelos.py", o Git rastreia o seu conteúdo, ou seja, caracteres individuais e linhas, então o Git adiciona metadados para complementar as informações. Esta diferença é sutil, mas bastante importante.

3.3. Tags, branches, merging

Tags

Assim que os projetos progridem, eles atingem marcos. Em um projeto que utiliza uma metodologia ágil e tem o desenvolvimento em ciclos semanais, as funcionalidades são adicionadas a cada semana, já em um projeto que segue uma metodologia tradicional, as funcionalidades são lançadas em poucos meses.

Em qualquer uma das metodologias, faz-se necessário acompanhar e marcar o estado do repositório quando um marco importante for atingido, como por exemplo, uma entrega.

As tags marcam um certo ponto no histórico do repositório tal que elas podem facilmente ser referenciadas mais tarde. Assim, uma tag é simplesmente um nome que se pode usar para marcar algum ponto específico na história do repositório, como por exemplo, uma entrega ou algum ponto de correção de algum bug. As tags são diferentes dos branches, que será visto na próxima seção.

Branches

O Git permite a criação de branches que marcam um ponto onde os arquivos do repositório divergem. Cada branch mantém um registro das

alterações feitas ao seu conteúdo separadamente de outros branches para que seja permitido criar histórias alternativas no projeto.

O branch master é a linha principal de desenvolvimento, também conhecido como trunk nos VCSs. O branch quebra essa linha em outros "ramos" (como uma árvore que tem o tronco e vários ramos a partir daquele tronco principal).

As boas práticas indicam que um branch pode existir durante toda a vida de um projeto ou por apenas algumas horas, como um branch que foi criado para realizar algum experimento no projeto. Um branch também pode ser mesclado (merged) com outro branch, mas também isto não é uma regra, ou seja, um branch não precisa ser sempre mesclado com outro branch.

Como tudo no Git, um branch pode ser criado localmente sem precisar compartilhá-lo no repositório remoto. Isto pode ser usado para o desenvolvedor criar experimentos com algumas alterações e deletá-lo se não funcionar.

De fato, a maioria dos branches não precisam ser mescladas com outros branches para mantê-lo atualizado. Na parte prática do artigo isso será mais bem exemplificado.

Merging

O merge nada mais é do que combinar dois ou mais históricos de branches em um único. O Git compara dois ou mais conjuntos de alterações e tenta determinar onde essas alterações ocorreram.

Quando as alterações ocorrem em diferentes partes de um arquivo, o Git pode fazer um merge automático deles. Algumas vezes ocorre um conflito, que é quando existem alterações em partes iguais do código, nesse caso o Git pede que o desenvolvedor interfira no processo e faça o merge de forma manual. Desta forma, o desenvolvedor precisa analisar qual parte do

código precisa ser mantida como o código final para o repositório. O Git possui vários métodos para gerenciar conflitos.

Outra situação interessante do Git é que ele rastreia os merges, ou seja, quais commits sofreram um merge e quais não.

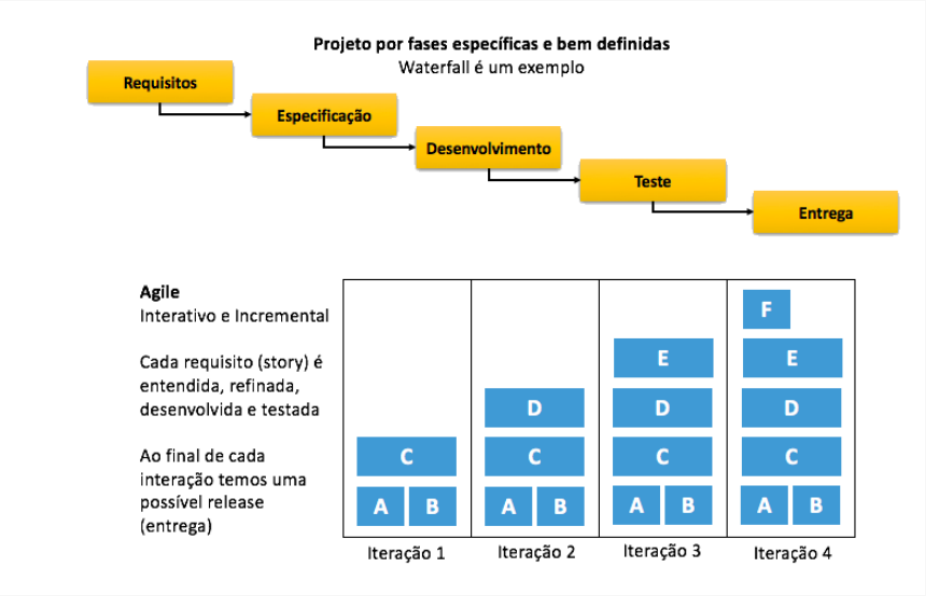
> Capítulo 4

Capítulo 4. Testes Automatizados

A abordagem de testes tradicionais considera os testes uma etapa após a etapa de desenvolvimento. Já a abordagem de testes ágeis considera os testes uma atividade conjunta ao desenvolvimento e entrega de cada iteração.

Ao final de cada pequena iteração temos uma pequena entrega que deve ser testável e entregue direto em produção através da Integração, Entrega e Implantação Contínua.

Figura 11 - Modelo waterfall vs. Modelo Ágil Interativo e Incremental.



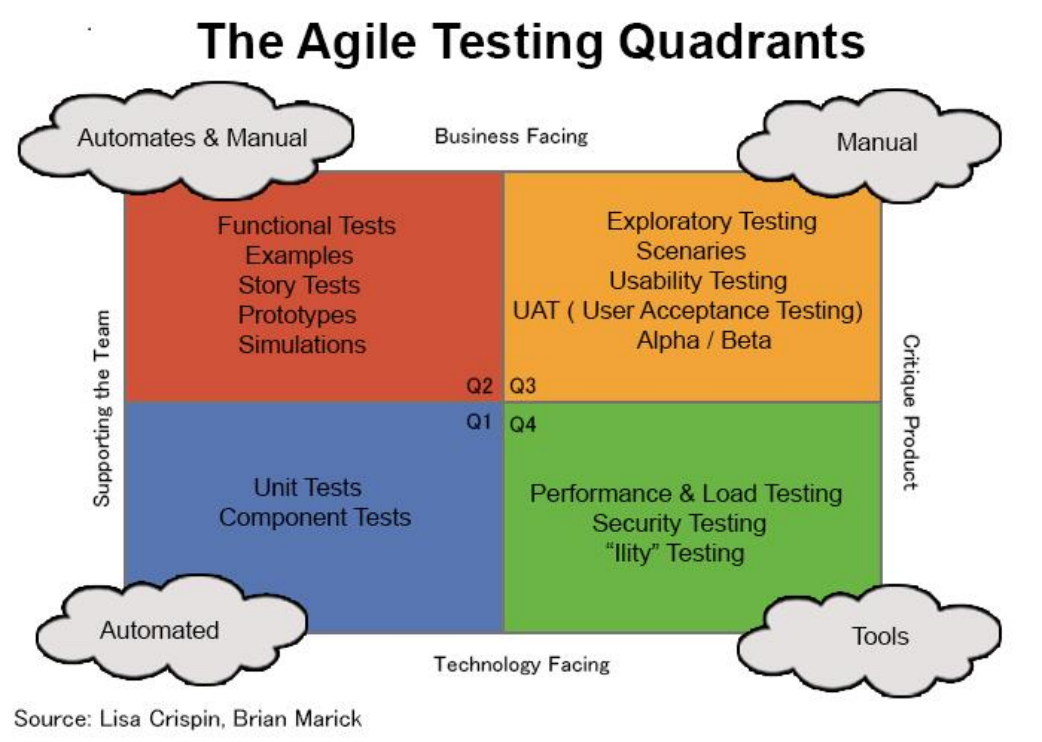
Fonte: Adaptworks.

Quadrante dos Testes Ágeis

O quadrante dos Testes Ágeis ajuda o time a pensar na estratégia de testes para cobrir novas funcionalidades, o que deve ser pensado a cada nova pequena entrega. Os testes devem sempre acompanhar a velocidade de desenvolvimento e devem ser estimados como uma atividade conjunta tanto quanto possível. Se não forem possíveis serem realizados na Sprint,

devem ser planejados n + 1, ou seja, na Sprint seguinte para que não se torne um Débito Técnico juntamente com outros problemas futuros.

Figura 12 - Quadrante dos Testes Ágeis.



Fonte: https://www.c-sharpcorner.com/blogs/testing-quadrants-in-agile.

4.1. Tipos de Testes e Níveis de Testes

Tipos de Testes:

- Smoke Tests
- Testes de Regressão
- Testes de Usabilidade
- Testes de Segurança
- Testes de Stress, Carga, Volume, Performance
- Testes de Integridade, de Dados, de Compliance

- Testes de Acessibilidade
- Testes de Confiabilidade, de Escalabilidade, de Portabilidade
- Dentre outros...

Níveis de Testes:

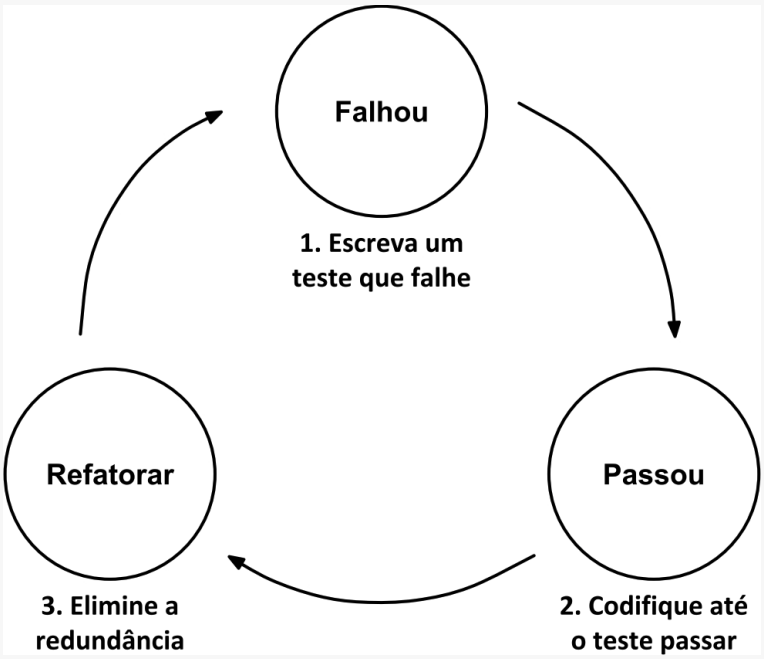
- Testes Unitários
- Testes de Integração
- Testes de Componente
- Testes de Sistema
- Testes de Aceitação
- Testes Alfa & Beta

4.2. Abordagens de Desenvolvimento Orientadas a Testes

Test Driven Development

Test Driven Development (TDD) ou Desenvolvimento orientado por testes é uma técnica de desenvolvimento de software que se baseia em um ciclo curto de repetições: Primeiramente o desenvolvedor escreve um caso de teste automatizado que define uma melhoria desejada ou uma nova funcionalidade. Então, é produzido código que pode ser validado pelo teste para posteriormente o código ser refatorado para um código sob padrões aceitáveis. Kent Beck, considerado o criador ou o 'descobridor' da técnica, declarou em 2003 que TDD encoraja designs de código simples e inspira confiança. Desenvolvimento dirigido por testes é relacionado a conceitos de programação de Extreme Programming, iniciado em 1999, mas recentemente tem-se criado maior interesse pela mesma em função de seus próprios ideais. Através de TDD, programadores podem aplicar o conceito de melhorar e depurar código legado desenvolvido a partir de técnicas antigas.

Figura 13 - Etapas do Ciclo de TDD.



Fonte: Livro Jornada Java, Cap 46. Desenvolvimento Orientado a Testes.

Passos:

1. Escreva um teste que falhe: escreva um teste de unidade para uma nova funcionalidade e execute o teste. Ele deve falhar considerando que o código ainda não foi desenvolvido;

2. Codifique até o teste passar: implemente o código mínimo ou método correspondente para o teste passar garantindo que a implementação funciona de acordo com o comportamento esperado;

3. Elimine a redundância: refatore o código o deixando limpo, legível e otimizado. Os testes já implementados vão garantir que a funcionalidade continuará cumprindo o comportamento esperado.

Behavior Driven Development

BDD é uma prática que exercita a conversação do time e que usa exemplos concretos de como o software deveria se comportar.

Um problema comum é um analista de negócio especificar e escrever o que precisa, e o time de desenvolvimento simplesmente codificar. Parece simples, mas há muita complexidade envolvida em como se espera que seja a aplicação e como ela realmente é entregue. O processo de realizar estas duas tarefas de forma separada acaba gerando vários problemas, um deles é a inviabilidade técnica de o sistema se comportar exatamente como deveria ou até mesmo de um componente ser imaginado se comportando de uma forma e acabar se comportando de outra na entrega final.

Exemplo Concretos de como o BDD nos ajuda

- Identifica e ilustra regras de negócio;
- Nos ajuda a levantar questões;
- Foca no que fazer e não em como fazer;
- Permite pensar nos entregáveis do projeto.

BDD é uma prática comum e que envolve todo o time para um completo entendimento do que será entregue. Geralmente as conversas começam com um dos integrantes citando a estória que será discutida e um outro participante perguntando ”me dê um exemplo”.

Apesar de serem técnicas similares, o BDD é descrito como uma metodologia externa, porque o ponto de partida é estabelecer o que o usuário final deseja que aconteça na prática através das iterações com o software.

Ele geralmente acontece anterior à codificação e, idealmente, deve fazer parte de toda a entrega da funcionalidade (ou feature), conforme exemplificado na figura 14 abaixo.

Figura 14 - Processo de escrita dos cenários em BDD.

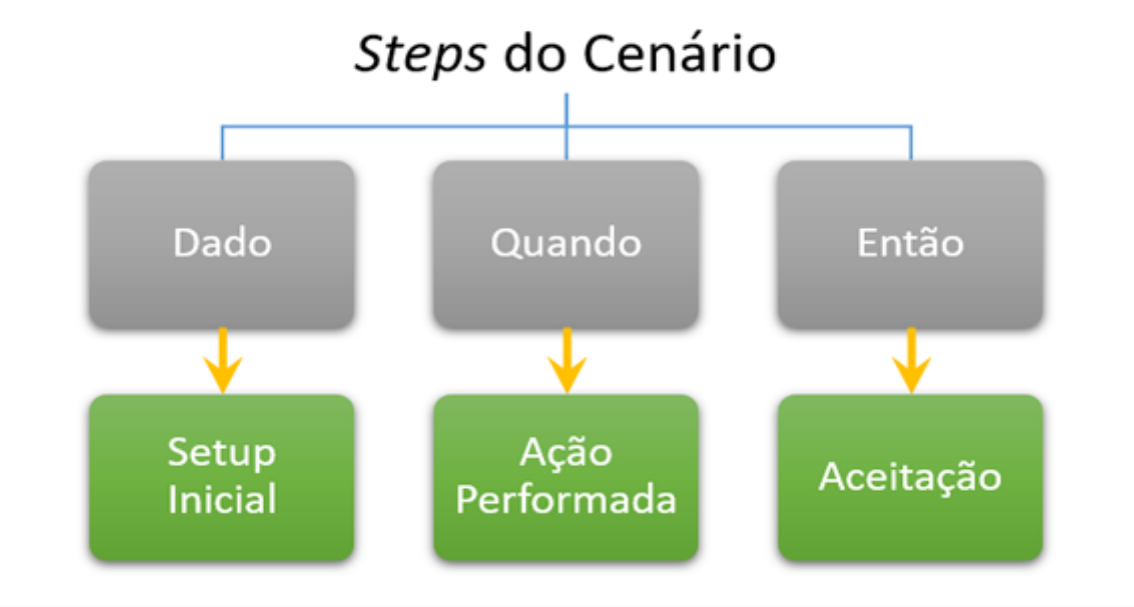
Fonte: DINESH, 2018.

Sintaxe Gherkin

> Gherkin é uma Business Readable Domain Specific Language, ou seja, uma linguagem criada especificamente para descrever comportamentos desejados de um software, removendo os detalhes lógicos dos testes de comportamento. O Gherkin pode ser utilizado como documentação do seu software e para os testes automatizados. (MUNIZ, 2019 em Jornada Ágil de Qualidade).

Gherkin é uma sintaxe projetada pelos criadores do Cucumber (framework de automação de BDD) com o objetivo de facilitar a escrita e, por consequência, a compreensão dos artefatos (features), tanto para desenvolvedores quanto para não desenvolvedores.

Figura 15 - Passos do cenário em Gherkin no BDD.



Funcionalidade: nome da funcionalidade (cada arquivo de feature geralmente corresponde a uma funcionalidade).

Exemplo de escrita:

Funcionalidade: Realizar Operação de Soma

Eu como um Contador

Quero realizar operações matemáticas

A fim de obter o resultado das operações

Contexto:

Dado que eu tenho uma calculadora

Cenário: Somar dois números

Quando a entrada "3" e "2"

E solicito a realização do cálculo de "soma"

Então o resultado é "5"

Cenário: Diminuir dois números

Quando a entrada "8" e "2"

E solicito a realização do cálculo de "subtração"

Então o resultado é "6"

Contexto: seção da feature que concentra os passos que são comuns em todos os cenários. O objetivo é deixar os cenários mais enxutos, legíveis e promover a reutilização do código caso eles sejam automatizados.

Cenário: cada cenário é um exemplo único e concreto de como o sistema deve se comportar em uma determinada situação e seu respectivo resultado esperado.

Esquema do Cenário: quando existe um conjunto de cenários e percebemos que as únicas coisas que variam são as entradas e as saídas. Um exemplo é reescrever com valores de entrada para o cenário e a saída variar em função do parâmetro enviado. Desta forma:

Esquema do Cenário: Somar dois números

Quando a entrada <entrada 1> e <entrada 2>

E solicito a realização do cálculo de <operação>

Então o resultado é <resultado>

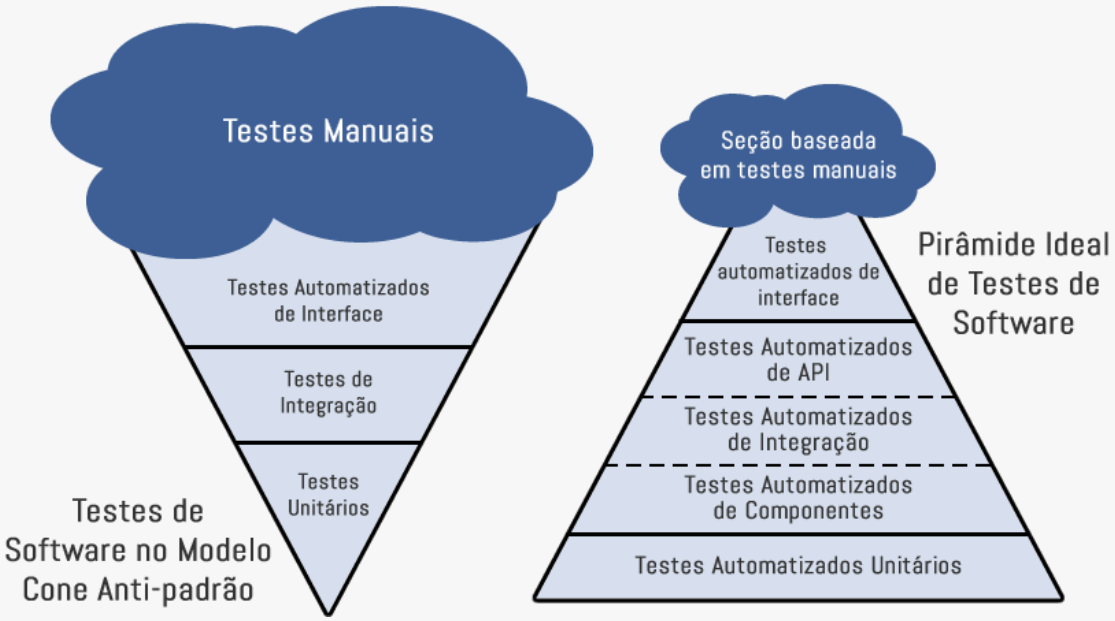
Exemplos:

entrada 1	operação	entrada 2	resultado
"3"	"+"	"2"	"5"
"8"	"-"	"2"	"6"

4.3. Pirâmide de Testes

A analogia da pirâmide é útil para trazer uma imagem da quantidade de testes esperada em cada uma dessas camadas, é como você deve pensar sua estratégia de testes: a base da pirâmide é composta em seu número maior por testes unitários, em menor número de testes devem estar os testes de integração e no topo, ou seja, em menor número devem estar os testes End-to-end (seja eles de API ou UI). Logo, os testes manuais devem ser sempre considerados, na cultura DevOps, como um menor número de testes possível, apenas aqueles que não forem realmente possíveis de serem automatizados ou que não agregam valor ao pipeline do produto.

Figura 16 - Pirâmide de Testes baseada em volume ou quantidade.



Fonte: Prime Control.

4.4. Hands-On de API

O teste de API é um processo que ajuda a garantir a qualidade do produto de software através de chamadas de API, incluindo as entradas como headers e parâmetros e as saídas como os dados de resposta e a validação dos possíveis códigos de erros de resposta. Aqui, estamos mais

inclinados a testar a precisão dos dados, códigos de status HTTP, formato de dados e códigos de erro. O que devemos considerar em um teste de API:

- Verificar o valor de retorno da API de acordo com a entrada;
- Verificar se a API está retornando a resposta errada ou nada;
- Verificar se a API está chamando alguma outra API ou invocando um evento;
- Verificar se a API está conectada às estruturas de dados ou não.

1. Teste de Funcionalidade

O teste funcional é um tipo de teste de API, que verifica os requisitos operacionais do produto. Essas funções usam cenários específicos para garantir que a API esteja funcionando de acordo com os parâmetros esperados. Os erros são corrigidos ou gerenciados se o resultado não for o esperado. Dentro desses testes podemos testar os cenários positivos, que são os comportamentos esperados e os cenários negativos.

O teste negativo verifica se sua API foi projetada para lidar com respostas inesperadas e inválidas do usuário de maneira adequada. Por exemplo, se o usuário digitar um número em um campo de letra, o que você diria ao usuário ou que mensagem exibiria? Você pode simplesmente mostrar “Resposta inválida. Por favor, digite uma letra”.

2. Teste de confiabilidade

Quando conectamos uma API a mais de um dispositivo é necessário verificar se há alguma desconexão. O teste de confiabilidade nos permite verificar exatamente isso. Por meio desse teste, você pode ver se a API funcionará sem falhas em qualquer ambiente específico e oferecerá resultados consistentes.

3. Teste de carga

Como o nome sugere, o teste de carga verifica se a API tem o poder de lidar com uma determinada quantidade de carga. Ele analisa como uma API específica se comporta sob cargas mais altas do que deveria.

Nesse tipo de teste, é interessante medir os tempos de resposta, verificar as condições severas, analisar a taxa de transferência e avaliar outros fatores semelhantes. Todo o objetivo é ver como o sistema reagiria para entender o cenário de alta carga de usuários.

4. Teste de segurança

O teste de segurança inclui as necessidades de segurança da API. Ele contém permissões, autenticações e controles de acesso.

- A autenticação de que você precisa para a API;
- A criptografia necessária para manter os dados confidenciais seguros;
- As verificações de autorização e controles em vigor para o gerenciamento de recursos.

Coletivamente, esses três revelam a estrutura de segurança de uma API.

5. Teste de documentação API

A documentação é frequentemente ignorada, mas é crucial para a equipe de desenvolvimento. O documento da API contém o procedimento para usar a API. Cada requisito complexo, necessidade técnica e necessidade de engenharia são descritos na documentação da API. Entender se este documento pode orientar o usuário a extrair valor da API ou não é o objetivo real do teste de documentação da API.

O que é REST?

REST significa Transferência de Estado Representacional. É um estilo arquitetônico de software que possui regras (restrições), que são necessárias para serem seguidas pelos desenvolvedores. No entanto, uma das restrições mais vitais é que o aplicativo web deve ser capaz de fornecer os dados (informações) sempre que um comando é dado.

Métodos HTTP para testes de automação de API REST

A API REST usa cinco métodos HTTP para solicitar um comando:

- GET: Para recuperar as informações em uma URL específica.
- PUT: Para atualizar o recurso anterior ou criar novas informações em uma URL específica.
- PATCH: Para atualizações parciais.
- POST: É usado para desenvolver uma nova entidade. Além disso, ele também é usado para enviar informações para servidores, como upload de um arquivo, informações do cliente etc.
- DELETE: Excluir todas as representações atuais em uma URL específica.

Códigos de status HTTP

Códigos de status são a resposta dada por um servidor à solicitação de um cliente. Eles são classificados em cinco categorias:

- 1xx (100 – 199): A resposta é informacional;
- 2xx (200 – 299): Garante uma resposta bem sucedida;

- 3xx (300 – 399): Você é obrigado a tomar mais medidas para atender ao pedido;
- 4xx (400 – 499): Há uma sintaxe ruim e a solicitação não pode ser concluída;
- 5xx (500 – 599): O servidor não completa totalmente a solicitação.

Esses códigos ajudam a interpretar os resultados. Assim, se os resultados dos testes de automação de API REST estiverem entre as faixas 2xx, isso significa que as funções do aplicativo estão funcionando idealmente. Para a maioria dos usuários normais e cotidianos da Internet, os códigos de status nunca serão algo que eles vão encontrar ou ir à procura. Para códigos de status 1xx, 2xx e 3xx, estes realmente não são considerados erros, uma mensagem bastante informativa e não necessariamente afetarão a experiência do usuário.

No entanto, quando começamos a entrar nos códigos de status 4xx e 5xx, essas são consideradas mensagens de erro e quando algo dá errado, os usuários serão confrontados com mensagens de erro enquanto navegam através de APIs. As mensagens de erro de código de status 4xx geralmente ocorrem quando algo acontece no nível cliente/navegador. As mensagens de erro de código de status de 5xx resultam em erros no nível do servidor. Embora nunca seja bom ver erros, estes são especialmente importantes para remediar o mais rápido possível, pois indicam problemas sérios e afetarão muito a satisfação do usuário.

Para obter mais informações sobre códigos de status HTTP, juntamente com uma lista abrangente de todos os códigos e mensagens de erro diferentes, você pode obter diretamente aqui no Códigos de status de respostas HTTP do MDN Web Docs community.

Na prática:

Instalar Insomnia: https://insomnia.rest/download

Vamos empregar como aplicação-base para o nosso hands-on a ferramenta criada pelo profissional Paulo Gonçalves a fim de utilizá-la para a comunidade para fins de estudos acerca de automação:

- https://serverest.dev/

Os demais passos estão nos slides e nas videoaulas.

Referências:

- Post Entendendo e colocando a mão na massa com Postman
- https://jsonplaceholder.typicode.com/
- 10 Ferramentas de Testes de API

4.5. Hands-On de UI (Interface Gráfica)

Testes de UI ou de Interface gráfica são testes que simulam a navegação do usuário por cima do browser. O objetivo é garantir que todos os itens da interface gráfica estão funcionando corretamente e a interação do usuário com a aplicação ou aplicativo está funcionando com o comportamento esperado. Esse tipo de teste também é chamado muitas vezes de End-2-End, ou seja, ele serve para testar uma aplicação de ponta a ponta, para validar cenários reais executados pelo usuário final da aplicação. Os frameworks de testes mais utilizados usam como base o Selenium para a identificação dos elementos de tela do browser, apesar de algumas ferramentas mais atuais não utilizarem mais o Selenium. Existem vantagens e desvantagens para o uso do Selenium.

Cada linguagem de programação tem seu próprio conjunto de frameworks de testes para aplicação de testes de UI. Um exemplo disso, em

Java, pode-se utilizar diretamente o JUnit como framework-base juntamente com o Selenium, ou o TestNG diretamente com o Selenium. Mas, se você for aplicar BDD, você pode utilizar outras configurações como:

- TestNG + Cucumber + Selenium
- JUnit + Cucumber + Selenium
- SerenityBDD + (JUnit ou TestNG) + Cucumber + Selenium

Há também a possibilidade de implementar testes híbridos, ou seja, de UI com API. Em Java poderíamos utilizar a configuração:

- TestNG + SerenityBDD + SerenityRest + RestAssured + (JUnit ou TestNG) + Cucumber + Selenium

Já se você for utilizar Javascript existe uma diversidade muito grande de frameworks. Alguns deles, mais famosos, são: webdriverio, cypress.io, codecept.js, dentre outros.

Vamos empregar como aplicação-base para o nosso hands-on a ferramenta criada pelo profissional Paulo Gonçalves a fim de utilizá-la para a comunidade para fins de estudos acerca de automação:

- https://www.saucedemo.com/inventory.html

Também existe o aplicativo do cypress para estudo desta ferramenta de automação: https://github.com/cypress-io/cypress-realworld-app

Os demais passos estão nos slides e nas videoaulas.

> Capítulo 5

Capítulo 5. Estratégias de Branch e Release

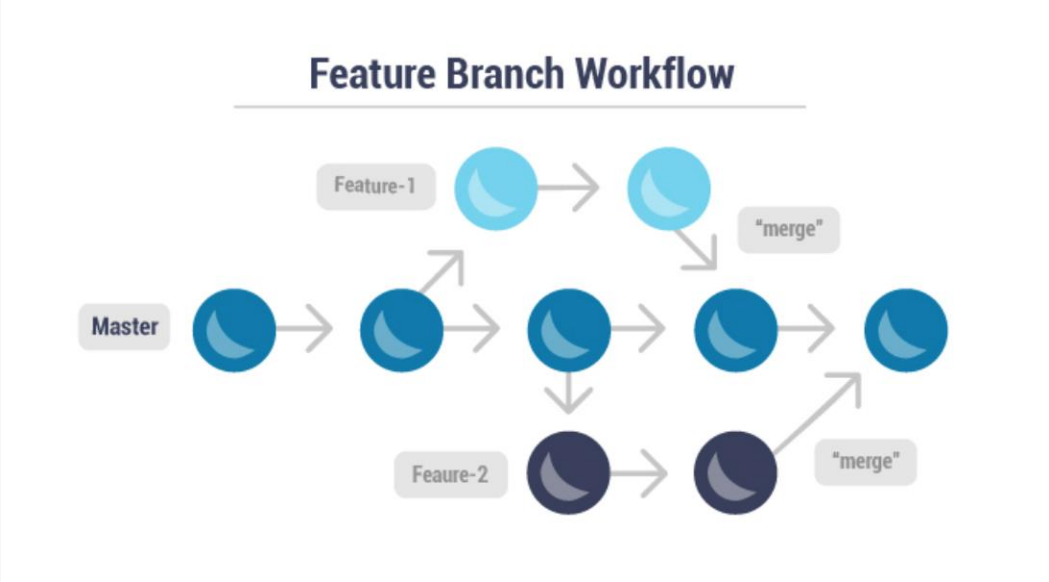
No processo de desenvolvimento, uma das definições mais críticas para o gerenciamento do versionamento do código são as estratégias de branch e release. Como manter o código consistente? Como manter um histórico das alterações feitas pelo time e como mesclar as alterações feitas pelos membros da equipe?

5.1. Estratégias de Branch: Feature Branch, Forking workflow, Gitflow, Direto no Trunk

Feature Branch Model

Nessa abordagem, os desenvolvedores devem criar uma nova ramificação do código para cada feature a ser implementada. Com isso, o merge entre a feature e o ramo principal só acontecem quando aquela feature já tiver seu desenvolvimento finalizado. Assim, novas features nunca estarão no ramo master sem que a mesma esteja finalizada.

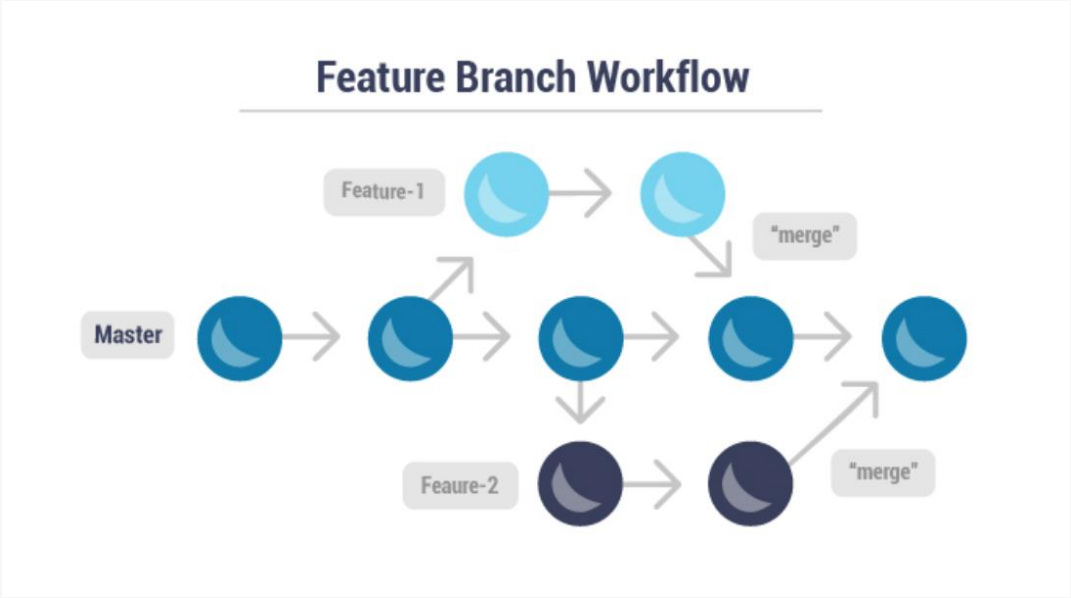
Figura 17 - Modelo de ramificação por recurso.



Fonte: https://blog.enacom.com.br/2019/01/07/controle-de-versao-e-gerenciamento-de-codigo/.

Nessa abordagem, os desenvolvedores devem criar uma nova ramificação do código para cada feature a ser implementada. Com isso, o merge entre a feature e o ramo principal só acontecem quando aquela feature já tiver seu desenvolvimento finalizado. Assim, novas features nunca estarão no ramo master sem que a mesma esteja finalizada.

Figura 18 - Modelo de ramificação por recurso.



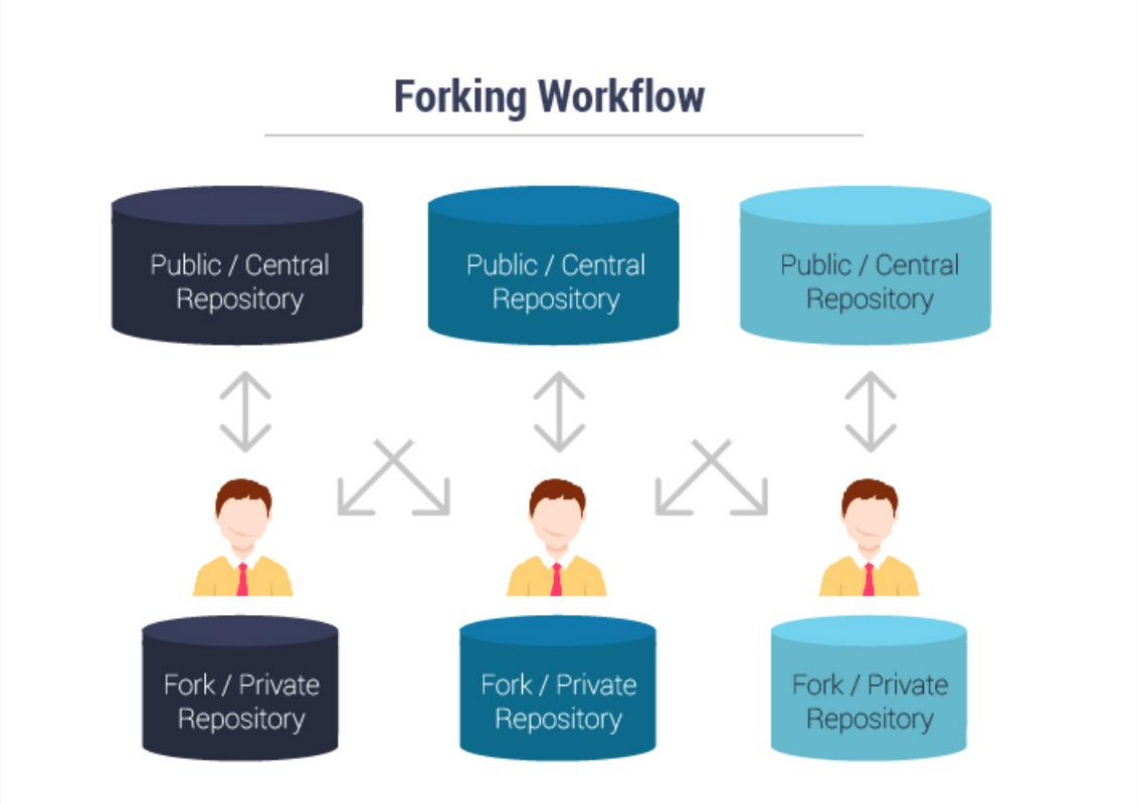
Fonte: https://blog.enacom.com.br/2019/01/07/controle-de-versao-e-gerenciamento-de-codigo/.

Contudo, o modelo de ramificação por recurso ainda não evita um problema importante. Caso seja identificado um bug no código que está em produção, não é possível realizar correções no mesmo código e gerar uma nova versão do sistema sem que as features desenvolvidas no intervalo entre a última release e a identificação do bug estejam presentes. Para corrigir o bug e gerar uma nova versão do sistema consistente, será necessário retroceder o código até a última versão, corrigir o problema e criar uma nova release a partir dessa correção.

Forking Workflow

Nesse modelo, cada desenvolvedor possui seu próprio repositório e as modificações são realizadas somente nesse repositório. Para integrar o seu código com o repositório central, o desenvolvedor deve realizar o processo chamado de Pull Request. Isso faz com que os desenvolvedores integrem o código produzido sem a necessidade de realizar as modificações direto no repositório central.

Figura 19 - Fluxo de trabalho de bifurcação.



Fonte: https://blog.enacom.com.br/2019/01/07/controle-de-versao-e-gerenciamento-de-codigo/.

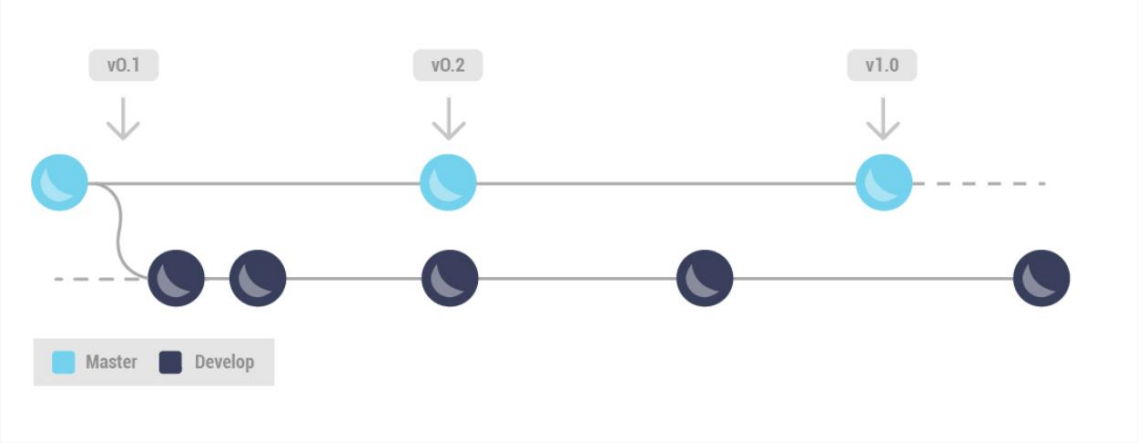
Entretanto, apenas o “gestor” do repositório poderá aceitar o Pull Request, fazendo com que o código seja realmente integrado. Isso faz com que os desenvolvedores tenham a capacidade de contribuir com o projeto, sem a necessidade de ter acesso ao repositório oficial. Essas vantagens acabam gerando a necessidade obrigatória da figura do “gestor”, que deverá

revisar e aceitar os Pull Requests. O “gestor” normalmente é o desenvolvedor com maior conhecimento e capacidade técnica e deverá ficar destinado quase que integralmente a esse processo de análise, revisão e aceitação.

Gitflow Workflow

No Gitflow Workflow existem dois tipos de ramificações: ramificações principais e ramificações de suporte. As ramificações principais são chamadas de Develop e Master. A ramificação Master é onde se encontra o código correspondente ao software de produção. A ramificação Develop é um ramo criado a partir do ramo Master e é nesse ramo onde todos os desenvolvedores trabalham.

Figura 20 - Modelo estrito de ramificação projetado em torno do lançamento do projeto.



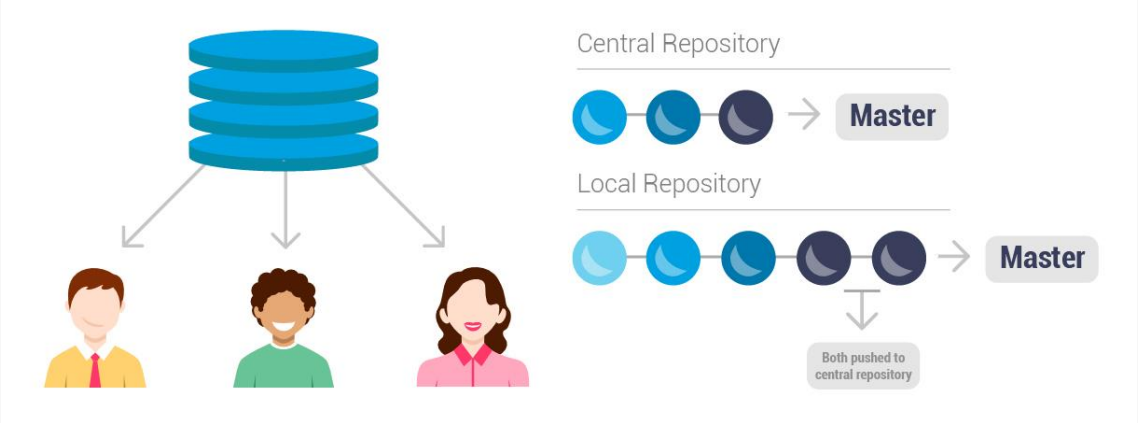
Fonte: https://blog.enacom.com.br/2019/01/07/controle-de-versao-e-gerenciamento-de-codigo/.

As ramificações de suporte podem ser do tipo Feature, Release ou Hotfix. Ao iniciar o desenvolvimento de uma nova funcionalidade, o desenvolvedor deve criar uma ramificação do tipo Feature a partir do ramo Develop. Ele irá implementar a nova funcionalidade e mesclar o código produzido ao ramo Develop apenas quando finalizar a funcionalidade. Isso faz com que o ramo Develop tenha apenas funcionalidades finalizadas.

Direto no Trunk (ou Fluxo de Trabalho Centralizado)

Nessa abordagem nenhuma ramificação (branch) do código é criada e os desenvolvedores trabalham diretamente no único ramo existente, o master. Como vantagem, esse fluxo de trabalho não necessita que a equipe tenha que gerenciar vários ramos de código. Como desvantagem, aumenta a probabilidade de existirem conflitos no código, já que todos os desenvolvedores estão utilizando o mesmo ramo de código, o que pode inserir bugs no sistema.

Figura 21 - Modelo Centralizado de gerenciamento de código.



Fonte: https://blog.enacom.com.br/2019/01/07/controle-de-versao-e-gerenciamento-de-codigo/.

Outra desvantagem do fluxo de trabalho centralizado é observada quando há a necessidade de criar uma nova release em produção e existem novas features no repositório que ainda não foram finalizadas e não irão compor a release. Nesse caso, será necessário adotar medidas para impedir que a feature não finalizada não faça parte da release.

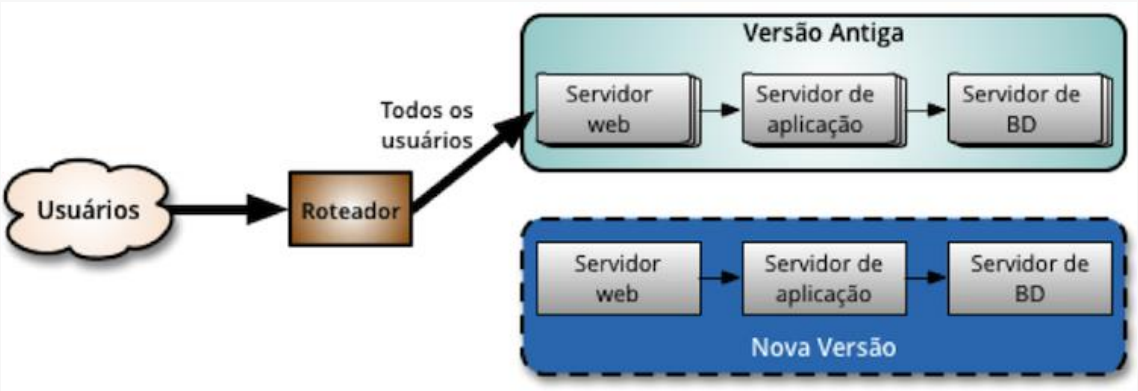
5.2 .Release Canário, Azul / Verde

Release Azul / Verde

Um dos desafios da implantação automatizada é a própria transição, levando o software do estágio final de teste para a produção. Geralmente, você precisa fazer isso rapidamente para minimizar o tempo de inatividade.

A abordagem de implantação azul-verde faz isso garantindo que você tenha dois ambientes de produção, o mais idênticos possível. A qualquer momento um deles, digamos o verde por exemplo, está on-line. À medida que você prepara uma nova versão do seu software, você faz seu estágio final de teste no ambiente azul. Uma vez que o software está funcionando no ambiente azul, você alterna o roteador para que todas as solicitações de entrada vão para o ambiente azul - o verde agora está ocioso.

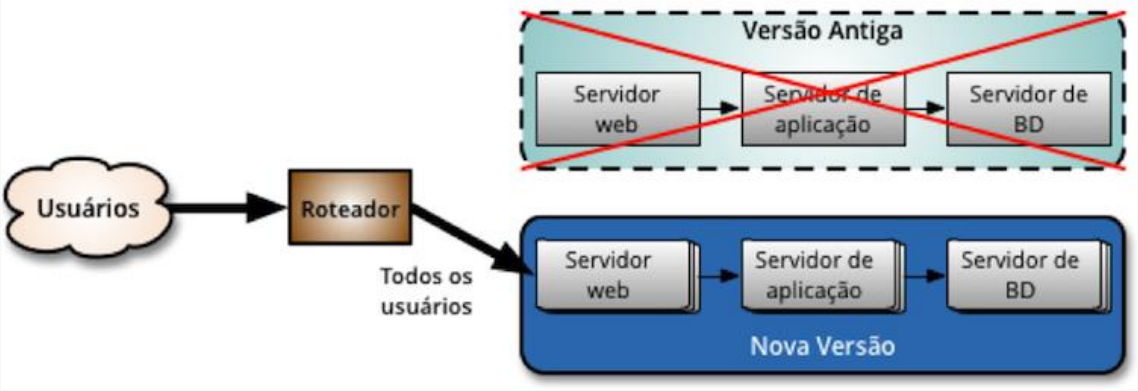
Figura 22 - Release Blue/Green: Antes.



Fonte: https://www.thoughtworks.com/pt-br/insights/blog/implanta%C3%A7%C3%B5es-can%C3%A1rio.

A implantação azul-verde também oferece uma maneira rápida de reverter – se algo der errado, você muda o roteador de volta para o ambiente verde. Ainda há o problema de lidar com transações perdidas enquanto o ambiente azul ainda não estava ativo, mas dependendo do seu design, você pode alimentar transações para ambos os ambientes de forma a manter o ambiente verde como backup enquanto o azul ainda não estiver ativo. Ou você pode colocar o aplicativo no modo somente leitura antes da transição, executá-lo por um tempo no modo somente leitura e, em seguida, alterná-lo para o modo leitura-gravação. Isso pode ser suficiente para eliminar muitas questões pendentes.

Figura 23 - Release Blue/Green: Depois.



Fonte: https://www.thoughtworks.com/pt-br/insights/blog/implanta%C3%A7%C3%B5es-can%C3%A1rio.

Os dois ambientes precisam ser diferentes, mas o mais idênticos possível. Em algumas situações, eles podem ser peças diferentes de hardware ou podem ser máquinas virtuais diferentes em execução no mesmo (ou em hardware diferente). Eles também podem ser um único ambiente operacional particionado em zonas separadas com endereços IP separados para as duas fatias.

Depois de ativar seu ambiente azul e ficar satisfeito com sua estabilidade, use o ambiente verde como seu ambiente de teste para a etapa final de teste para sua próxima implantação. Quando estiver pronto para sua próxima versão, você alternará de azul para verde da mesma forma que fez de verde para azul anteriormente. Dessa forma, os ambientes verde e azul alternam regularmente entre a versão anterior on-line (para reversão) e a preparação da próxima versão.

Os bancos de dados geralmente podem ser um desafio com essa técnica, principalmente quando você precisa alterar o esquema para oferecer suporte a uma nova versão do software. O truque é separar a implantação de alterações de esquema de atualizações de aplicativos. Portanto, primeiro aplique uma refatoração de banco de dados para alterar o esquema para oferecer suporte à versão nova e antiga do aplicativo, implante isso, verifique se tudo está funcionando bem para que você tenha

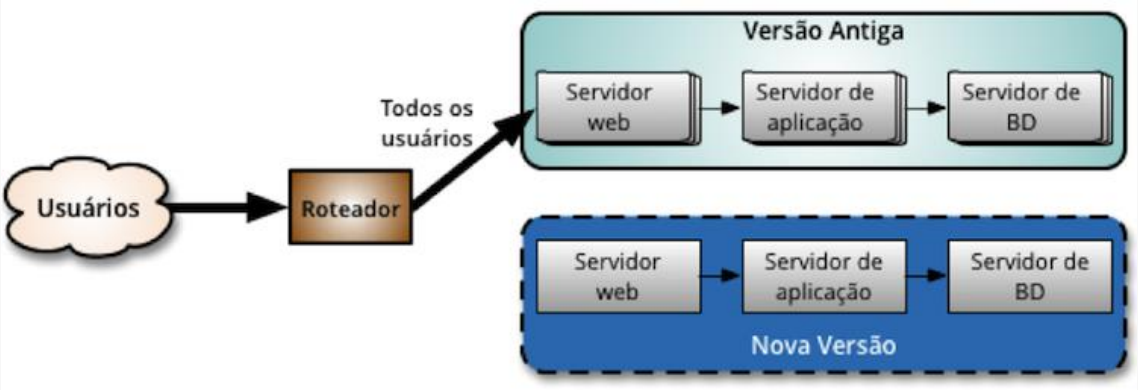
um ponto de reversão e, em seguida, implante a nova versão do aplicativo. (E quando a atualização terminar, remova o suporte de banco de dados para a versão antiga).

Release Canário

Implantação Canário (ou Canary Release) é uma técnica para reduzir o risco da introdução de uma nova versão do software em produção, fazendo o lançamento gradual da mesma para uma pequena parte do conjunto de usuários antes de implantá-la em toda a infraestrutura e torná-la acessível a todos.

De forma similar às Implantações Azul-Verde, você começa com a implantação da nova versão do seu software em uma parte da infraestrutura para a qual nenhum usuário é roteado.

Figura 24 - Release Canário: ANTES.

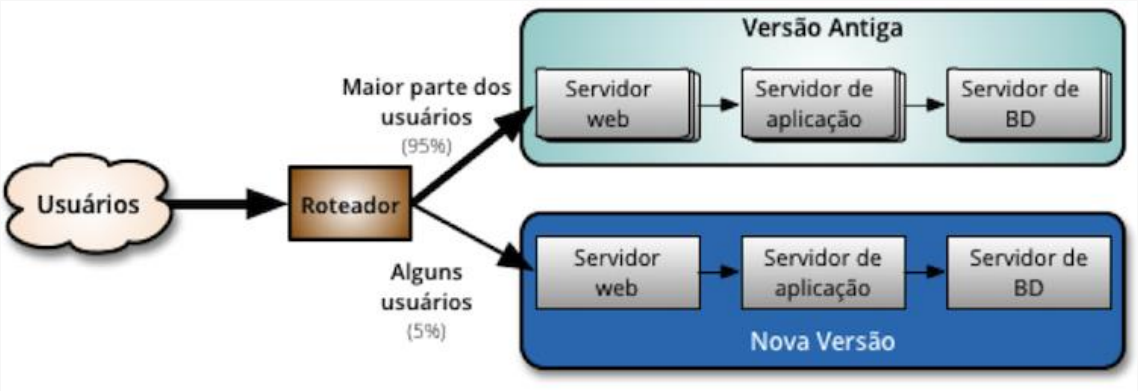


Fonte: https://www.thoughtworks.com/pt-br/insights/blog/implanta%C3%A7%C3%B5es-can%C3%A1rio.

Quando você estiver satisfeito com a nova versão, você pode começar a rotear alguns usuários selecionados para ela. Existem diversas estratégias para escolher quais são os usuários que verão a nova versão: uma estratégia simples é usar uma amostra aleatória; algumas empresas optam por liberar a nova versão para seus usuários internos e funcionários antes de liberá-la para seu público geral; outra abordagem mais sofisticada

é escolher os usuários com base em seu perfil ou outros dados demográficos.

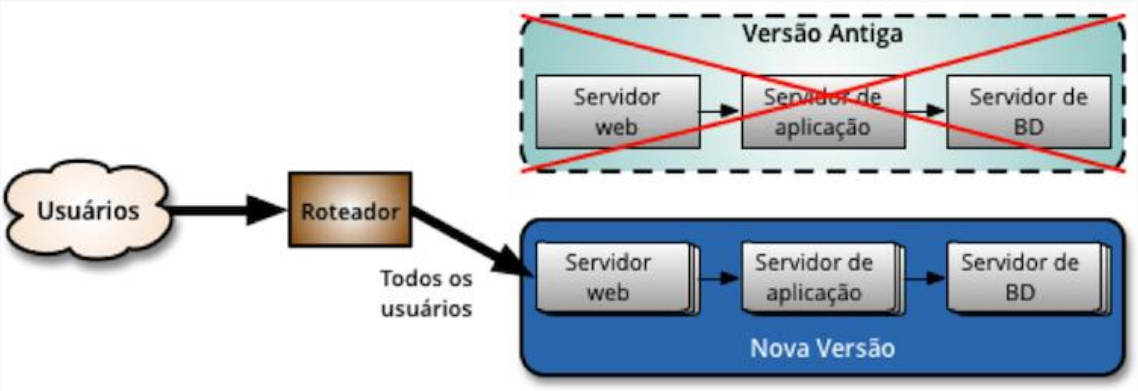
Figura 25 - Release Canário: DEPOIS.



Fonte: https://www.thoughtworks.com/pt-br/insights/blog/implanta%C3%A7%C3%B5es-can%C3%A1rio.

A Implantação Canário é uma aplicação do padrão Mudança Paralela, na qual a fase de migração dura até que todos os usuários sejam direcionados para a nova versão. Nesse momento, você pode desativar a infraestrutura antiga. Se você encontrar qualquer problema na nova versão, a estratégia de reversão é simples: basta redirecionar os usuários de volta para a versão antiga até que o problema seja resolvido.

Figura 26 - Release Canário: DEPOIS.



Fonte: https://www.thoughtworks.com/pt-br/insights/blog/implanta%C3%A7%C3%B5es-can%C3%A1rio.

Uma vantagem de usar Implantações Canário é a possibilidade de realizar testes de carga na nova versão em um ambiente de produção com uma estratégia de reversão segura se problemas forem encontrados. Incrementando lentamente a carga, você pode capturar e monitorar métricas de como a nova versão impacta o ambiente de produção. Esta é uma abordagem alternativa à criação de um ambiente para testes de performance totalmente separado, pois o ambiente será o mais próximo possível do ambiente de produção.

Apesar do nome para esta técnica não soar familiar, a prática da Implantação Canário vem sendo adotada há algum tempo. É também conhecida como lançamento em fases (ou phased rollout) ou lançamento incremental (ou incremental rollout).

5.3 .Arquitetura Monolítica / Microsserviços

Aplicações monolíticas são aquelas desenvolvidas para ser instaladas num só lugar, geralmente possuem camadas sobrepostas com responsabilidades distintas e quando é preciso realizar algum ajuste todo o sistema precisa ser atualizado no servidor de produção.

Vários autores atribuem o conceito de Microsserviços como uma ramificação do padrão de Arquitetura Orientada a Serviço (SOA), cujo objetivo é disponibilizar as funcionalidades das aplicações na forma de serviços baseando-se na coleção de pequenos serviços desacoplados que se comunicam geralmente através de HTTP. Na arquitetura de microsserviços a aplicação é distribuída em diversas subaplicações independentes e desacopladas (serviços), cada uma responsável por uma funcionalidade ou módulo da aplicação.

Escolha uma arquitetura monolítica:

- Se seu time não tem experiência alguma com microsserviços.

- Um sistema monolítico é mais fácil de entender, possui uma curva de aprendizado menor e, obviamente, existem mais desenvolvedores familiarizados com este modelo.

- Se você está criando algo com nível alto de incerteza, tipo um MVP ou uma PoC para validar um negócio ou produto.

- Se seu projeto for de curto prazo para entrega e você não tem tempo para gastar com automatização de deploy e orquestração dos serviços

Após a criação, se for preciso escalar:

Você precisará pensar em duplicar a infraestrutura (e colocar os servidores embaixo de um load balancer ou simplesmente aumentar o tamanho do servidor) ou trabalhar na arquitetura e quebrar a aplicação em microsserviços.

Escolha uma arquitetura de microsserviços:

- Se escalabilidade é um requisito fundamental;

- Se você não tem um prazo apertado;

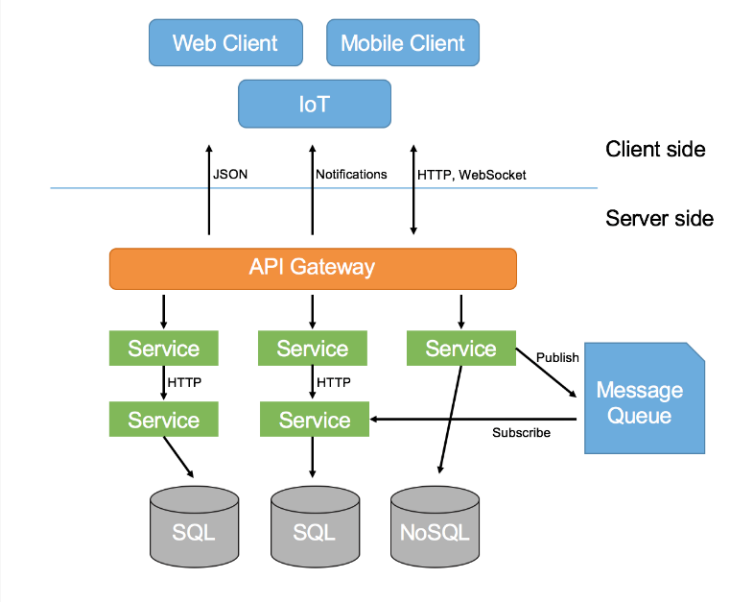
- Se independência e velocidade de release dos serviços são importantes;

- Se uma parte da aplicação precisa ser extremamente eficiente;

- Se você tem um time experiente e capaz de desenvolver em várias linguagens;

- Se você tem profissionais com experiência em infraestrutura como código e CI/CD na equipe ou alguém que possa exercer este papel.

Figura 27 - Exemplo de Arquitetura Orientada a Serviços (SOA):

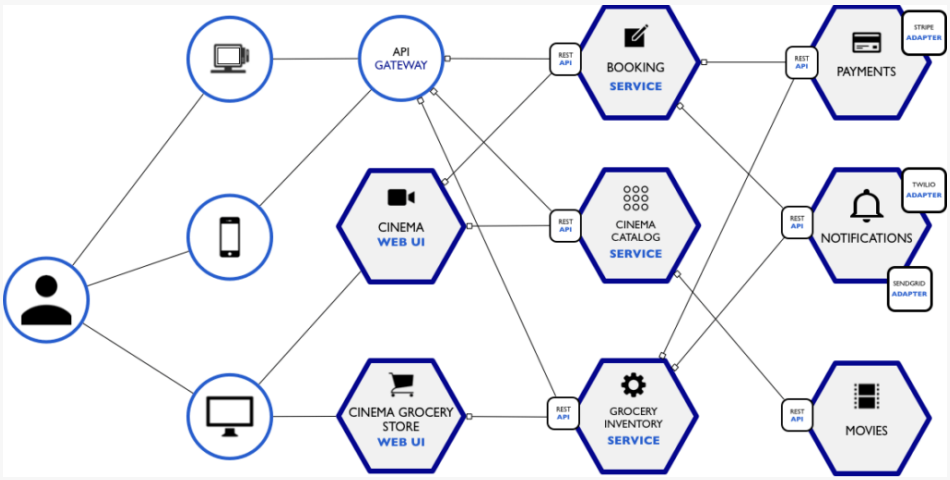


Fonte: https://blog.grancursosonline.com.br/ti-em-foco-microsservicos/.

Os microsserviços permitem o desenvolvimento paralelo, numa mesma equipe ou separando as demandas em times distintos. Neste caso, é quase impossível ser produtivo se não houver uma boa automatização de deploy, pois os microsserviços são frequentemente implantados em suas próprias máquinas virtuais ou contêineres, o que é muito trabalhoso para gerenciar manualmente.

Além disso, você pode precisar de API Gateway para orquestrar os microsserviços e também para implementar uma camada de segurança para acessos externos aos microserviços, permitindo a instalação de ferramentas que bloqueiam ataques cibernéticos coordenados ou não, como mostra na figura a seguir:

Figura 28 - Exemplo de Arquitetura de um Cinema com API Gateway.



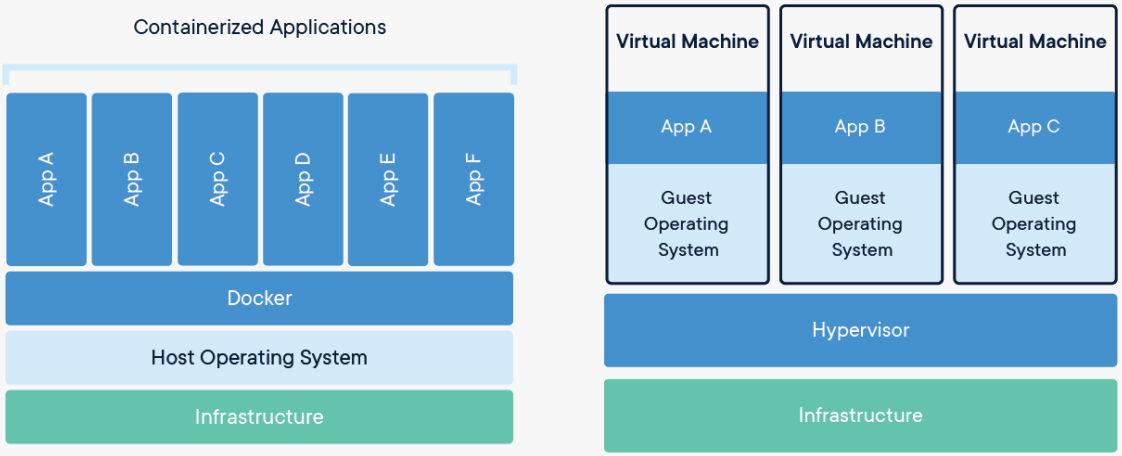
Fonte: https://www.luiztools.com.br/post/arquitetura-de-micro-servicos-em-node-js-mongodb/.

> Capítulo 6

Capítulo 6. Docker e Infraestrutura as a Service (IaaS)

Docker é uma plataforma voltada para a criação e execução de containers. Diferente de uma VM, um container empacota somente o código, dependências e variáveis de ambiente necessárias para a execução de uma aplicação. Sendo assim, ao invés de termos uma infraestrutura onde cada VM tem seu próprio SO sendo responsável por uma aplicação diferente, temos uma infraestrutura onde, com apenas um sistema operacional e através do Docker, podemos gerenciar diversas aplicações. Cada uma dessas aplicações é executada de forma isolada, sem a necessidade de um sistema operacional próprio.

Figura 29 – Containers x VMs.



Fonte: https://medium.com/cwi-software/primeiros-passos-com-docker-conceitos-b%C3%A1sicos-%C3%A0-cria%C3%A7%C3%A3o-de-sua-primeira-imagem-f6ac2a3c9d25.

6.1. VM's e Conteinerização

Máquina Virtual (VM)

Ter uma Máquina Virtual significa que um disco rígido virtual foi criado dentro de um ambiente físico por meio de um software específico e

opera independentemente dele. Ter um hardware dentro de outro possibilita rodar sistemas operacionais e todo o tipo de programas.

Esta é uma das grandes vantagens de ter uma Máquina Virtual, uma vez que você pode usar as aplicações e os programas que precisar sem depender do sistema operacional que tem no disco físico. Como você pode criar um ambiente virtual diferente dentro daquele que já existe, fica livre para usar todos os programas de que precisa em uma mesma máquina.

Container

Os Containers são tipos de ambiente nos quais é possível agrupar aplicações e seus elementos, já que os recursos utilizados por cada Container são totalmente isolados.

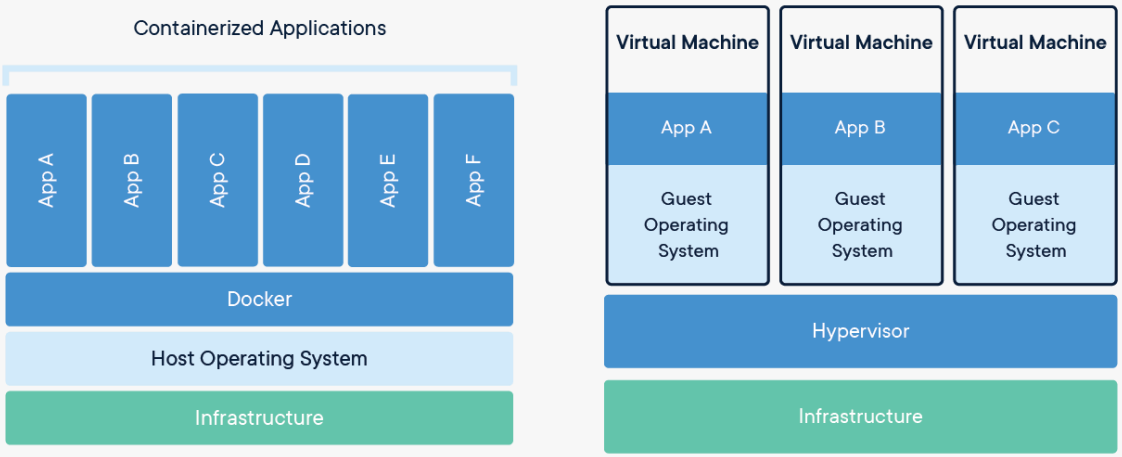
Tanto Container quanto a Máquina Virtual utilizam a virtualização. O que muda é que virtualização se dá no nível do sistema operacional e os Containers não utilizam hypervisor como as Máquinas Virtuais, e sim os recursos do sistema e processos de kernel para criar os ambientes. Logo, o Container não tem uma visão geral do ambiente físico fora de seu espaço, como a Máquina Virtual tem.

Na prática, o Container promove a comunicação do hardware para o sistema operacional e diretamente para outros containers, que cuidam do isolamento e da inicialização das aplicações. Na Máquina Virtual o hardware se comunica com o hypervisor e depois com a máquina, ou seja, haverá inicialização do sistema operacional e, só então, as aplicações estarão ativadas.

Uma das principais vantagens do Container é a possibilidade de criar serviços e códigos independentes, que podem ser movidos sem dificuldade entre máquinas e ambientes diferentes sem a perda de dados.

Quando consideramos a segurança, o Container é um pouco menos robusto que a Máquina Virtual, que ainda garante mais proteção ao usuário, mas, em compensação, sua operação tende a ser mais rápida, uma vez que só as aplicações precisam ser iniciadas e não todo o sistema operacional.

Figura 30 – Containers vs. VMs.



Fonte: https://medium.com/cwi-software/primeiros-passos-com-docker-conceitos-b%C3%A1sicos-%C3%A0-cria%C3%A7%C3%A3o-de-sua-primeira-imagem-f6ac2a3c9d25.

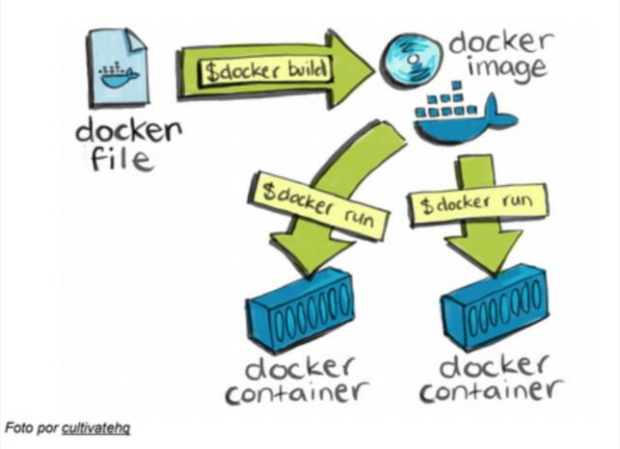
Docker

Docker permite automatizar a implantação de aplicativos no ambiente do contêiner. Para isso, a pessoa usuária define as etapas de criação de um contêiner como instruções em um Dockerfile e Docker usa ele para construir uma imagem.

As imagens definem o software disponível nos contêineres. Uma imagem do Docker contém código de aplicativo, bibliotecas, ferramentas, dependências e outros arquivos necessários para executar um aplicativo. Quando alguém executa uma imagem, ela pode se tornar uma ou várias instâncias de um contêiner.

Em outras palavras, se você criar uma imagem, qualquer user do Docker poderá iniciar seu aplicativo através de um comando (docker run).

Figura 31 – Começando com Docker.



Fonte: https://cultivatehq.com/posts/docker/.

Para gerenciar os containers é comum usar plataformas de orquestração como o Kubernetes, Docker Swarm ou Apache Mesos para usar Docker em produção. Essas ferramentas são projetadas para lidar com várias réplicas de contêiner, o que melhora a escalabilidade e a confiabilidade. A orquestração pode ser usada em todos os ambientes que executam containers, incluindo servidores on-premise ou nuvens públicas e privadas.

Kubernetes

O Kubernetes é uma ferramenta open source de orquestração de containers projetada e desenvolvida originalmente por engenheiros do Google. Em 2015, o Google doou o projeto Kubernetes à recém-formada Cloud Native Computing Foundation.

Com a capacidade de orquestração do Kubernetes, é possível criar serviços de aplicações que abrangem múltiplos containers, programar o uso dos containers no cluster, escalá-los e gerenciar a integridade deles ao longo do tempo.

O Kubernetes elimina grande parte dos processos manuais necessários para implantar e escalar aplicações em container. É possível

agrupar em clusters os hosts, sejam eles físicos ou máquinas virtuais, executados nos containers Linux.

Em termos mais abrangentes, com o Kubernetes é mais fácil implementar e confiar totalmente em uma infraestrutura baseada em containers para os ambientes de produção.

Esses clusters podem abranger hosts em nuvens públicas, privadas ou híbridas. Por isso, o Kubernetes é a plataforma ideal para hospedar aplicações nativas em nuvem que exigem escalabilidade rápida.

O Kubernetes também ajuda com o balanceamento de carga e a portabilidade de cargas de trabalho, possibilitando a migração de aplicações sem precisar recriá-las.

Principais componentes do Kubernetes:

- Cluster: plano de controle e pelo menos uma máquina de computação ou nós.
- Plano de controle: conjunto de processos que controlam os nós do Kubernetes. É nele que todas as atribuições de tarefas se originam.
- Kubelet: um serviço executado nos nós que lê os manifestos do container e assegura que os containers definidos sejam iniciados e executados.
- Pod: um grupo de um ou mais containers implantados em um nó. Todos os containers em um pod têm o mesmo endereço IP, IPC, nome de host e outros recursos.

Como a orquestração de containers funciona?

Ao usar uma ferramenta de orquestração de containers, como o Kubernetes, você define a configuração de uma aplicação usando um

arquivo JSON ou YAML. Esse arquivo informa à ferramenta de gerenciamento de configurações o local das imagens do container, como estabelecer uma rede e onde armazenar os registros.

Quando você implanta um novo container, a ferramenta de gerenciamento de containers programa automaticamente esse processo em um cluster e atribui o host adequado, levando em consideração todas as restrições ou requisitos definidos. Depois, a ferramenta de orquestração gerencia o ciclo de vida do container com base nas especificações determinadas no arquivo de composição.

É possível usar os padrões do Kubernetes para gerenciar a configuração, o ciclo de vida e a escalabilidade dos serviços e aplicações baseadas em container. Esses padrões reproduzíveis são as ferramentas necessárias para que os desenvolvedores do Kubernetes criem sistemas completos.

A orquestração pode ser usada em todos os ambientes que executam contêineres, incluindo servidores on-premise ou nuvens públicas e privadas.

A finalidade da orquestração de containers é automatizar e gerenciar tarefas como:

- Provisionamento e implantação.
- Configuração e programação.
- Alocação de recursos.
- Disponibilidade dos containers.
- Escala ou remoção de containers com base no balanceamento de cargas de trabalho na infraestrutura.

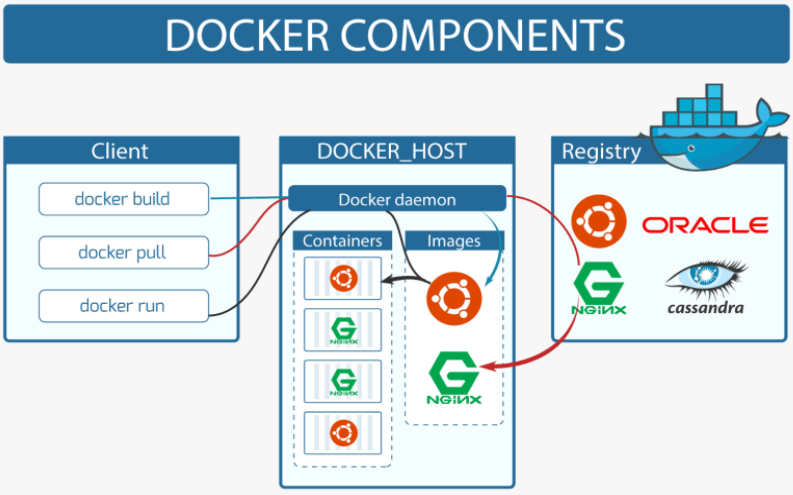
- Balanceamento de carga e roteamento de tráfego.
- Monitoramento da integridade do container.
- Configuração da aplicação com base no container em que ela será executada.
- Proteção das interações entre os containers.

6.2. Conhecendo o Dockerhub e primeiros passos com Docker

O Docker Hub, ou Docker Store, é onde nós colocamos nossas imagens e elas ficam disponíveis para download. Para compartilhar minhas imagens com outros desenvolvedores no Dockerhub eu preciso primeiro criar um repositório de imagens local.

Para um container ser criado, ele necessita de uma imagem: o básico para a aplicação ser executada, desde bibliotecas até variáveis de ambiente. Essas imagens são criadas através de um Dockerfile e podem ser baixadas e publicadas em diversos *registries*, sendo o mais famoso deles o Docker Hub.

Figura 32 – Resumo do Fluxo da Estrutura do Docker e componentes.



Fonte: https://medium.com/cwi-software/primeiros-passos-com-docker-conceitos-b%C3%A1sicos-%C3%A0-cria%C3%A7%C3%A3o-de-sua-primeira-imagem-f6ac2a3c9d25.

1. Instalar o Docker

A primeira coisa que devemos fazer é criar uma conta no Docker Hub (https://hub.docker.com/signup), ela vai ser utilizada tanto para ativação do Docker em nossa máquina quanto para gerenciar nossas imagens.

Depois disso, acesse a página para dar início ao docker e faça o download para a sua máquina, selecionando a versão que atende seu sistema operacional:

- https://www.docker.com/get-started/

Uma vez feito o download, basta conferir a versão com o seguinte comando:

```
$ docker --version
```

2. Fazer o Download da Imagem

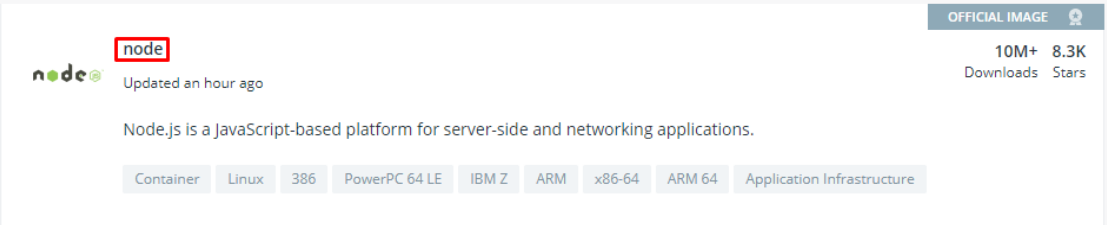
Para fazer o download de uma imagem, usamos o comando:

```
$ docker pull [nome-do-usuario]/[nome-da-imagem]
```

Note que passamos como argumento o nome da imagem que queremos baixar e o nome do usuário. Essas imagens são baixadas de um *registry*, que funciona como plataforma de compartilhamento dessas mesmas imagens. Uma vez cadastrado no registry, o usuário pode fazer download e alterar essas mesmas imagens conforme sua necessidade.

Todavia, imagens oficiais não seguem esse padrão, sendo identificadas apenas pelo seu nome. Se pesquisarmos por “node”, encontraremos mais de 50 mil resultados, sendo o primeiro deles a imagem oficial:

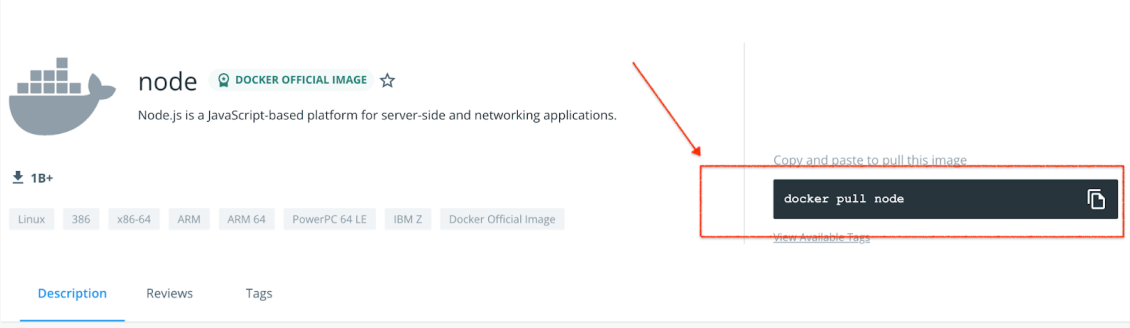
Figura 33 – Figura docker hub: download oficial do Node.



Fonte: https://hub.docker.com/search?q=docker.

Se acessarmos a página da imagem oficial do Node, encontraremos o comando para fazer download da imagem:

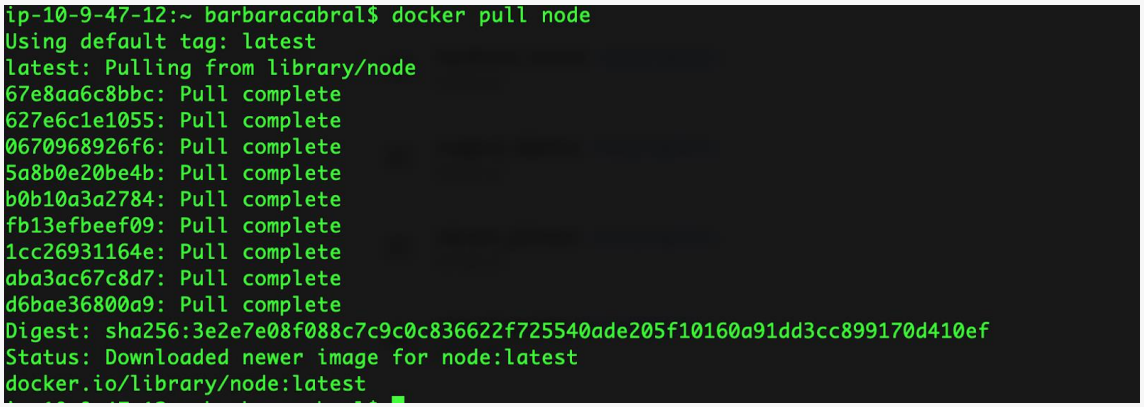
Figura 34 – Figura download do Docker.



Fonte: https://hub.docker.com/_/node.

Ao receber o comando pull sem uma versão específica, o docker client requisita ao docker daemon se já existe uma versão mais recente da imagem em nossa máquina. Se a imagem não existir no repositório local, ele busca em seu registry padrão pelo nome que passamos por parâmetro e realiza o download, como na imagem abaixo:

Figura 35 – Print screen da linha de comando da execução de download do node.



Executando Containers

A partir da imagem podemos iniciar quantos containers quisermos através do comando:

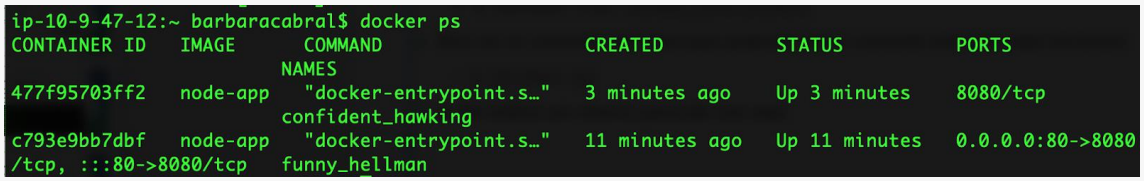
```
$ docker run [nome_container]
```

Para ver os containers em execução podemos usar o comando (em outra aba/terminal):

```
$ docker ps
```

E ele exibirá um retorno parecido com esse:

Figura 36 – Print screen da linha de comando.



Aqui temos informações sobre os containers em execução, como id, imagem base, comando inicial, há quanto tempo foi criado, status, quais portas estão disponíveis e/ou mapeadas para acesso e o nome do mesmo.

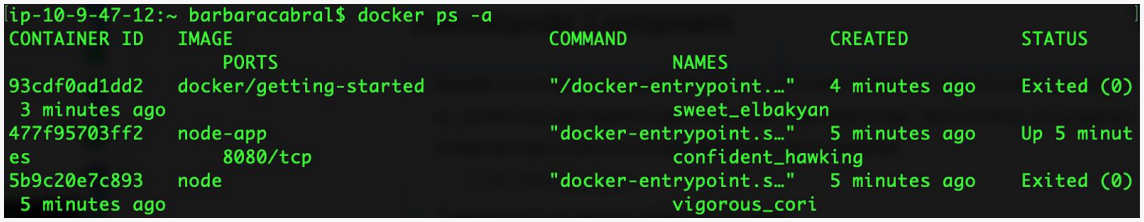
Quando encerramos um container ele não será mais exibido na saída do comando docker ps, porém isso não significa que o container não existe

mais. Para verificar os containers existentes que foram encerrados, podemos usar o comando:

```
$ docker ps -a
```

E ele exibirá um retorno parecido com esse:

Figura 37 – Print screen da linha de comando.



Para remover o container podemos usar o comando:

```
$ docker rm [id_container ou nome_container
```

Caso tenhamos a necessidade de remover todos os container (em execução ou encerrados) podemos usar o comando:

```
$ docker rm $(docker ps -qa)
```

A opção -q do comando docker ps tem como saída somente os ids dos containers, essa lista de ids é passado para o docker rm e com isso será removido todos os containers.

Só será possível remover um container caso o mesmo não esteja em execução, do contrário temos que encerrar o container para removê-lo.

```
$ docker stop [id_container]
```

6.3. Hands On - Criando uma imagem para a minha aplicação

1. Criar uma Imagem

Para criar uma imagem precisamos criar um arquivo, chamado Dockerfile, usando uma outra imagem como base. Também é possível especificar uma série de comandos para criar estruturas específicas para o deploy de nossas aplicações.

Figura 38 – Exemplo de Dockerfile escrito em JavaScript.

Fonte: https://medium.com/cwi-software/primeiros-passos-com-docker-conceitos-b%C3%A1sicos-%C3%A0-cria%C3%A7%C3%A3o-de-sua-primeira-imagem-f6ac2a3c9d25.

Cada um dos comandos acima exerce uma função diferente:

- FROM: especifica a imagem base que será usada.
- ENV: declara uma variável de ambiente.
- RUN: executa um comando.
- WORKDIR: muda o diretório atual (similar ao comando cd em um terminal).

- COPY: copia arquivos e diretórios do host para o build da imagem (sendo o segundo argumento o nome que será usado no build).
- EXPOSE: expõe uma porta qualquer.
- CMD: indica o comando que será rodado para iniciar a aplicação em um container construído usando essa imagem.

Para conhecer os demais comandos para criar imagens no docker, você pode acessar as seguintes referências na documentação do Docker:

- Create a base image.
- Dockerfile Reference.

Criar uma Pequena aplicação usando a Imagem

Usando o arquivo abaixo conseguiremos ver o processo em ação. Basta copiar para um diretório de sua escolha e nomeá-lo para index.js.

```
const http = require('http');

http.createServer((request, response) => {

 response.setHeader('Content-Type', 'text/html');

 response.write('<h1>Muito bem! Aprenda mais sobre Docker nas <a
href="https://docs.docker.com/">docs</a></h1>'

 + '<img src="https://media2.giphy.com/media/mGK1g88HZRa2FlKGbz/giphy.gif" alt="Gif
de duas pessoas aplaudindo">');

 response.end();

}).listen(8080);

console.log('Server running at http://localhost:8080/');
```

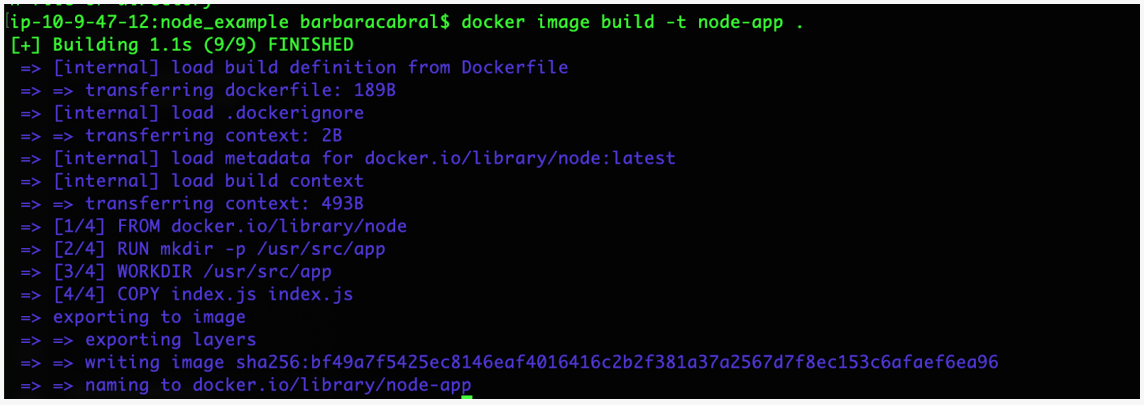
Feito isso, rodamos o seguinte comando:

```
$ docker image build -t node-app .
```

Com o comando acima estamos indicando para o docker construir uma imagem utilizando uma tag para identificá-la como a última versão criada da imagem node-app e dizendo que o Dockerfile necessário para construir a imagem está no diretório atual.

O resultado no terminal deve ser este aqui:

Figura 39 – Exemplo linha de comando para dar um build na imagem.



Após esse comando ser executado, poderemos montar um container usando a imagem gerada. Se eu executar o comando:

```
$ docker ps
```

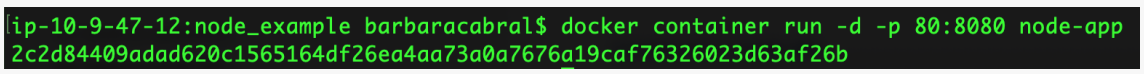
Eu percebo que ainda não vejo nenhum container criado... está vazio! Então agora precisamos usar a imagem para executar o container.

2. Usar a imagem para executar um container

Com o nosso Dockerfile criado e nossa imagem criada, basta executar o comando abaixo para a mágica acontecer:

```
$ docker container run -d -p 80:8080 node-app
```

Figura 40 – Exemplo linha de comando para executar um container.



Com esse comando, estamos basicamente dizendo ao Docker “crie um container(docker container run) que rode no background, sem ocupar meu terminal atual(-d), mapeando uma porta local para uma outra exposta no container(-p 80:8080) e usando a imagem node-app”. Caso a imagem não exista, o docker daemon fará uma busca pelo nome em algum registry. Se eu executar o comando:

```
$ docker ps
```

Posso ver o container rodando na porta 8080:

Figura 41 – Exemplo linha de comando para executar um container.

Agora basta seguir a documentação do Docker para aprender muitas outras possibilidades para montar imagens e configurar containers. Por fim, não se esqueçam de acessar o http://localhost:8080/ e ver o resultado dos quatro passos que seguimos ao longo deste hands-on.

> Capítulo 7

Capítulo 7. Lab pipeline automatizado CI/CD – Parte 1

7.1. Elementos de pipelines e ferramentas

As etapas que compõem um pipeline de CI/CD são subconjuntos distintos de tarefas agrupadas no que chamamos de estágio do pipeline. Os estágios típicos do pipeline são:

- Compilação: estágio em que a aplicação é compilada.
- Teste: estágio em que o código é testado. O uso da automação neste estágio poupa tempo e esforços.
- Lançamento: estágio em que a aplicação é enviada ao repositório.
- Implantação: estágio em que o código é implantado no ambiente de produção.
- Validação e conformidade: etapas para validar uma versão são determinadas pelas necessidades da empresa. É possível usar ferramentas de verificação da segurança de imagens, como o Clair, para ter certeza da qualidade das imagens ao compará-las com vulnerabilidades conhecidas.
- Ferramentas mais utilizadas:

  Jenkins

O Jenkins é uma ferramenta open source escrita em Java, usada para implementar pipelines de integração e entrega contínua (CI/CD).

Você pode configurar o Jenkins para observar qualquer mudança de código em lugares como GitHub, Bitbucket ou GitLab e fazer uma construção automática com ferramentas como Maven e Gradle. Você pode utilizar a

tecnologia de contêiner, como Docker e Kubernetes, iniciar testes e, em seguida, realizar ações como retroceder ou avançar na produção.

Vantagens:

- Builds periódicos, Mailler, Credenciais, Agente SSH, dentre outros.
- Testes automatizados.
- Possibilita análise de código.
- Identificar erros mais cedo.
- Fácil de operar e configurar.
- Builds em diversos ambientes.
- GitHub e Gitlab

Embora ambos tenham semelhanças, até no próprio nome que começa com Git, porque ambos são baseados na famosa ferramenta de controle de versão escrita por Linus Torvalds, mas nem um nem outro são exatamente iguais.

Ambas se baseiam no Sistema de controle de versão Git. Assim, é possível operar sobre o código-fonte dos programas e realizar um desenvolvimento ordenado. Além disso, esta plataforma foi escrita em Ruby on Rails.

O Gitlab também oferece hospedagem para wikis e sistema de rastreamento de bugs. Uma suíte completa para criar e gerenciar projetos de todos os tipos, já que, como no GitHub, atualmente estão hospedados projetos que vão além do código fonte.

Características	GitLab	GitHub
Plano grátis	Repositórios públicos e privados ilimitados	Gratuito apenas para repositórios públicos
Funções de revisão de código	Sim	Sim
Wiki	Sim	Sim
Rastreamento de bugs e problemas	Sim	Sim
Sistema de construção (build)	Sim	Sim (com serviço de terceiros)
Importar projetos	Sim	Não
Projetos de exportação	Sim	Não
Controle de tempo	Sim	Não
Hospedagem na web	Sim	Sim
Auto-hospedagem	Sim	Sim (com plano de negócios)

popularidade	Mais de 546.000 projetos	Mais de 69.000.000 projetos

Outras diferenças:

- Níveis de autenticação: GitLab pode definir e modificar permissões para diferentes colaboradores de acordo com sua função. No caso do GitHub, você pode decidir quem tem direitos de leitura e gravação, mas é mais limitado nesse aspecto.

- Importação e exportação: GitLab contém informações muito detalhadas sobre como importar projetos para movê-los de uma plataforma para outra, como GitHub, Bitbucket, ou trazê-los para o GitLab. Além disso, quando se trata de exportação, o GitLab oferece um trabalho muito sólido. No caso do GitHub, a documentação detalhada não é fornecida, embora o GitHub Importer possa ser usado como uma ferramenta, embora possa ser um pouco mais restritivo quando se trata de exportação.

- Cicle CI

Lançado em 2011, o CircleCI monitora repositórios do GitHub, GitHub Enterprise e Atlassian Bitbucket, e lança compilações para cada nova alteração de código.

O software suporta Go, Java, Ruby, Python, Scala, Node.js, PHP, Haskell e qualquer outra linguagem que rode em Linux ou macOS.

Um painel e uma API permitem rastrear o status de compilações e métricas relacionadas a compilações – enquanto a integração com o Slack notifica os times DevOps quando surgem falhas e outros problemas.

Pipelines na AWS

- AWS CodeBuild – Um serviço de integração contínua totalmente gerenciado que compila o código-fonte, executa testes e produz pacotes de software que estão prontos para implantação.
- AWS CodeCommit – Um serviço totalmente gerenciado de controle de código-fonte que hospeda repositórios seguros baseados em Git.
- AWS CodeDeploy – É um serviço totalmente gerenciado de implantação que automatiza implantações de software em diversos serviços de computação como Amazon EC2, AWS Fargate, AWS Lambda e servidores locais.
- AWS CodePipeline – É um serviço gerenciado de entrega contínua que ajuda a automatizar pipelines de liberação para oferecer atualizações rápidas e confiáveis de aplicativos e infraestrutura.
- AWS Lambda – É um serviço de computação sem servidor que permite executar código sem provisionar ou gerenciar servidores, pagando apenas pelo tempo de computação que consome.
- Amazon SNS – Simple Notification Service – É um serviço de mensagens totalmente gerenciado para a comunicação de aplicação para aplicação (A2A) e de aplicação para pessoa (A2P).
- Amazon S3 – Simple Storage Service – É um serviço de armazenamento de objetos que oferece escalabilidade, disponibilidade de dados, segurança e performance líderes do setor.
- AWS Systems Manager Parameter Store – Oferece um armazenamento centralizado para gerenciar os dados de configuração em texto simples, como strings de bancos de dados, ou secretos, como senhas.

- Ferramentas de Testes Contínuo

  Integração com ferramentas externas ou serviços de terceiros:

- SonarQube (SAST) – Captura bugs e vulnerabilidades em seu aplicativo, com milhares de regras automatizadas de análise de código estático.

- OWASP Zap (DAST) – Ajuda a encontrar automaticamente vulnerabilidades de segurança em seus aplicativos da web enquanto você desenvolve e testa seus aplicativos.

- Lighthouse - Integra em qualquer servidor de integração contínua.

- Serviços de Log e Monitoramento Contínuo

- AWS CloudWatch Logs – Permite monitorar, armazenar e acessar seus arquivos de log de instâncias EC2, AWS CloudTrail, Amazon Route 53 e outros recursos.

- AWS CloudWatch Events – Oferece um fluxo quase em tempo real de eventos do sistema que descreve as mudanças nos recursos da AWS.

- Serviço de Auditoria e Segurança

- AWS CloudTrail – Permite governança, conformidade, auditoria operacional e auditoria de risco de sua conta AWS.

- AWS IAM – Identity and Access Management – Permite que você gerencie o acesso aos serviços e recursos da AWS com segurança. Com o IAM, você pode criar e gerenciar usuários e grupos da AWS e usar permissões para permitir e negar seu acesso aos recursos da AWS.

- AWS Config – Permite acessar, auditar e avaliar as configurações de seus recursos AWS.

- Serviços de Operações

- AWS Security Hub – Oferece uma visão abrangente de seus alertas de segurança e postura de segurança em suas contas AWS. Esta postagem usa o Security Hub para agregar todas as descobertas de vulnerabilidade como um único painel centralizado.

- AWS CloudFormation – Oferece uma maneira fácil de modelar uma coleção de recursos relacionados da AWS e de terceiros, provisioná-los de forma rápida e consistente e gerenciá-los ao longo de seus ciclos de vida, tratando a infraestrutura como código.

- AWS Systems Manager Parameter Store – Fornece armazenamento hierárquico seguro para gerenciamento de dados de configuração e gerenciamento de segredos. Você pode armazenar dados como senhas, strings de banco de dados, IDs de Amazon Machine Image (AMI) e códigos de licença como valores de parâmetro.

- AWS Elastic Beanstalk – Um serviço fácil de usar para implantar e escalonar aplicativos e serviços da web desenvolvidos com Java, .NET, PHP, Node.js, Python, Ruby, Go e Docker em servidores familiares, como Apache, Nginx, Passenger e IIS. Esta postagem usa Elastic Beanstalk para implantar a pilha LAMP com WordPress e Amazon Aurora MySQL. Embora usemos o Elastic Beanstalk para esta postagem, você pode configurar o pipeline para implantar em vários outros ambientes na AWS ou em outro lugar, conforme necessário.

Pipelines na Google Cloud

Nuvem Aberta e Várias Nuvens

A nuvem aberta depende do código aberto. A Google tem um longo histórico de liderança em código aberto por projetos como o Kubernetes, o TensorFlow e outros.

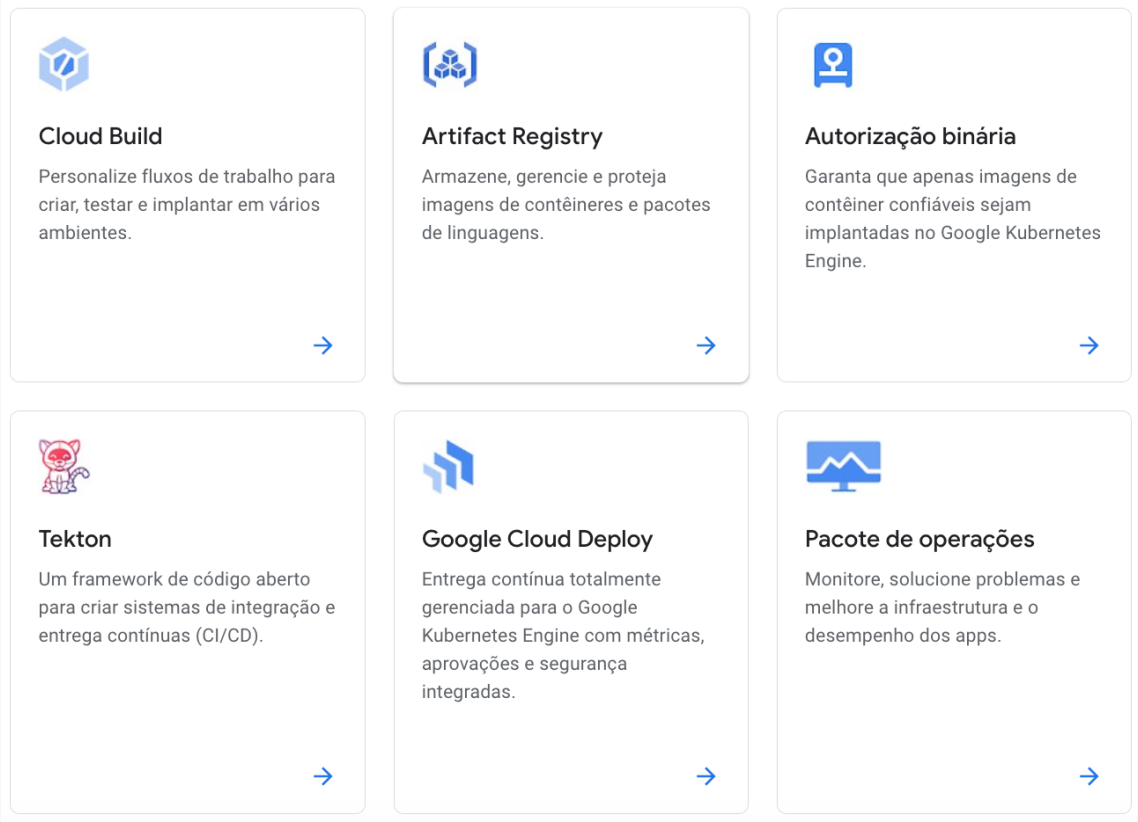
O código aberto proporciona flexibilidade de implantação e, se necessário, de migração para cargas de trabalho essenciais dentro ou fora de plataformas de nuvem pública.

Tem a flexibilidade de migrar, criar e otimizar aplicativos em ambientes híbridos e com várias nuvens, minimizando a dependência do fornecedor, aproveitando as melhores soluções e atendendo aos requisitos regulatórios.

Possibilidades:

- Promover a transformação com as soluções de várias nuvens do Google.
- Gerenciar apps e dados em qualquer lugar.
- Detalhar os silos e descobrir novos insights.
- Acelerar a entrega de aplicativos.
- Escalonar com tecnologia aberta e flexível.
- Obter vantagens da Infraestrutura Global.

Figura 42 – Google Cloud DevOps Tools.



Fonte: https://www.devops-research.com/research.html#capabilities.

Google Cloud Build

Criar, testar e implantar na plataforma Google de CI/CD sem servidor.

Vantagens:

- Criar software rapidamente em todas as linguagens de programação, incluindo Java, Go, Node.js e mais.
- Escolher entre 15 tipos de máquinas.
- Executar centenas de builds simultâneas por pool.
- Implantar em vários ambientes, como VMs, sem servidor, Kubernetes ou Firebase.

- Acessar fluxos de trabalho CI/CD totalmente gerenciados, hospedados na nuvem e na sua rede particular.
- Manter seus dados em repouso dentro de uma região geográfica ou um local específico com a residência de dados.
- Google Artifact Registry

Nova geração do Container Registry para Armazenar, gerenciar e proteger os artefatos do seu build.

É uma central unificada para sua organização gerenciar imagens de contêiner e pacotes de instaladores de linguagem (como Maven e NPM). Totalmente integrado às ferramentas e aos ambientes de execução do Google Cloud e é compatível com protocolos de artefato nativo.

Dessa forma, é fácil integrar o Artifact Registry às suas ferramentas de CI/CD para configurar pipelines automatizados.

Google Kubernetes Engine

O GKE oferece um ambiente gerenciado para implantação, gerenciamento e escalonamento de aplicativos em contêineres usando a infraestrutura do Google.

O ambiente do GKE consiste em várias máquinas (especificamente, instâncias do Compute Engine) agrupadas para formar um cluster.

O GKE funciona com aplicativos em contêiner. Ou seja, aplicativos agrupados em instâncias de espaço do usuário isoladas e independentes da plataforma em um Docker, por exemplo.

Para aplicativos ou trabalhos em lote, esses contêineres são chamados coletivamente de cargas de trabalho no GKE e no Kubernetes.

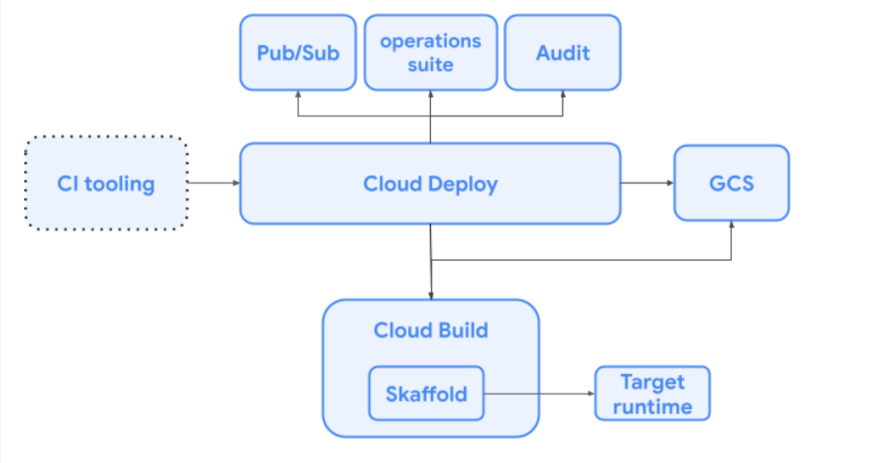
Antes de implantar uma carga de trabalho em um Cluster do GKE, é necessário empacotá-la em um contêiner.

Quando você executa um cluster do GKE, também recebe os benefícios dos recursos avançados de gerenciamento de cluster que o Google Cloud oferece, como:

- Balanceamento de carga do Google Cloud para instâncias do Compute Engine.
- Pools de nós para designar subconjuntos de nós em um cluster, o que proporciona mais flexibilidade.
- Escalonamento automático da contagem de instâncias de nós do cluster.
- Atualizações automáticas do software de nós do cluster.
- Reparação automática de nós para manter a disponibilidade e a integridade do nó.
- Geração de registros e monitoramento com o pacote de operações do Google Cloud para visibilidade no cluster.
- Google Cloud Deploy

O Google Cloud Deploy é um serviço gerenciado que automatiza a entrega de aplicativos para uma série de ambientes de destino em uma sequência de promoções definidas. Quando você quiser implantar seu aplicativo atualizado, será necessário criar uma versão cujo ciclo de vida é gerenciado por um pipeline de entrega.

Figura 43 – Visão Geral do Cloud Build.



Fonte: https://cloud.google.com/build/docs/overview?hl=pt-br.

7.2. Github Actions

GitHub Actions é uma plataforma de integração contínua e entrega contínua (CI/CD) que permite automatizar seu build, testar e executar seu pipeline de implantação. Você pode criar fluxos de trabalho que constroem e testam cada pull request no seu repositório ou fazem o deploy de pull requests mergeados para produção.

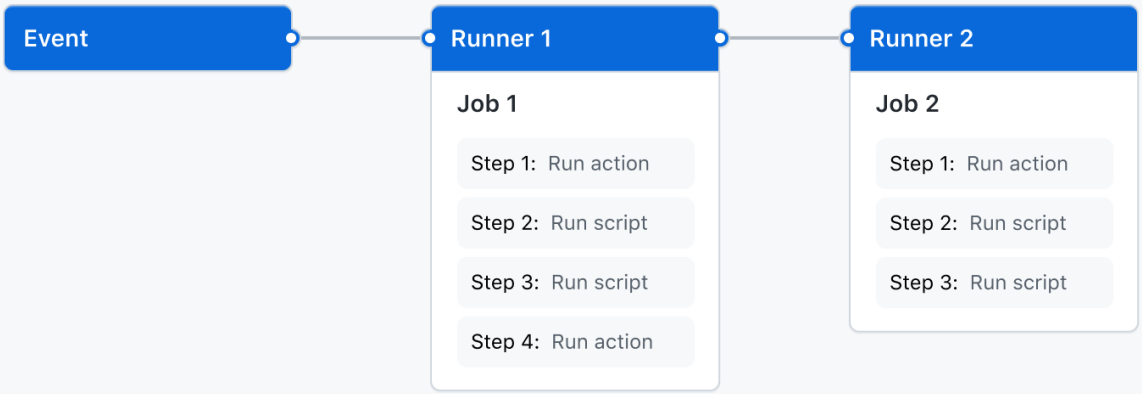
Ele permite também executar fluxos de trabalho quando outros eventos acontecem em seu repositório. Por exemplo, você pode executar um fluxo de trabalho para adicionar automaticamente as "*tags*" apropriadas sempre que alguém criar uma nova *issue* em seu repositório.

O GitHub fornece máquinas virtuais Linux, Windows e macOS para executar seus fluxos de trabalho *(workflows)*, ou você pode hospedar seus próprios executores auto-hospedados em seu próprio data center ou infraestrutura de nuvem.

Você pode configurar um fluxo de trabalho do GitHub Actions para ser acionado quando ocorrer um evento em seu repositório, como uma solicitação de pull request sendo aberta ou um problema sendo criado. Seu

fluxo de trabalho contém um ou mais *jobs* que podem ser executados em ordem sequencial ou em paralelo:

Figura 44 – Visão Geral do Cloud Build.



Fonte: https://docs.github.com/pt/actions/learn-github-actions/understanding-github-actions.

Fluxos de Trabalho / Workflows

Um fluxo de trabalho é um processo automatizado configurável que executa um ou mais jobs.

- Os fluxos de trabalho são definidos no diretório *.github/workflows* por um arquivo YAML em um repositório;
- Um repositório pode ter um ou vários fluxos de trabalho, cada um deles pode executar um conjunto diferente de tarefas.
- Podem ser acionados por um evento no repositório, acionado manualmente ou de acordo com um cronograma definido.
- P. ex., você pode ter um fluxo de trabalho para criar e testar pull requests, outro fluxo de trabalho para implantar seu aplicativo toda vez que uma versão for criada, e outro fluxo de trabalho que adiciona uma tag toda vez que alguém abre uma nossa issue.

Para saber mais, acesse: "Usando fluxos de trabalho".

Eventos / Events

Um evento é uma atividade específica em um repositório que aciona a execução de um fluxo de trabalho.

- Por exemplo, a atividade pode originar-se de GitHub quando alguém cria uma solicitação de pull request, abre um problema ou faz envio por push de um commit para um repositório.
- Você também pode acionar a execução de um fluxo de trabalho em um cronograma, postando em uma API REST, ou manualmente.

Para obter uma lista completa de eventos que podem ser usados para acionar fluxos de trabalho, consulte Eventos que acionam fluxos de trabalho.

Trabalhos / Jobs

Um trabalho é um conjunto de etapas em um fluxo de trabalho que é executado no mesmo executor.

- Cada etapa é um script do shell que será executado, ou uma ação que será executada.
- As etapas são executadas em ordem e dependem uma da outra. Uma vez que cada etapa é executada no mesmo executor, você pode compartilhar dados de um passo para outro.
- Por exemplo, você pode ter uma etapa que compila a sua aplicação seguida de uma etapa que testa ao aplicativo criado.

Você pode configurar as dependências de um trabalho com outros trabalhos; por padrão, os trabalhos não têm dependências e são executadas em paralelo um com o outro.

- Quando um trabalho leva uma dependência de outro trabalho, ele irá aguardar que o trabalho dependente seja concluído antes que possa executar.

- Por exemplo, você pode ter vários trabalhos de criação para diferentes arquiteturas que não têm dependências e um trabalho de pacotes que depende desses trabalhos.

- Os trabalhos de criação serão executados em paralelo e, quando todos forem concluídos com sucesso, o trabalho de empacotamento será executado.

Para obter mais informações sobre os trabalhos, consulte "Usando trabalhos".

Ações / Actions

Uma ação é uma aplicativo personalizado para a plataforma de GitHub Actions que executa uma tarefa complexa, mas frequentemente repetida. Use uma ação para ajudar a reduzir a quantidade de código repetitivo que você grava nos seus arquivos de fluxo de trabalho. Uma ação pode extrair o seu repositório git de GitHub, configurar a cadeia de ferramentas correta para seu ambiente de criação ou configurar a autenticação para seu provedor de nuvem.

Você pode gravar suas próprias ações, ou você pode encontrar ações para usar nos seus fluxos de trabalho em GitHub Marketplace.

Para obter mais informações, consulte "Criar ações".

Executores / Runners

Um executor é um servidor que executa seus fluxos de trabalho quando são acionados. Cada executor pode executar uma tarefa por vez.

GitHub fornece executores para Ubuntu Linux, Microsoft Windows e macOS para executar seus fluxos de trabalho.

Cada fluxo de trabalho é executado em uma nova máquina virtual provisionada. GitHub também oferece larger runners, que estão disponíveis em configurações mais amplas. For more information, see “Using larger runners”.

Se você precisar de um sistema operacional diferente ou precisar de uma configuração de hardware específica, você poderá hospedar seus próprios executores.

Para mais informações sobre executores auto-hospedados, consulte “Hospedando os seus próprios executores”.

Um exemplo de workflow

No seu repositório, crie o diretório `.github/workflows`, crie um novo arquivo denominado learn `-github-actions.yml` e adicione o código a seguir:

Figura 45 – Exemplo de workflow.

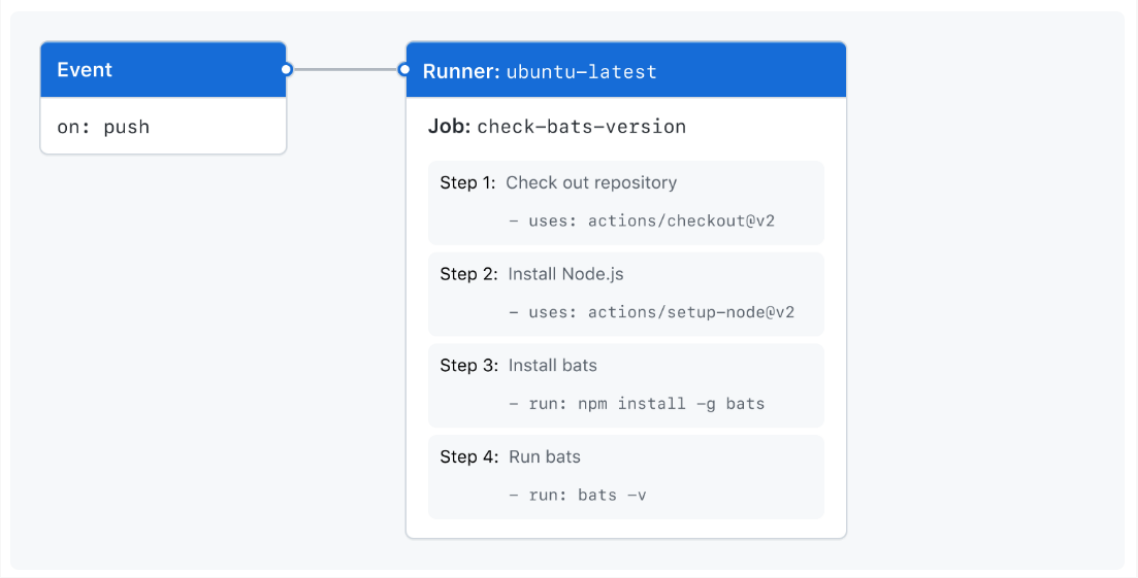
```yaml
name: learn-github-actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

Fonte: https://docs.github.com/pt/actions/learn-github-actions/understanding-github-actions.

Visualizando o arquivo

Neste diagrama, você pode ver o arquivo de fluxo de trabalho que acabou de criar e como os componentes de GitHub Actions estão organizados em uma hierarquia:

Figura 46 – Hierarquia de Passos.

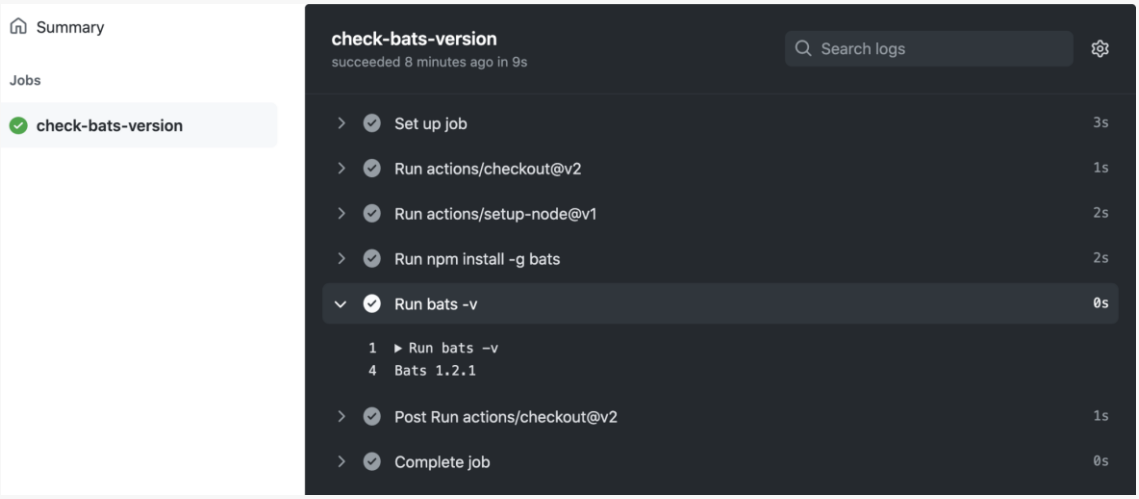


Fonte: https://docs.github.com/pt/actions/learn-github-actions/understanding-github-actions.

Visualizando o resultado de cada etapa

Quando o workflow é lançado, uma execução é criada. Depois disso, você pode visualizar um gráfico com o progresso de cada atividade no GitHub.

Figura 47 – Gráfico com progresso de cada etapa



Fonte: https://docs.github.com/pt/actions/learn-github-actions/understanding-github-actions.

> Capítulo 8

Capítulo 8. Lab pipeline automatizado CI/CD – Parte 2

8.1. Outras ferramentas de auditoria de código

Integração com ferramentas externas ou serviços de terceiros:

- SonarQube (SAST) – Captura bugs e vulnerabilidades em seu aplicativo, com milhares de regras automatizadas de análise de código estático.
- OWASP Zap (DAST) – Ajuda a encontrar automaticamente vulnerabilidades de segurança em seus aplicativos da web enquanto você desenvolve e testa seus aplicativos.
- Lighthouse – Íntegra em qualquer servidor de integração contínua.

Referências

{JSON} Placeholder. JSONPlanceholder, out. 2021. Disponível em: <https://jsonplaceholder.typicode.com/>. Acesso em: 12 abr. 2022.

ALDER, Alice. 10 Ferramentas de Teste de APIs para você conhecer. InfoQ, 21 ago. 2018. Disponível em: <https://www.infoq.com/br/articles/10-ferramentas-teste-api/>. Acesso em: 12 abr. 2022.

CÓDIGOS de status de respostas HTTP. MDN web docs, 16 ago. 2021. Disponível em: <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Status>. Acesso em: 12 abr. 2022.

CONTINUOUS Delivery Pipeline. SAFe: Provided by Scaled Agile, 27 set. 2021. Disponível em: <https://www.scaledagileframework.com/continuous-delivery-pipeline/>. Acesso em: 12 abr. 2022.

CONTINUOUS Deployment. SAFe: Provided by Scaled Agile, 27 set. 2021. Disponível em: <https://www.scaledagileframework.com/continuous-deployment/>. Acesso em: 12 abr. 2022.

DOWNLOAD Insomnia for Windows. Insomnia by Kong, c2022. Disponível em: <https://insomnia.rest/download>. Acesso em: 12 abr. 2022.

EXPLICAÇÃO da integração contínua. AWS: Amazon, c2022. Disponível em: <https://aws.amazon.com/devops/continuous-integration/>. Acesso em: 12 abr. 2022.

GRESPI, Thiago. Testes de API (Parte 1) - Entendendo e botando a mão na massa com Postman. Medium, 26 fev. 2019. Disponível em: <https://medium.com/@thiagogrespi/testes-de-api-parte-1-entendendo-e-botando-a-m%C3%A3o-na-massa-com-postman-b365923b83e1>. Acesso em: 12 abr. 2022.

MEDEIROS, Rafael. Controle de Versão e Gerenciamento de Código. Enacom: da ciência ao produto, c2017. Disponível em: <https://blog.enacom.com.br/2019/01/07/controle-de-versao-e-gerenciamento-de-codigo/>. Acesso em: 12 abr. 2022.

MUNIZ, Antônio; CABRAL, Bárbara; BOAS, Carol Vilas; COLARES, Rodolfo. Jornada Ágil de Qualidade: Aplique técnicas de qualidade no início do ciclo para implantação contínua de software (Jornada Colaborativa). Editora Brasport. 2020

MUNIZ, Antônio; SANTOS, Rodrigo; IRIGOYEN, Analia; MOUTINHO, Rodrigo; Jornada DevOps: unindo cultura ágil, Lean e tecnologia para entrega de software de qualidade (Jornada Colaborativa). Editora Brasport. Maio, 2019

TAKE the DORA DevOps Quick Check. Google Cloud. Disponível em: <https://www.devops-research.com/quickcheck.html>. Acesso em: 12 abr. 2022.