Tópicos Especiais em Desenvolvimento Back End

Bootcamp Desenvolvedor(a) Node.js

Lucas Goulart Silva

2021

Tópicos Especiais em Desenvolvimento Back End

Bootcamp Desenvolvedor(a) Node.js

Lucas Goulart Silva

Sumário

Capítulo 1. Testes Unitários

Testes de software são uma importante etapa integrante do processo de desenvolvimento de software. Por meio dos testes, é possível identificar cenários que necessitem de ajustes ou correções antes que esses erros aconteçam de fato. Neste capítulo, serão abordados os seguintes conceitos:

- Testes de Software.
  - Pirâmide de Testes.
- Metodologia XP.
  - TDD.
- Testes Unitários.
- A biblioteca Jest.

1.1. Testes de Software

O surgimento do termo *"bug"* – palavra da língua inglesa que significa inseto – nos remete à ideia de um defeito ou falha que pode ser encontrada no código fonte de um programa e que, portanto, é responsável pelo mau funcionamento dele. O termo, porém, é bem mais antigo. Thomas Edison, importante empresário que financiou o desenvolvimento de diversos equipamentos eletroeletrônicos industriais, já usava o termo para se referir a problemas causados pela presença de insetos em suas ferramentas. O termo foi, posteriormente, utilizado para relatar a presença de uma traça no computador Mark II, em 1947. Desde então, o termo tem se tornado cada vez mais comum em meio aos profissionais de TI e, mais recentemente, tem sido utilizado até mesmo por pessoas cujas atividades profissionais não tenham qualquer relação com o desenvolvimento e com a engenharia de software.

Tão antigo quanto o termo é a necessidade que os envolvidos na engenharia de um produto têm em encontrá-los e corrigi-los com a maior eficiência possível. Muito

além da capacidade de antever problemas por meio de um desenho – seja com cálculo, projetos, simulação ou observação –, bugs acontecem naturalmente e, portanto, é extremamente necessário que sejam realizados testes. A submissão de produtos ao processo de testes tem por **objetivo demonstrar a presença de erros** e, assim, permitir que sejam corrigidos.

Os testes de software podem ser classificados de diferentes maneiras:

- Quando classificados com relação à característica do produto que está sendo avaliada, subdividimos os testes em testes de segurança, testes de performance – realizando-se testes com grandes volumes de dados, por exemplo; ou testes funcionais – que visam verificar se o comportamento obtido é, de fato, o adequado.

- Quando relacionados ao ponto de vista daqueles que executam os testes, estes podem ser classificados como: testes de caixa branca, quando aquele que o realiza tem acesso ao código fonte do objeto testado; ou como testes de caixa preta, quando não há acesso ao código fonte. Para este último caso, o foco dos testes é exclusivo no comportamento e nos resultados esperados.

Como toda interação humana é sujeita a falhas, as boas práticas recomendam que os testes devam ocorrer de forma automatizada. Testes devem, portanto, ser codificados por meio de uma linguagem de programação, em algoritmos que descrevam o comportamento e os resultados esperados. Desta forma, é possível garantir seu maior sucesso.

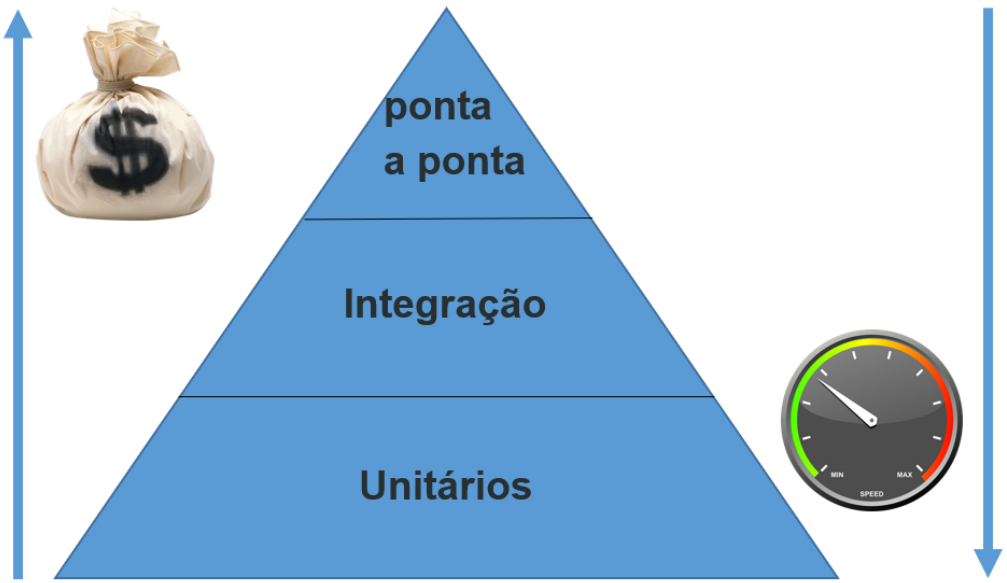
Nos modelos mais tradicionais, existe a ideia de que os testes sejam realizados durante uma fase específica do ciclo de vida de desenvolvimento. Contudo, é um consenso cada vez mais comum que é de fundamental importância, para a qualidade e garantia de sucesso de um projeto, que ocorram etapas de verificação durante todo o ciclo de desenvolvimento. Dessa maneira, é possível encontrar falhas antecipadamente e corrigi-las o quanto antes.

Com relação à etapa do desenvolvimento em que ocorrem, é possível classificar os testes em testes de unidade, testes de integração e testes de ponta a ponta (ou aceitação).

A visualização dessa classificação, por meio da pirâmide de testes, permite discorrer sobre as principais características dos itens presentes em cada uma destas etapas:

- É na **base** que estão presentes os **testes unitários**, pois são os primeiros a serem construídos. A maior largura indica que o volume de testes unitários será maior do que os demais testes;
- É no **meio** que são apresentados os testes de integração, cuja principal finalidade é integrar pequenas unidades e verificar o seu comportamento quando atuando em conjunto;
- No **topo**, os testes de ponta a ponta representam aqueles com marcos de maior importância no projeto e que, normalmente, estão associados a entregas e finais de ciclo.

Figura 1 - Pirâmide de Testes.



Os testes tendem a se tornar mais rápidos na medida em que se caminha do topo para a base da pirâmide. Por outro lado, os testes tendem a se tornar mais caros e valiosos na medida em que se caminha para o topo da pirâmide.

Um resumo sobre as principais características dos testes, de acordo com a etapa em que ocorrem, pode ser visualizado na **Tabela 1 - Características dos Testes por Etapa**.

Tabela 1 - Características dos Testes por Etapa.

	Unitários	Integração	Ponta a ponta
Objetivo	Testar uma pequena unidade – pode ser uma função ou classe	Testar a integração entre algumas unidades do sistema	Testar o sistema como um todo
Periodicidade	Constantemente durante o desenvolvimento	Diariamente durante o desenvolvimento	Ao final de uma etapa
Quantidade	Grande	Intermediária	Pequena
Velocidade	Rápido	Razoável	Lento
Valor agregado	Pouco	Mediano	Muito
Custo	Baixo	Intermediário	Alto

1.2. Metodologia XP

A programação extrema (e***X***treme ***P***rogramming – XP) é uma metodologia de desenvolvimento de software focada na geração de entregas de qualidade. É considerada uma metodologia de desenvolvimento de software ágil e, portanto, prevê

a capacidade de que as entregas ocorram em pequenos ciclos, em contrapartida ao tradicional modelo de cascata.

Em meio às revoluções tecnológicas acarretadas pela rápida expansão da internet em meados dos anos 90, um desenvolvedor de software chamado Kent Beck percebeu que a realização de entregas de qualidade não era suportada pelos processos da época. Ele, então, decidiu agrupar uma séria de boas práticas, algumas das quais já existentes à época, mas ainda utilizadas de forma isolada, criando, assim, uma metodologia inovadora.

Uma das principais ideias da metodologia XP é que boas práticas devem ser levadas ao extremo para garantir a boa qualidade das entregas. São algumas das boas práticas preconizadas:

- *Pair-programming*: Também chamada de programação em pares, é uma técnica que prega que dois programadores trabalhem juntos em um único ambiente. Enquanto o primeiro escreve o código, o outro observa o resultado que está sendo produzido. Ao final do processo, os envolvidos devem negociar a melhor solução para o problema que estão resolvendo;

- *Stand-up meeting*: Realização de reuniões diárias nas quais os participantes devem permanecer de pé, evitando, assim, que durem mais tempo do que o considerado estritamente necessário;

- **Testes unitários:** Testes unitários devem ser desenvolvidos constantemente e têm por objetivo encontrar falhas em cada um dos componentes do sistema produzido. A forma pela qual os testes são produzidos é representada pelos ciclos do TDD.

O TDD é uma técnica **de desenvolvimento de software** onde os testes devem guiar o desenvolvedor na criação das tarefas. Os testes devem ser, portanto, criados antes do código de produção. Inicialmente, a ideia surgiu como uma necessidade para projetos ágeis, especialmente o XP. Mas, com o passar do tempo, essa técnica foi ganhando cada vez mais espaço, sendo amplamente adotada na

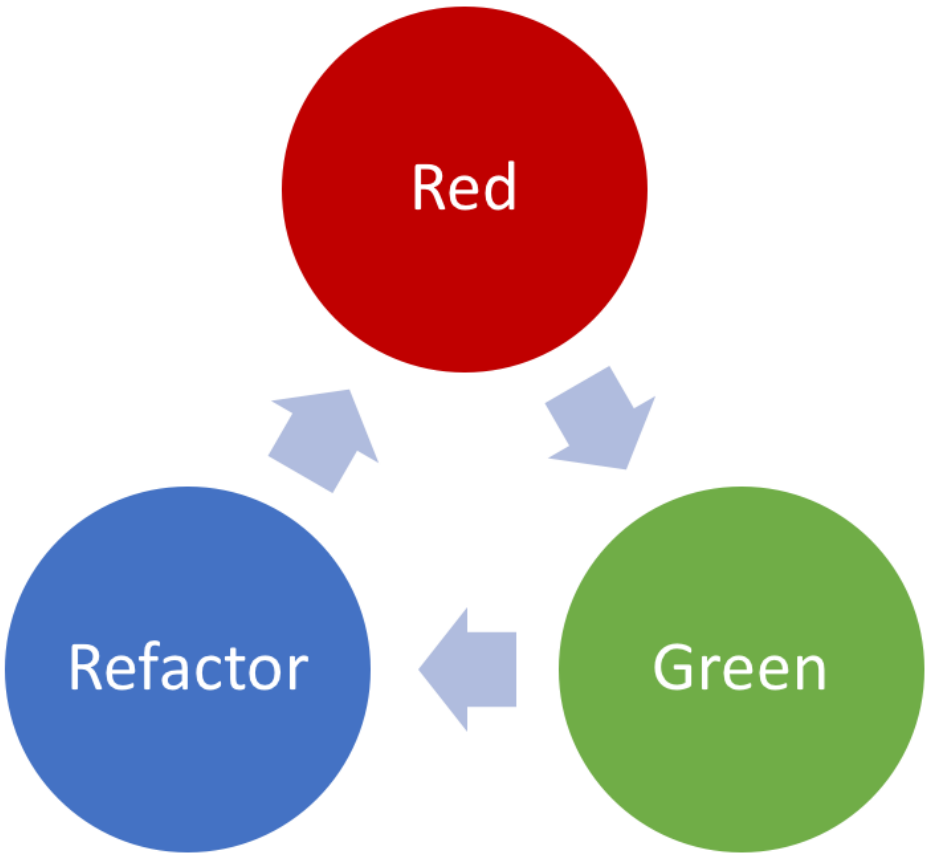
indústria do desenvolvimento de software. Hoje em dia, o TDD é amplamente utilizado mesmo para casos em que se esteja trabalhando com metodologias mais tradicionais.

A técnica prevê a existência de 3 fases e, para cada uma das fases, o desenvolvedor deve desempenhar um papel distinto. Ao final da terceira e última fase, o processo se inicia novamente e novos recursos serão criados.

FASE 01 – TESTE

O desenvolvedor deve codificar um novo teste. O teste deve, então, ser executado. Como a função que atende a esse código ainda não existe, o teste deve falhar. Caso não ocorram falhas, o ciclo se encerra neste ponto e nenhuma alteração será realizada no código. Desta forma, fica claro que a motivação para a codificação é a existência de um teste que esteja falhando.

Figura 2 - Ciclos do TDD.



FASE 02 – CODIFICAÇÃO

Após visualizar um teste falhando, o foco do desenvolvedor passa, então, a ser a escrita de uma solução que possa atender ao novo teste que está falhando. Ao mesmo tempo, o desenvolvedor deve se manter atento para que suas alterações não quebrem os outros testes. Nesta fase não é esperado que o código desenvolvido apresente a solução ótima. Pequenas imperfeições são aceitáveis a fim de garantir a entrega do teste passado.

FASE 03 – REFATORAÇÃO

O ciclo de refatoração é aquele no qual o desenvolvedor poderá alterar a forma como a solução está implementada sem, no entanto, alterar o que está sendo feito. Como resultado deste ciclo, é esperado um código mais limpo, uma solução menos acoplada, sem, no entanto, alterar o que a solução de fato faz.

As consecutivas execuções dos ciclos são chamadas de *baby steps*, porque remetem ao processo cuidadoso de evoluir a aplicação em pequenos passos, semelhante a passos de bebê.

1.3. Testes Unitários

Testes unitários são aqueles criados para verificar o correto funcionamento de uma unidade de código. Normalmente, essas unidades são representadas por uma classe, uma função, ou um pequeno conjunto de funções que possam ser semanticamente agrupadas. São testes de caixa branca, automatizados e normalmente tem caráter de atender a uma especificação funcional.

A fim de isolar cada uma das unidades testadas durante o desenvolvimento de testes unitários, é bastante comum que sejam utilizados simuladores para outras unidades senão a que está sendo submetida aos testes. Esses simuladores são chamados de *Mock* e têm a função de simular de maneira simples, por vezes até mesmo rudimentar, o comportamento de outros módulos do sistema. *Mocks* são de

fundamental importância para que se possa realizar testes unitários em unidades que possuem um conjunto grande de interfaces com outras unidades.

Os testes unitários são localizados na base da pirâmide e são, portanto, os primeiros a serem desenvolvidos e, normalmente, os mais volumosos do conjunto de testes automatizados de uma aplicação. Por realizarem testes em unidades isoladas do código, os testes unitários não são capazes de agregar grande valor à entrega, uma vez que não asseguram o correto funcionamento das unidades em conjunto. Porém, devido a sua velocidade de realização e baixo custo, são de grande importância.

1.4. Jest

Jest é um framework para testes em Javascript que suporta diversas das bibliotecas mais utilizadas do mercado, como React, Angular e **Node.js**. A arquitetura do framework é dividida em dois pontos principais: um executor de testes (*test-runner*) responsável por localizar os arquivos de teste do projeto e verificar sua execução; uma biblioteca de comandos que permite que os desejos sejam expressados na forma de assertivas, que o framework dá o nome de **matchers.**

O Jest possui matchers para comparação de números, textos, booleanos, arrays, entre outros.

Figura 3 - Exemplo de teste unitário escrito com Jest.

```
test('Valor da soma das prestações deve ser igual ao montante com duas casas decimais', () => {
  // Dado (given)
  const numeroPrestacoes = 3
  const montante = 100

  // Quando (when)
  const prestacoes = calculaValor.calcularPrestacoes(montante, numeroPrestacoes)

  // Então (then)
  expect(prestacoes.length).toBe(numeroPrestacoes)
  expect(prestacoes).tenhaSomaDeValoresIgual(montante)
  expect(prestacoes).sejaDecrescente()
})
```

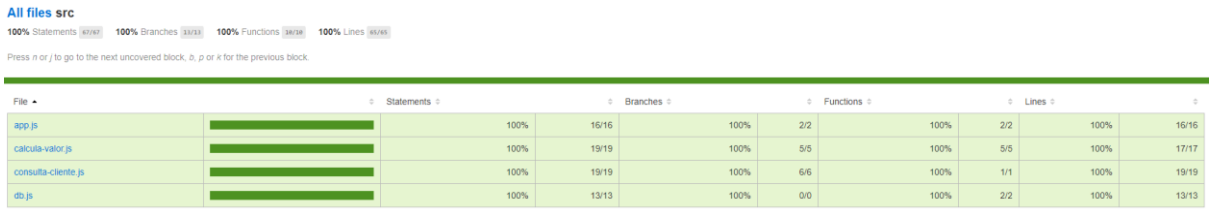
No exemplo apresentado na **Figura 3 - Exemplo de teste unitário escrito com Jest**, é possível visualizar algumas das funções mais básicas do Jest:

- test: Função que cria um teste. Deve receber, como parâmetros, uma *string* com o nome do teste e uma função que executa o teste;
- expect: Função utilizada após a realização de um teste contendo uma expectativa de resultado que deve ser atendida pelo teste. Recebe, como parâmetro, o valor real;
- toBe: Uma das principais *assertions* do jest é utilizada encadeada com a função 'expect'. Caso os valores esperado e real sejam iguais, o teste é considerado sucesso. Caso contrário, o teste falha.

Dada uma aplicação em Javascript, a sua instalação pode ser feita por meio do comando **'npm install –save-dev jest'**. A configuração inicial poderá ser feita por meio do comando **'jest –init'**. Após realizada a configuração, a realização dos testes deverá ocorrer por meio do comando **'npm test'**.

O módulo de execução de testes do Jest é capaz de coletar estatísticas dos testes realizados e gerar um relatório contendo as principais métricas.

Figura 4 - Relatório de Cobertura do Jest.



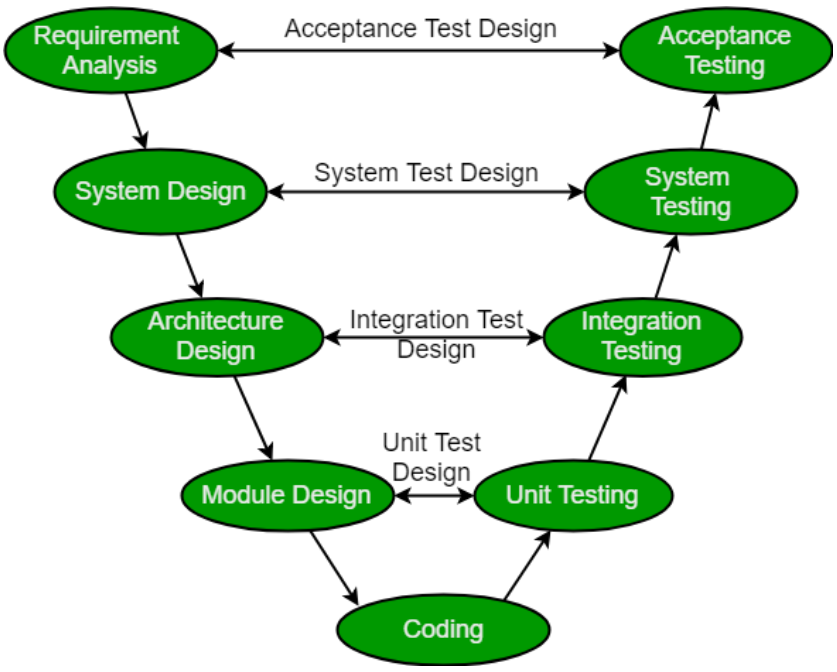
Capítulo 2. Testes de Integração

Testes de integração são previstos pela metodologia XP como de fundamental importância para garantir a qualidade do software. Estes devem ser executados diariamente, usando, como base, o código já integrado ao Sistema de Controle de Versão. São criados para verificar o correto funcionamento das unidades quando executadas de forma integrada. Normalmente, são capazes de representar uma visão bem próxima das regras de negócio solicitadas pelo cliente.

Normalmente, durante a realização dos testes de integração é o momento em que ocorre a primeira interação entre componentes e, portanto, importantes módulos da aplicação que, até então, estavam isolados, passam a ser requisitados. Por exemplo: Banco de Dados, Cache, entre outros. Por isso, podem fornecer métricas a respeito de itens não funcionais, como o desempenho da aplicação em determinados cenários.

De acordo com o modelo em V do Ciclo de Vida de Desenvolvimento de Software, os Testes de Integração estão para o desenho arquitetural da aplicação, assim como os Testes Unitários estão para o desenho de um módulo.

Figura 5 - Modelo em V do Ciclo de Vida de Desenvolvimento de Software.



2.1. supertest

O **supertest** é uma biblioteca do Node.js para a realização de testes de WEB APIs utilizando o Jest. São, portanto, muito utilizados para o desenvolvimento de testes de integração.

Dada uma aplicação em Javascript que esteja expondo uma interface REST e que já tenha o **express** instalado, a instalação do supertest pode ser feita por meio do comando **'npm install –save-dev supertest'**. Para o correto funcionamento, a aplicação deve ter os arquivos de inicialização e montagem dos endpoints separados para evitar que a porta HTTP, utilizada pela aplicação, fique em uso após a execução dos testes.

Figura 6 - app.js.

```
const express = require('express')
const app = express()

app.use(express.json())

app.get('/', async (req, res) => {
  res.status(200).send('Bootcamp desenvolvedor back end - Tópicos especiais!')
})

module.exports = app
```

Figura 7 - server.js.

```
const app = require('./app')

const db = require('./db')

app.listen(5678, () => {
  console.log('Aplicação de exemplo ouvindo na porta 5678!')
})
```

Figura 8 - server.test.js.

```
const supertest = require('supertest')

const request = supertest('http://localhost:5678')

test('Servidor na porta 5678', async () => {
  const resposta = await request.get('/')
  expect(resposta.status).toBe(200)
})
```

O método de teste apresentado na **Figura 8 - server.test.js** apresenta uma única mudança em relação ao previamente observado nos testes unitários. Por se tratar de um teste que faz uma requisição, a função é declarada como assíncrona, utilizando o modelo **async/await**. Também seria possível utilizar uma abordagem com uma **promise**.

A implementação dos métodos de teste de integração utilizando o Jest segue a mesma lógica semântica da utilizada nos testes unitários. A maior diferença é dada pelo ponto de vista dos objetivos de cada teste.

2.2. Configuração e Desmontagem

Quando se está trabalhando com um framework de testes automatizados, é bem comum que o desenvolvedor precise executar cenários de configuração e desmontagem dos testes. Isso é requerido especialmente por dois motivos:

- Caso exista algum recurso a ser alocado e que viabilize o cenário desejado. Por exemplo, estabelecendo uma nova conexão com um banco de dados.
- Caso exista algum processamento necessário para iniciar os testes e que monte o cenário desejado. Por exemplo, inserindo dados no banco.

É interessante observar que, normalmente, quando é necessário fazer a montagem, quase sempre também será necessário realizar a desmontagem. O que nos exemplos acima significaria remover os dados de teste e desconectar do banco. Observe que realizar a montagem dentro do caso de teste, apesar de ser, por vezes, inviável, é bastante possível. Por outro lado, realizar a desmontagem é normalmente mais trabalhoso. Isso porque caso ocorra alguma falha dentro de um dos testes, ele será abortado e a execução do método não chegará até o fim. Desta maneira, um teste que falhasse poderia implicar em um recurso não desalocado corretamente, inviabilizando o ambiente.

As funções de configuração e desmontagem do Jest são:

- Configuração Geral – Utilizados em situações em que o processamento será utilizado para cada um dos testes.
  - beforeEach – Executado antes de cada um dos testes;
  - afterEach – Executado após cada um dos testes.
- Configuração Única – Utilizados em situações em que o processamento único é capaz de gerar a configuração para todos os testes.
  - beforeAll – Executado antes do primeiro teste;
  - afterAll – Executado após o último teste.

Figura 9 - Configuração e Desmontagem.

```
beforeEach(async () =>
  await db.cliente.destroy({ where: {} })
  await db.consulta.destroy({ where: {} })
)

afterAll(async () => await db.sequelize.close())
```

A **Figura 9 - Configuração e Desmontagem** apresenta um caso em que duas das funções de montagem/desmontagem funções foram utilizadas com finalidades distintas.

A função **beforeEach** chama o método **destroy** da biblioteca **sequelize** para duas entidades: cliente e consulta. Como a propriedade **where** do objeto enviado por parâmetro está vazia, todos os registros dessas entidades serão removidos. Dessa forma, é possível observar que o desenvolvedor deseja garantir que todos os testes serão inicializados sem nenhum registro das entidades cliente e consulta no banco de dados.

Já a função **afterAll** é chamada para que a conexão com o banco de dados seja desfeita ao final da bateria de testes e que, portanto, o recurso utilizado não esteja mais sendo utilizado.

Capítulo 3. Sistemas de Controle de Versão/Git

Sistemas de Controle de Versão (**SCV**) são aqueles que armazenam as informações a respeito das mudanças nos arquivos de um conjunto que é chamado de projeto. Isso permite que posteriormente as partes interessadas possam restaurar versões, desfazer mudanças, comparar a evolução do projeto ao longo do tempo, verificar quem foi o autor de uma alteração, entre outras coisas. A principal ideia de um SCV é dar a capacidade de alterar os arquivos do projeto com segurança e com pouco trabalho a mais.

3.1. Tipos de SCV

Uma das formas de se classificar um SCV é por meio da sua propriedade de distribuição dos dados:

- **Sistemas Locais:** Quando a base de dados contendo as versões dos arquivos alterados é armazenada em um diretório local. Apesar de muito simples, esse é um tipo de controle de alto risco, pois a perda dos dados armazenados localmente implicaria na perda das informações do projeto. Além disso, não permite a colaboração de diferentes profissionais no projeto;

- **Sistemas centralizados:** Os sistemas centralizados resolvem o problema da colaboração entre diferentes partes, por meio da implementação de um servidor central compartilhado no qual estão localizados os projetos. Cada membro pode fazer copias dos arquivos para sua máquina local e realizar alterações. Isso, porém, não impede que uma falha no servidor signifique a perda de informações;

- **Sistemas distribuídos:** Nos sistemas de controle de versão distribuídos, cada ponto armazena uma cópia completa do repositório de arquivos. Dessa forma, é garantida a maior segurança com relação à perda de informações. Além disso, as partes envolvidas podem se organizar em subgrupos menores e trabalhar em pontos específicos que serão posteriormente agrupados.

Além do mais, é permitido classificar os SCV de acordo com a forma com que armazenam as alterações entre as diferentes versões:

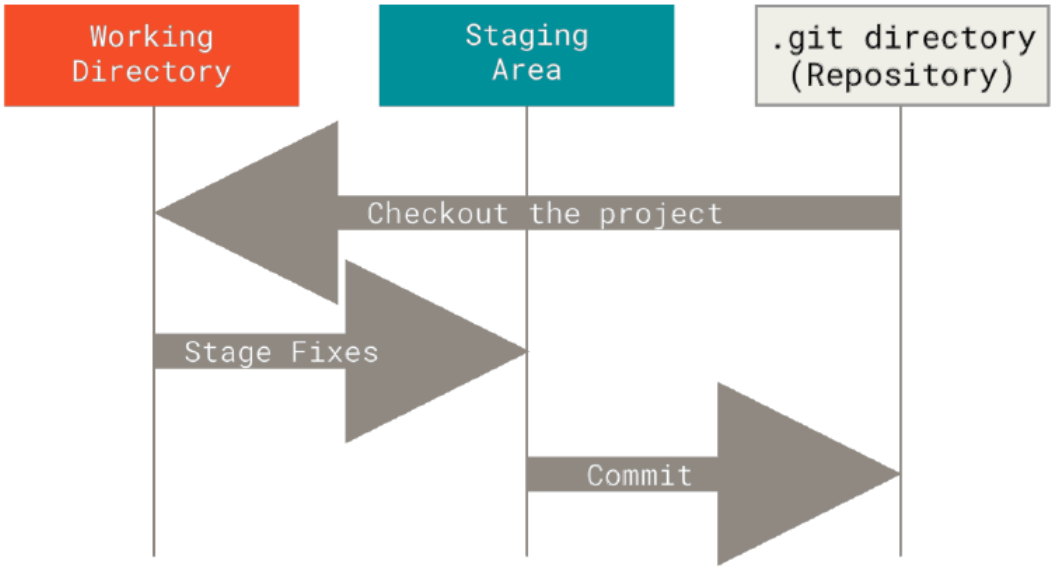
- Deltas: A grande maioria dos sistemas de controle de versão armazena as modificações realizadas entre as diferentes versões de um arquivo a cada nova versão. Essas modificações armazenadas são chamadas de deltas.

- *Snapshots*: Os sistemas baseados em snapshots, por outro lado, entendem os dados como uma miniatura do sistema de arquivos, no qual a cada novo commit é realizado um novo snapshot do sistema. Para ser eficiente, se os arquivos não mudaram, é possível que o sistema não armazene o arquivo novamente, apenas um link para o arquivo idêntico ao anterior já armazenado.

3.2. Git

O Git é um SCV distribuído, baseado em snapshots, e com suporte à verificação de integridade dos arquivos. Por isso, é considerado um sistema de controle de versão, seguro e de alta performance. Nesse sistema, cada arquivo pode estar em três estados possíveis: arquivos **modificados**, mas cujo commit ainda não foi realizado; arquivos marcados para serem submetidos no próximo commit, que são chamados de **staged**; e os arquivos **commitados**, que são aqueles armazenados de forma segura na base de dados.

Figura 10 - Seções do Git.



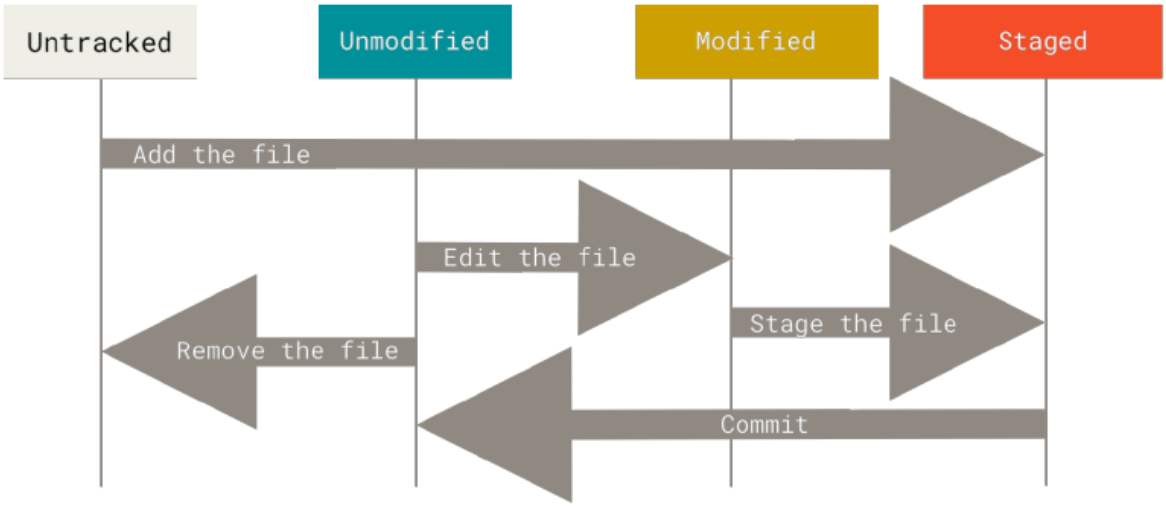
Fonte: ProGit.

A

Figura 10 - Seções do Git apresenta os possíveis fluxos de mudança de estados. Os arquivos do projeto, inicialmente, são comprimidos e armazenados no diretório '.git' juntamente com os metadados do projeto. Uma cópia desses arquivos é mantida no diretório de trabalho. Quando essa cópia do arquivo é alterada, o git consegue observar por meio de uma comparação com o arquivo original do repositório. Esse mesmo mecanismo é válido para arquivos alterados e removidos. Caso o usuário deseje confirmar essas alterações no próximo commit, ele deve acrescentá-las ao staging área, que é, na realidade, um arquivo armazenado dentro

do diretório do projeto com as informações a respeito do próximo commit a ser realizado.

Figura 11 - Ciclo de vida de um arquivo no Git.



Fonte: ProGit.

Além disso, é possível armazenar, em um arquivo chamado '.gitignore', uma lista com os arquivos ou padrões de diretórios e nomes de arquivos que se deseja ignorar.

3.3. Principais comandos Git

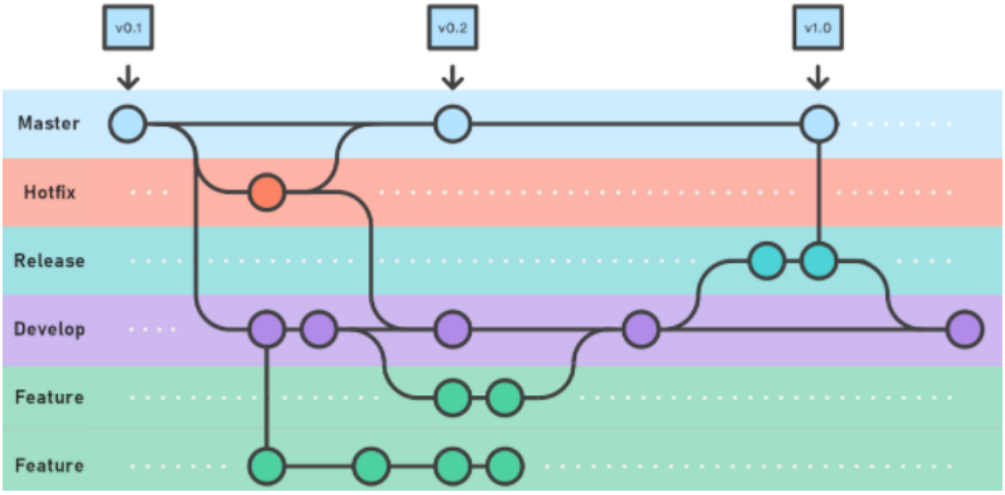
A interação do usuário com o Git se dá por meio da interface de comandos, cujas principais funções são listadas a seguir:

- **git init** - Inicializa um novo repositório, criando um arquivo.
- **git clone** - Realiza o clone de um repositório existente. Esse repositório é, então, adicionado à lista de repositórios remotos do repositório atual.
- **git status** - Exibe uma listagem com os arquivos que são considerados modificados, pelo git, de alguma maneira - alterados, removidos, staged, não rastreados.
- **git add** – Adiciona um novo arquivo ao projeto.
- **git rm** – Remove um arquivo do projeto.
- **git diff** – Exibe as mudanças realizadas.
- **git log** – Exibe o histórico de commits.
- **git commit** – Realiza um novo commit.
- **git fetch** – Obtém a versão atualizada do repositório remoto.
- **git push** – Solicita o envio dos arquivos locais ao repositório remoto.
- **git branch** – Realiza a criação de um novo branch.
- **git merge** – Realiza a combinação de diferentes branches em um único.

3.4. Branches de Trabalho

Para permitir que diferentes frentes de trabalho possam evoluir simultaneamente no mesmo repositório, o git permite a criação de branches.

Figura 12 - Linha do tempo do GitFlow.



Fonte: cronapp.

Branches são ramificações na linha de sequência de um arquivo, que permitem que o mesmo arquivo seja apontado por diferentes alterações em um determinado momento. Apesar de muito úteis, se usados de maneira incorreta, podem gerar confusão, acarretando perda de informações. A fim de reduzir a perda das informações presente nos branches, o padrão gitflow foi criado. Os branches no git flow são subdivididos em 5 grupos de acordo com sua durabilidade e utilização:

- **Master** - Contém o código da aplicação em produção.
- **Develop** - Contém o agrupado do que está sendo preparado pela equipe de desenvolvimento para a próxima versão.
- **Feature** - São as branches nas quais os desenvolvedores devem desenvolver as tarefas para, posteriormente, fazer o merge com a develop.
- **Hotfix** - É onde devem ser realizadas as correções críticas para entrar em produção (master) com um grau de urgência maior. Não seguem, portanto, toda o rito do fluxo de trabalho.
- **Release** - É onde são realizados os testes e ajustes da próxima versão. É gerado a partir de uma Develop e, posteriormente, reintegrado à master.

Capítulo 4. Contêineres

A distribuição de aplicativos pode ser uma tarefa árdua, dependendo do número de componentes e dependências envolvidas no processo. Quando o evento ocorre sob alta pressão por agilidade e qualidade, a experiência tende a se tornar especialmente traumática.

Neste capítulo, é apresentado o conceito de contêineres, abordando os principais aspectos que têm levado à sua ampla utilização pela indústria. Além disso, são apresentados os principais aspectos do docker, considerado a ferramenta padrão para se trabalhar com contêineres.

4.1. Contêineres e Máquinas virtuais

O container é um software empacotado, juntamente com todas as suas dependências, de maneira que a aplicação possa ser executada de forma segura, isolada e leve. Isso garante confiabilidade na transferência do software de uma máquina para outra. Toda a unidade do software é empacotada naquilo que é chamado de imagem. Uma imagem contém o que é necessário para executar a aplicação. Isso pode envolver o código fonte, scripts, bibliotecas e a configuração do ambiente em questão.

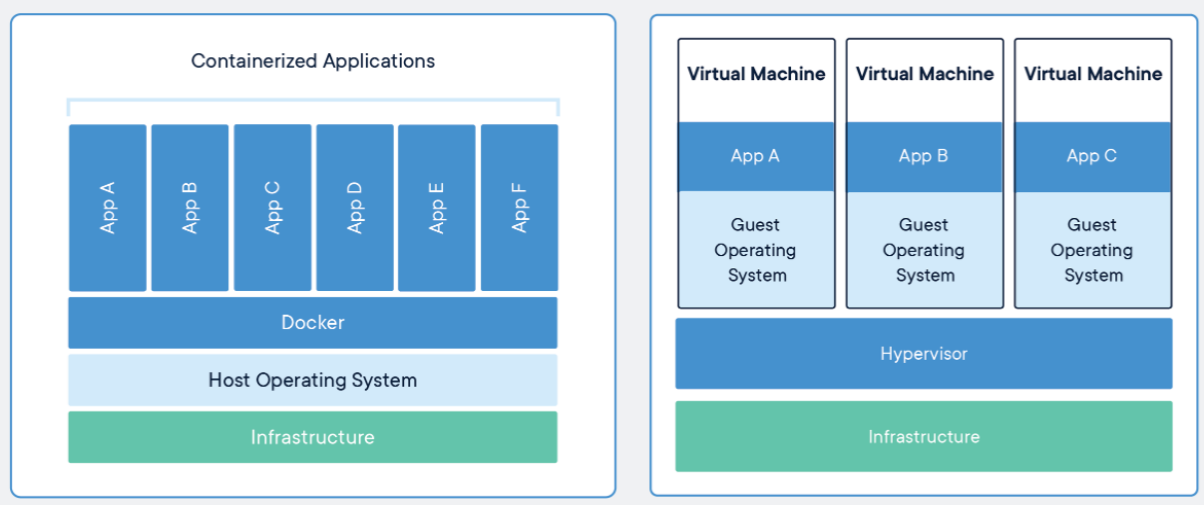
Quando uma imagem é executada (no caso do **docker**, pelo **docker engine**), um contêiner é criado. O software contido no **contêiner é uma instância da imagem** de origem e é capaz de garantir o funcionamento padronizado independentemente do ambiente em que está sendo executado.

O conceito de contêineres é frequentemente confrontado com a ideia de máquinas virtuais. **Máquinas virtuais executam softwares sob uma plataforma de hardware emulada, criada pelo seu monitor**. A esse monitor é dado o nome de *hypervisor*. Cada máquina virtual necessita, portanto, de um Sistema Operacional que seja capaz de fornecer acesso ao hardware para as aplicações em execução. Dessa forma, é possível que uma máquina real possa executar simultaneamente sistemas

operacionais diferentes (Linux e Windows, por exemplo), rodando, cada um, diferentes aplicações. Por meio desse mecanismo, as máquinas virtuais são capazes de fornecer ambientes isolados nos quais é possível garantir a consolidação da aplicação em um e maior capacidade de remontagem do ambiente em caso de perda das informações (disaster recovery).

O principal problema da abordagem do uso de máquinas virtuais é o alto consumo de recursos computacionais, uma vez que é necessária a execução de uma nova imagem do sistema operacional para cada imagem criada. Por outro lado, diferentes contêineres compartilham um único núcleo do mesmo sistema operacional, o que reduz o consumo de recursos. Em comparação às máquinas virtuais, os contêineres são, portanto, mais leves e mais facilmente portáveis.

Figura 13 - Contêineres x Máquinas Virtuais.



Fonte: docker.com.

4.2. Docker

A plataforma de contêineres **Docker**, criada pela empresa de mesmo nome, é, devido ao seu pioneirismo, considerada um padrão para que se trabalhe com contêineres. A **Tabela 2 - Conceitos e definições Docker** apresenta os conceitos e definições para as principais ferramentas do Docker.

Tabela 2 - Conceitos e definições Docker.

Conceito	Definição
Docker Engine	É a plataforma que realiza a instanciação do contêiner a partir de uma imagem e hospeda sua execução. Suporta diferentes sistemas operacionais como diferentes distribuições do Linux e o Windows.
Docker CLI	É uma ferramenta de linha de comando que permite a interação do usuário com a Docker Engine. É considerada a principal ferramenta para a criação e a manutenção de contêineres. Os comandos seguem a seguinte sintaxe: ***docker <comando> <subcomando> [opções]***
Docker Image	É um conjunto formado pela aplicação que se deseja executar, suas bibliotecas de dependência e as configurações necessárias ao ambiente.
Docker Container	É uma instância da imagem. É executado como um processo do sistema operacional que o hospeda. Podem, portanto, existir inúmeros contêineres em execução, e que tenham sido criados a partir da mesma imagem.
Docker Hub	Hospedado em https://hub.docker.com, é a plataforma onde estão hospedadas diversas imagens Docker, disponíveis para uso. São disponibilizadas, inclusive, imagens oficiais, criadas por fabricantes de software de ampla adoção no mercado, como SGBDs, servidores de aplicação e plataformas. A plataforma possui controle de versão das imagens para maior facilidade de uso.
Dockerfile	Arquivo que descreve os passos necessários para a criação de uma nova imagem. É um arquivo texto onde estão

	inseridos os comandos executados, normalmente buscando-se a partir de uma imagem base prévia. ***<Nome do Arquivo>.Dockerfile***
Docker Compose	É uma ferramenta utilitária que permite que sejam definidos serviços para aplicações que dependem simultaneamente de múltiplos contêineres que podem, inclusive, ser baseados em diferentes imagens. Dessa forma, com um único comando é possível carregar, sincronizar e configurar múltiplos containers. A definição dos serviços é feita por meio de um arquivo texto no formato .yml.

4.3. Principais comandos Docker CLI

As principais atividades realizadas pelo Docker CLI envolvem o acesso a imagens e o gerenciamento dos contêineres, por meio dos comandos a seguir:

- **docker image ls**: Exibe uma listagem das imagens existentes.
- **docker image pull <*imageName*>**: Realiza o download da imagem.
- **docker image inspect <*imageName*>**: Exibe as informações detalhadas da imagem.
- **docker container ls**: Exibe uma listagem com todos os contêineres em execução. Quando sucedido do parâmetro '-l', também serão exibidas as imagens paradas.
- **docker volume ls**: Exibe uma listagem com todos os volumes disponíveis.
- **docker volume inspect <*volumeId*>**: Exibe as informações detalhadas do volume informado.

- **docker network ls**: Exibe uma listagem com as redes disponíveis para uso pelos contêineres.
- **docker network connect**: Conecta um container a uma determinada rede.
- **docker network create:** Cria uma rede para comunicação entre os contêineres.
- **docker network disconnect**: Desconecta um contêiner de uma determinada rede.
- **docker system prune:** Realiza uma limpeza no ambiente.

4.4. Dockerfile

O docker permite a criação de novas imagens por meio da edição de um arquivo yml contendo instruções, como a imagem base para sua geração, arquivos a serem adicionados à imagem e comandos executados.

Figura 14 – Dockerfile.

```
FROM node:16

WORKDIR /usr/src/app

COPY package*.json ./
RUN npm install

COPY ./src/ ./src


EXPOSE 5678

CMD [ "npm", "start" ]
```

A **Figura 14 – Dockerfile** apresenta um exemplo simples de criação de uma nova imagem a partir da imagem node, versão 16. O arquivo detalha a cópia de arquivos, a porta exposta e a execução da aplicação.

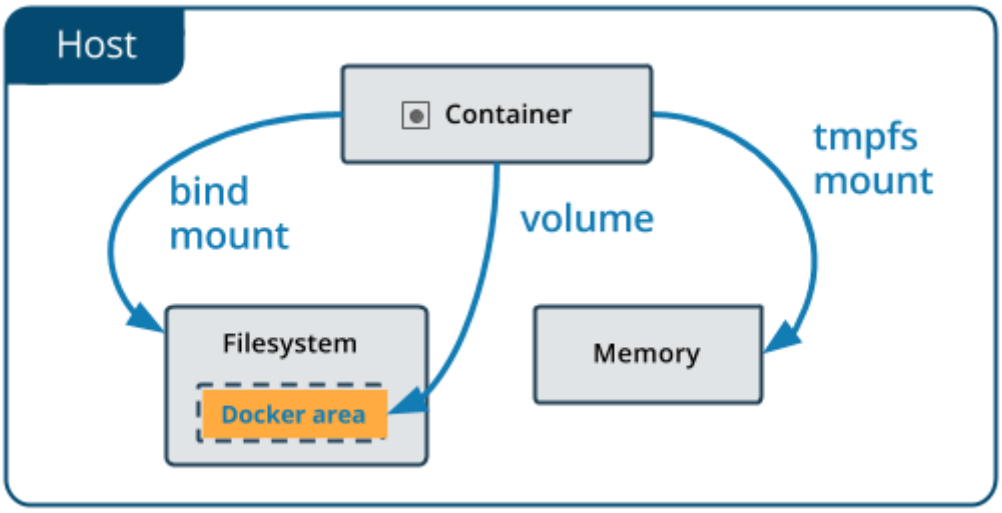
4.5. Volumes

Volumes são o principal mecanismo sugerido como boa prática para a persistência de dados ao trabalhar com contêineres.

Para os casos em que se está trabalhando com imagens que possuem volumes que armazenam dados não voláteis, é possível utilizar um diretório compartilhado do host com o contêiner. Assim, não é possível diminuir o risco de perda de dados por acidente ou mal uso. Além disso, o uso de um diretório padrão facilita a realização do trabalho de manutenção, movimentação dos arquivos e realização de backups.

Durante a criação de um novo contêiner, caso o parâmetro **-v** seja utilizado, a Docker engine irá realizar o mapeamento entre o sistema de arquivos local e o utilizado pelo contêiner.

Figura 15 – Volumes.



Fonte: docker.com.

O exemplo abaixo mostra como utilizar um volume para obter os dados de uma base de dados postgres ao criar um contêiner utilizando o PowerShell do Windows. Neste exemplo, o diretório utilizado será o **'docker-volumes'** localizado dentro daquele no qual o comando está sendo executado - **$pwd**.

docker run --name some-postgres -e POSTGRES_PASSWORD=mysecretpassword -d -P -v $pwd/docker-volumes:/var/lib/postgresql/data postgres

4.6. Docker network

O docker engine cria, no ambiente do seu host, uma rede virtual agnóstica entre os contêineres em execução e, por padrão, conecta todos os contêineres na rede **bridge.** É possível, no entanto, realizar a criação e a exclusão de novas redes, bem como a adição e a remoção de contêineres dessas redes.

Alguns dos comandos de rede mais básicos da CLI do docker são apresentados em **4.3. Principais comandos Docker** CLI.

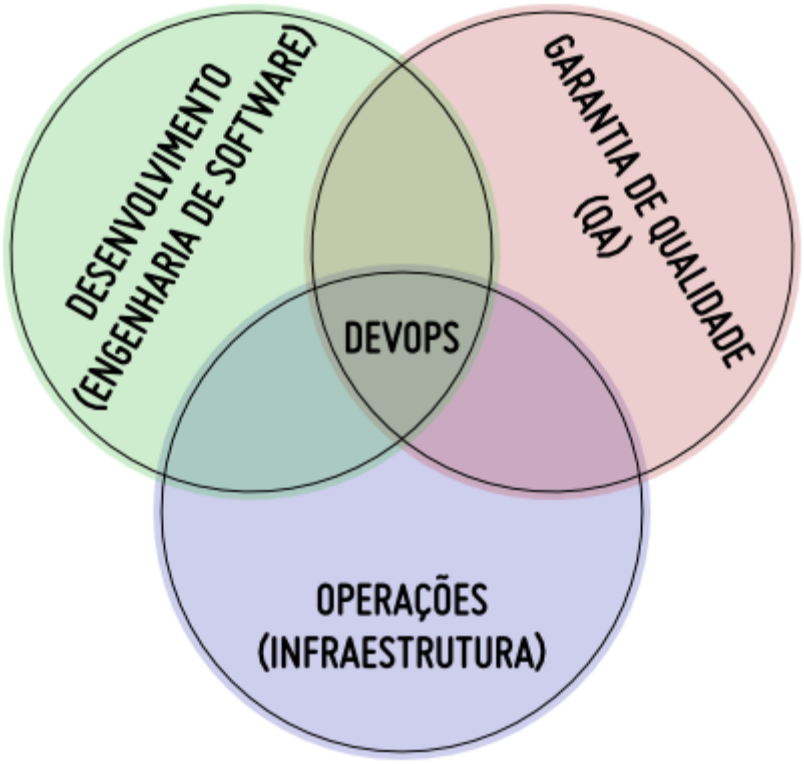
4.7. Docker Compose

Quando se trabalha com contêineres, é bastante comum que sejam criados serviços que possuem interdependência entre si. Para criar um serviço que possa operar com múltiplos contêineres de múltiplas instâncias simultaneamente, é possível utilizar uma ferramenta chamada Docker Compose. Com o Docker Compose, ao editar um arquivo no formato yml, o desenvolvedor pode definir portas e volumes, de maneira semelhante ao que é realizado no Dockerfile. Porém, essa definição pode ser feita para mais de uma imagem. Além disso, ainda é permitido definir a relação entre essas imagens, como a dependência na instanciação dos contêineres.

Capítulo 5. A Cultura DevOps

DevOps é um modo de pensar, uma cultura, uma forma de enxergar os processos de desenvolvimento de software, no qual os desenvolvedores (Dev) e a equipe de operação (Ops) - também chamados de operadores do sistema - devem trabalhar em conjunto, buscando gerar ganhos de qualidade e na agilidade das entregas. Muito frequentemente, os profissionais de QA também são adicionados como fundamentais dentro dessa cultura, sem que, no entanto, a sigla seja alterada.

Figura 16 - Intersecção dos Times DevOps.



Fonte: Wikipédia.

Está intimamente ligada ao uso de metodologias de desenvolvimento ágeis, ferramentas de virtualização dos ambientes de produção e desenvolvimento (frequentemente com **computação em nuvem**), utilização massiva de automatização de processos. Esta última especialmente realizada por meio de CI/CD:

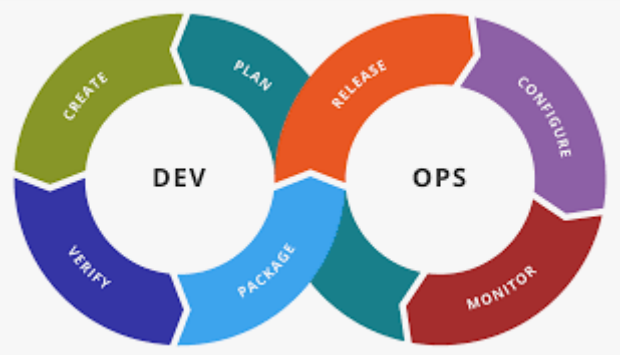
- Integração Contínua (**CI**): Boa prática de integrar os arquivos nas diferentes ramificações do Sistema de Controle de Versão da forma mais rápida possível

e deve ser verificada de forma automatizada. Dessa forma, é possível verificar se as novas funcionalidades desenvolvidas não quebram outras, ou se não são incompatíveis com alguma tarefa ocorrendo em paralela por outro desenvolvedor. Muitos times acham que essa abordagem leva a uma significante redução nos problemas de integração e permite que um time desenvolva software coeso mais rapidamente.

- Entrega Contínua (**CD**): Uma vez que é possível realizar a integração do código gerado de forma contínua, deve ser também possível entregar os artefatos produzidos e testados de forma automatizada e contínua. Essa entrega pode ser vista para os distintos momentos do processo do ciclo de vida de desenvolvimento de software, podendo tanto referir-se à liberação para testes (QA - **d**elivery) quanto usuário final (**d**eployment).

A principal vantagem da adoção dessa cultura em uma empresa pode ser diretamente pelos desenvolvedores diante da redução do nível de stress ao qual as equipes são submetidas no momento de lançamento de novos produtos, ou pacotes de atualizações. Mas, muito além dessa visão do desenvolvedor, a prática leva à maior qualidade das entregas e menores custos e riscos.

Figura 17 - Etapas DevOps.



Fonte: Wikipédia.

O DevOps é, portanto, uma agregação de valores, ferramentas, processos e boas práticas que visa encurtar o tempo que uma empresa gasta para liberar novas versões e monitorar sua performance de forma contínua e com garantia de qualidade.

Referências

BECK, Kent. **TDD Desenvolvimento Guiado por Testes**. Bookman, 16 de junho de 2010.

BECK, Kent. **Programação Extrema (XP) Explicada**: Acolha As Mudanças. Bookman, 26 de fevereiro de 2004.

CHACON, Scott; STRAUB, Ben. **ProGit:** Everything you need to know about Git. 2. ed. Apress, 9 de novembro de 2014.

CLI, **docker cli docs**, 2021. Disponível em: <https://docs.docker.com/engine/reference/commandline/cli/>. Acesso em: 27 jul. 2021.

FOWLER, Martin. **Padrões de Arquitetura de Aplicações Corporativas**. Bookman, 9 de julho de 2018.

GET-STARTED. **Docker**, 2021. Disponível em: <https://www.docker.com/get-started>. Acesso em: 27 jul. 2021.

GITHUB LEARNING LAB. **GitHubLabs**, 2021. Disponível em: <https://lab.github.com/>. Acesso em: 27 jul. 2021.

INICIANDO no Jest. **JestJS**, 2021. Disponível em: <https://jestjs.io/pt-BR/docs/getting-started>. Acesso em: 27 jul. 2021.

JEST. **npmjs**, 2021. Disponível em: <https://www.npmjs.com/package/jest>. Acesso em: 27 jul. 2021.

SUPERTEST. **npmjs**, 2021. Disponível em: <https://www.npmjs.com/package/supertest>. Acesso em: 27 jul. 2021.

TRAINING AND EDUCATION. **Heroku**, 2021. Disponível em: <https://www.heroku.com/training-and-education>. Acesso em: 27 jul. 2021.