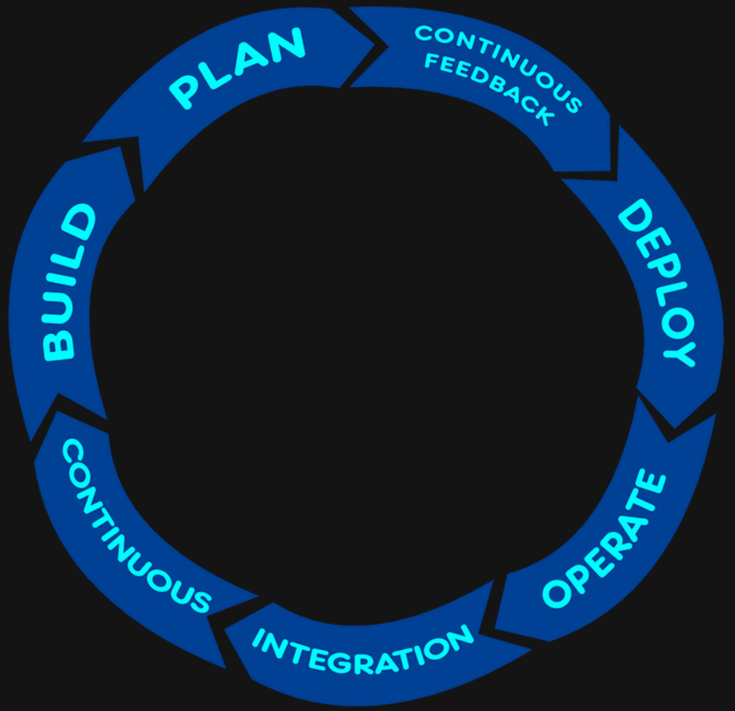
Guia de Informações

DevOps

A base da automação e colaboração

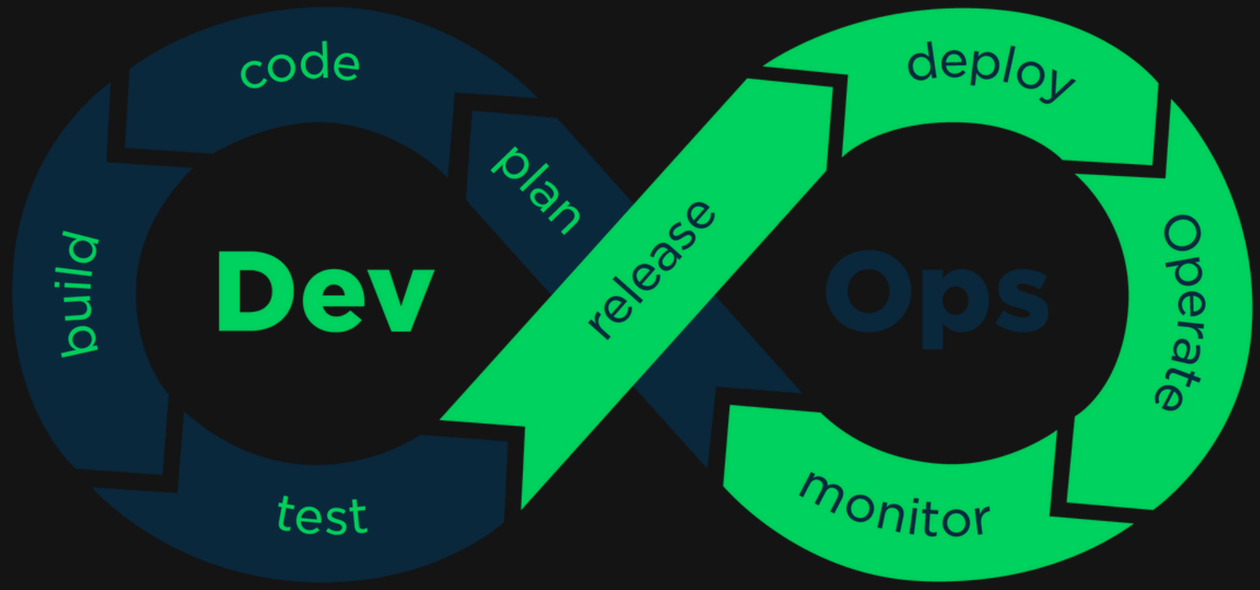


ELABORADORES

NELSON FELIPE ANDRADE ARAÚJO

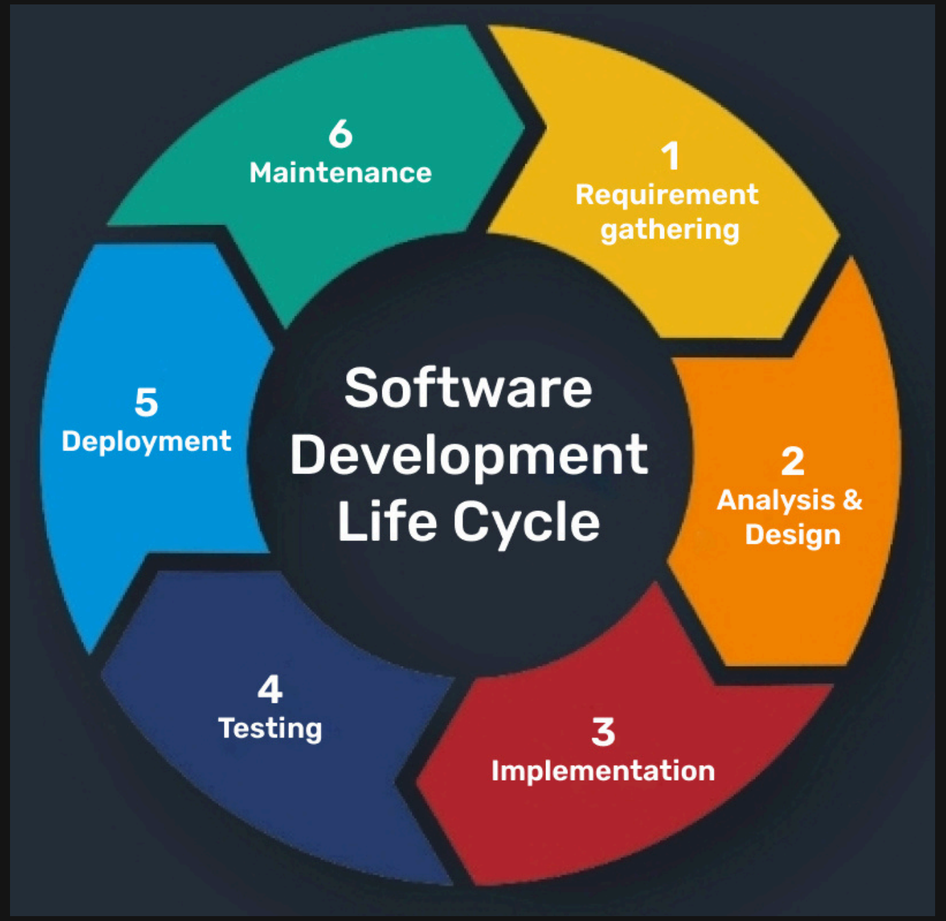
JOSÉ LEVY RODRIGUES DA SILVA

CONCEITOS DE DEVOPS



CÍCLO DE VIDA DE UM SOFTWARE

1. COLETA DE REQUISITOS (REQUIREMENT GATHERING)
2. ANÁLISE E PROJETO (ANALYSIS & DESIGN)
3. IMPLEMENTAÇÃO (IMPLEMENTATION)
4. TESTES (TESTING)
5. IMPLANTAÇÃO (DEPLOYMENT)
6. MANUTENÇÃO (MAINTENANCE)



COLETA DE REQUISITOS

Levantar requisitos é a fase inicial e a base de todo o projeto.
O objetivo é entender o que o software precisa fazer. Envolve coletar todas as informações e exigências dos stakeholders. O resultado desta fase é, geralmente, um documento que especifica detalhadamente tudo o que o sistema deve ser capaz de realizar.

stakeholders = indivíduos, grupos ou organizações que têm interesse direto ou indireto numa empresa ou projeto.

ANÁLISE E PROJETO

- Análise: A equipe estuda a viabilidade dos requisitos que foram levantados e com base neles se organiza.

- Projeto (Design): Cria-se a "planta" do software. Os arquitetos e desenvolvedores definem a estrutura geral do sistema, o design da interface, a arquitetura do banco de dados e outras especificações técnicas. O objetivo é ter um plano claro de como o software será construído.

IMPLEMENTAÇÃO

Fase de codificação. Os desenvolvedores pegam as especificações de design da fase anterior e começam a escrever o código-fonte do software. É nesta etapa que o software começa a ser construído de fato, transformando o projeto em um produto funcional.

TESTES

Após a implementação, o software passa por testes rigorosos para garantir sua qualidade. A equipe de QA (Quality Assurance) procura por defeitos, bugs e verifica se o software atende a todos os requisitos definidos na primeira fase. Se erros forem encontrados, eles são reportados aos desenvolvedores para correção.

IMPLEMENTAÇÃO

Uma vez que o software foi testado e aprovado, ele está pronto para ser lançado. Nesta fase, o software é instalado no ambiente de produção, o que significa que ele é disponibilizado para os usuários finais. Esta etapa pode envolver a configuração de servidores, bancos de dados e a migração de dados, se necessário.

MANUTENÇÃO

A fase de manutenção começa assim que o software está em produção:

- Correção de bugs: Resolver problemas que não foram encontrados durante a fase de testes.
- Atualizações: Fazer melhorias no sistema ou adaptá-lo a novas necessidades.
- Suporte: Estar disposto a ajudar os usuários a utilizar o software.

O ciclo não termina após o lançamento!

MODELOS DE CRIAÇÃO DE SOFTWARE

ANTERIORMENTE

- MODELO EM CASCATA (WATERFALL)

ATUALMENTE

- SCRUM
- KANBAN
- XP

O QUE É DEVOPS?

É uma cultura e um conjunto de práticas que visam unificar o desenvolvimento de software (Dev) e as operações de tecnologia da informação(Ops). O objetivo principal é encurtar o ciclo de vida de desenvolvimento e fornecer uma entrega contínua de software com alta qualidade. Em essência, o DevOps promove colaboração mais direta entre as equipes de desenvolvimento e operações, que tradicionalmente trabalham de forma dividida.

SURGIMENTO DO DEVOPS

O DevOps tem seu surgimento entre os anos de 2007 e 2009, como uma evolução natural e uma resposta direta às limitações dos modelos tradicionais de desenvolvimento de software, que não conseguiam atender à crescente necessidade de velocidade e eficiência do mercado. Dessa forma, as equipes de Desenvolvimento (Dev) e Operações (Ops) possuem uma maior interação.

DevOps = Desenvolvimento (Dev) + Operações (Ops)

FERRAMENTAS DE DEVOPS

1. PLANEAMENTO E COLABORAÇÃO: JIRA, CONFLUENCE, TRELLO.
2. CONTROLE DE VERSÃO: GIT, GITHUB, GITLAB, BITBUCKET.
3. INTEGRAÇÃO E ENTREGA CONTÍNUA (CI/CD): JENKINS, GITHUB ACTIONS, CIRCLECI.
4. CONTEINERIZAÇÃO E ORQUESTRAÇÃO: DOCKER, KUBERNETES.
5. INFRAESTRUTURA COMO CÓDIGO: TERRAFORM, ANSIBLE, PUPPET, CHEF.
6. MONITORIZAÇÃO E OBSERVABILIDADE: PROMETHEUS, GRAFANA, ELK STACK , DATADOG, NEW RELIC.
7. COMPUTAÇÃO NA NUVEM (CLOUD COMPUTING): PROVEDORES COMO AWS, MICROSOFT AZURE E GOOGLE CLOUD PLATFORM

CONTEÚDOS DO MINICURSO

(Versionamento de código)

Git & GitHub

(CI/CD)

GitHub Actions

(Conteinerização)

Docker

GIT E GITHUB

INTRODUZINDO GIT

Git é um Sistema open-source de gerenciamento de versões
Foi desenvolvido pelo Linus Torvards (desenvolvedor do Linux)
Criado em 2005 para gerenciar o desenvolvimento dos códigos do Linux

O GIT É TIPO UMA MÁQUINA
DO TEMPO PRO SEU CÓDIGO.

PORQUE USAR GIT?

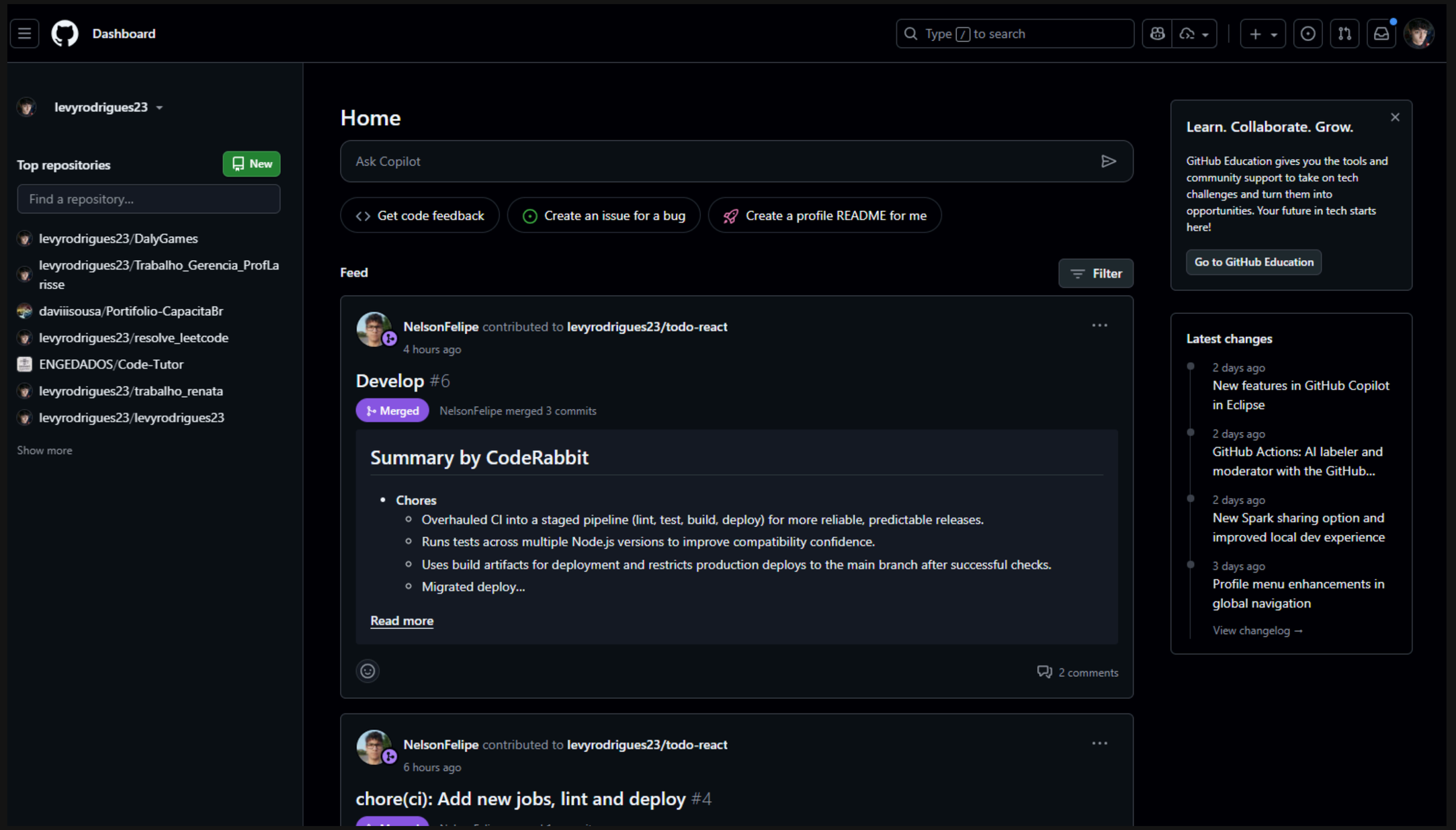
- Controle de versão – volta a qualquer versão do código.
- Trabalho colaborativo – várias pessoas trabalhando simultaneamente.
- Backup seguro – código salvo local e na nuvem.
- Histórico detalhado e rastreável – quem fez, quando e por quê.
- Integração com outras ferramentas – CI/CD e deploy automático.

INTRODUZINDO GITHUB

O **GitHub** é uma plataforma de hospedagem de código-fonte e colaboração que utiliza o Git para controle de versão. É o maior repositório de código do mundo, onde milhões de desenvolvedores colaboram em projetos open-source e privados.

O GIT HUB É ONDE SE HOSPEDA OS REPOSITÓRIOS

AMBIENTE DO GITHUB



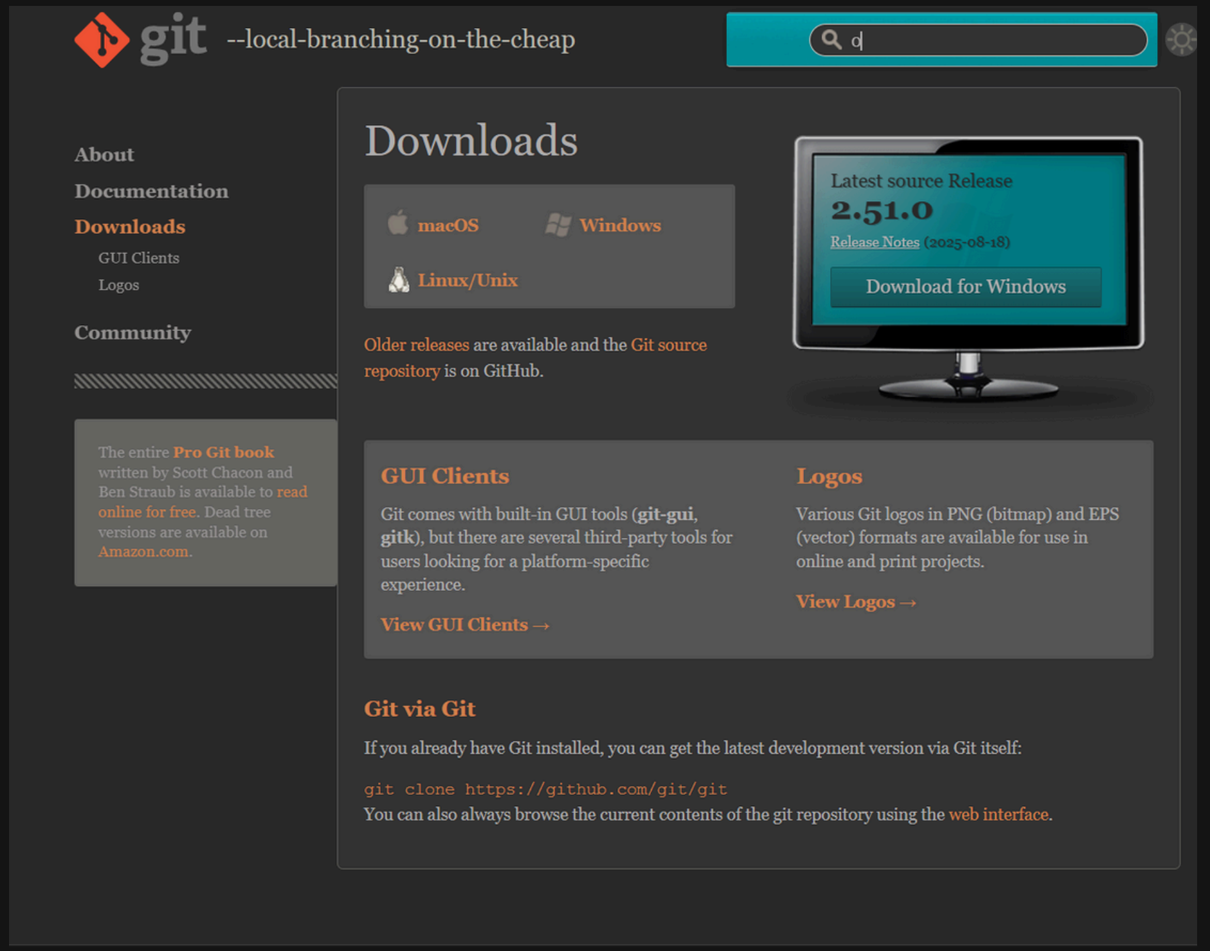
PORQUE USAR GITHUB?

- Repositórios remotos – teu código fica disponível online, acessível de qualquer lugar.
- Colaboração facilitada – issues, pull requests, comentários, revisão de código.
- Comunidade e portfólio – mostrar projetos, contribuir em open source, fortalecer currículo.
- Integração com Git – sincroniza tuas branches e commits locais com o remoto.

git +

INSTALAÇÃO DO GIT

- Acessar a documentação do git
- Executar o processo de instalação a partir do sistema operacional da máquina



INSTALAÇÃO DO GIT

💻 WINDOWS

- Acesse o site oficial:

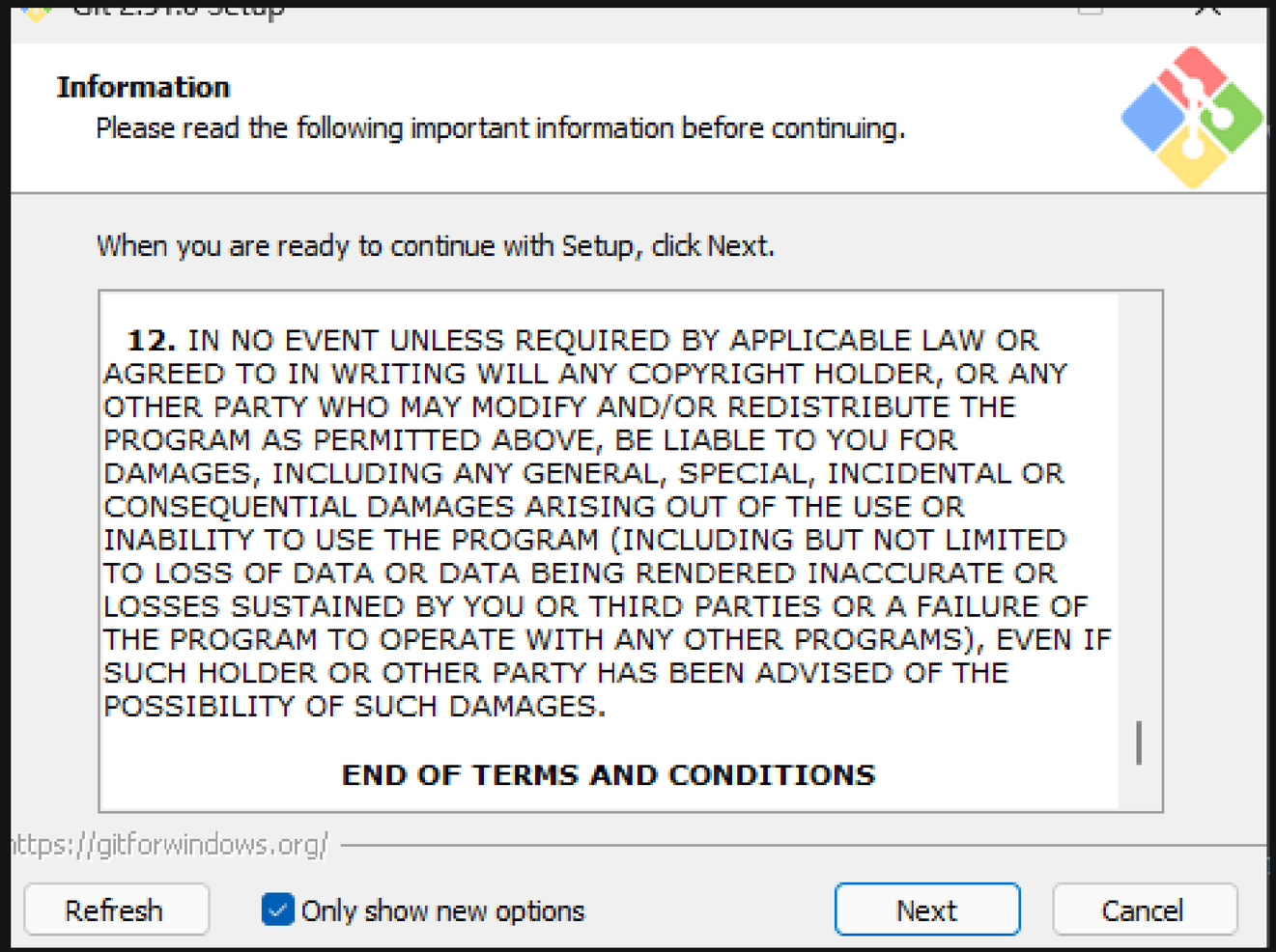
Vá para: https://git-scm.com/downloads
Clique em "Download for Windows"
O download começará automaticamente

- Execute o instalador:

Rode o arquivo .exe baixado
Aceite as configurações padrão (são ideais para iniciantes)
Clique "Next" em todas as telas

- Na tela "Adjusting your PATH environment": deixe "Git from the command line and also from 3rd-party software"
- Finalize a instalação

INSTALAÇÃO DO GIT



INSTALAÇÃO DO GIT

LINUX

```
# Atualize os pacotes
sudo apt update

# Instale o Git
sudo apt install git
```

Verifique a versão do git para ver se ele foi instalado

- git --version

INSTALAÇÃO DO GIT

POR QUE CONFIGURAR?

Antes de usar o Git, você PRECISA se identificar. Isso é obrigatório porque:

- Cada commit que você faz fica "assinado" com seu nome e email
- Colaboradores sabem quem fez cada mudança
- GitHub conecta seus commits ao seu perfil
- Histórico fica organizado e rastreável

INSTALAÇÃO DO GIT

⚙ CONFIGURAÇÕES OBRIGATÓRIAS

Use seu nome real (não nickname)
Coloque entre aspas se tiver espaços
Será exibido no GitHub e em todos os commits

```
git config --global user.name "José Silva"
```

INSTALAÇÃO DO GIT

⚙ CONFIGURAÇÕES OBRIGATÓRIAS

Use o mesmo email da sua conta GitHub,
senão seus commits não aparecerão conectados ao seu perfil

```
git config --global user.email "jose.silva@email.com"
```

INSTALAÇÃO DO GIT

VERIFICAR SE FUNCIONOU

```
git config --global user.name
# Deve mostrar: José Silva

git config --global user.email
# Deve mostrar: jose.silva@email.com
```

Apagar credenciais

Com esse comando é possível apagar necessariamente todas as credenciais

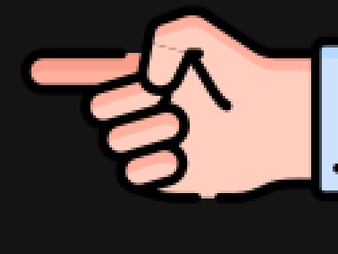
```
git config --global --unset user.name
git config --global --unset user.email
```

Conceitos fundamentais

- Repositório (repo) → pasta/projeto versionado.
- Repositório Local → fica na tua máquina.
- Repositório Remoto → fica hospedado online(neste caso, no GitHub)
- Versão → estado salvo do projeto (histórico de mudanças).
- Controle de versão → sistema que registra todas as alterações feitas no código.
- Histórico → linha do tempo com todos os commits feitos no projeto.

REPOSITÓRIO LOCAL ≠ REPOSITÓRIO REMOTO

Comandos básicos do git



git init

Cria um repositório Git dentro da pasta atual. É como se você dissesse: “A partir de agora, essa pasta vai ser monitorada pelo Git

```
git init
```

git add

Prepara os arquivos para o próximo commit
Move arquivos para a área de staging

```
git add arquivo.txt
git add .   # adiciona todos os arquivos modificados
```

git commit

Registra as mudanças no histórico do projeto com uma mensagem explicando o que foi feito

```
git commit -m "Adicionei a funcionalidade X"
```

git status

Mostra o estado atual do repositório: quais arquivos foram modificados, quais estão no staging e quais não estão.

```
git status
```

git remote

Ele serve para conectar o seu repositório local a um remoto, permitindo enviar (push) e receber (pull) alterações.

```
git remote add
origin
```

git log

Exibe o histórico de commits da branch atual, mostrando quem fez, quando fez e a mensagem do commit.

```
git log
```

git config

Define as configurações do Git, como nome e e-mail do autor dos commits. Normalmente feito só uma vez por máquina.

```
git config --global user.name "Seu Nome"
git config --global user.email "seuemail@email.com"
```

git push

Envia os commits locais para o repositório remoto

```
git push origin main
```

git pull

Atualiza o repositório local trazendo as alterações do repositório remoto. É como um “download” das mudanças.

```
git pull origin main
```

git clone

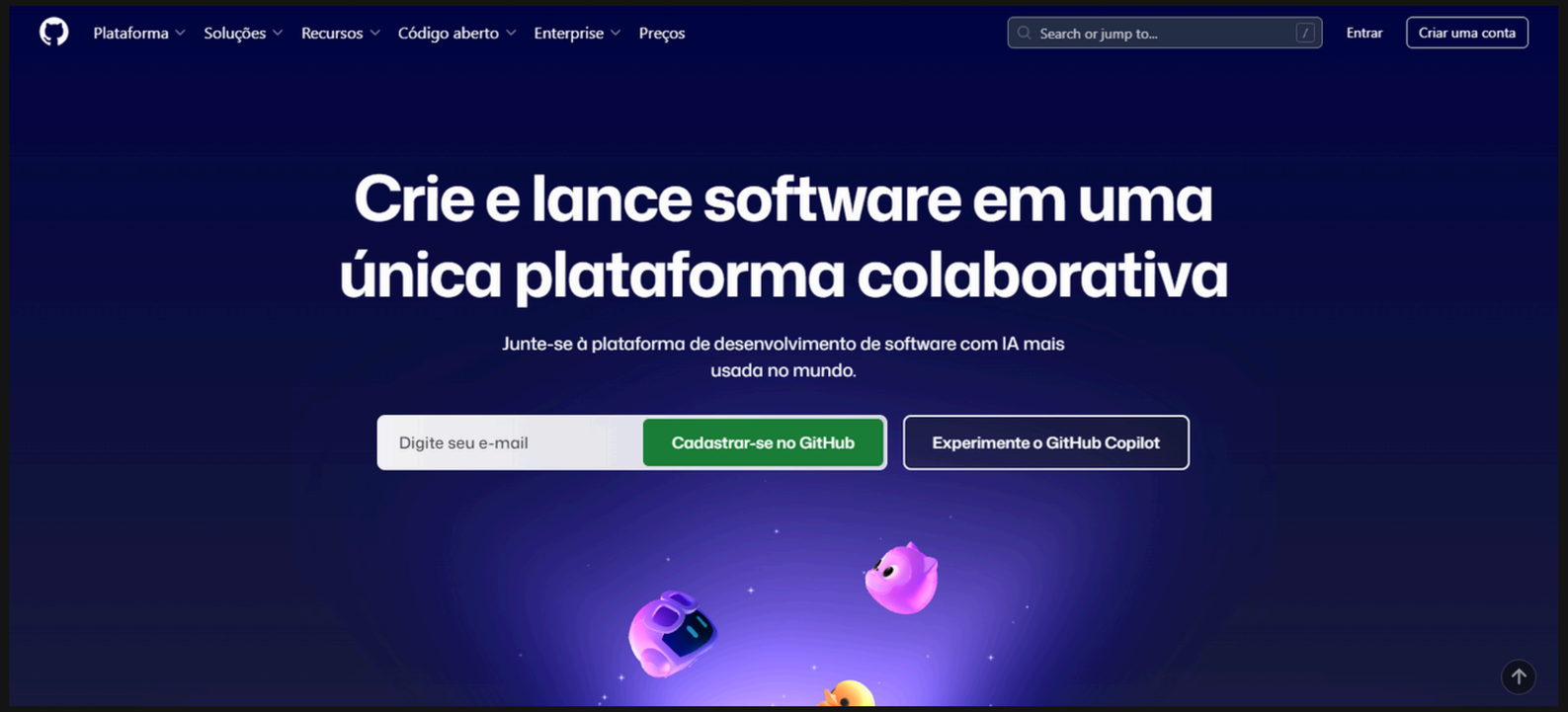
Clona o repositório remoto para a máquina local, trazendo consigo todos os arquivos possíveis (isso é meio óbvio já que é um clone)

```
git clone https://github.com/levyrodrigues23/todo-react.git
```

GITHUB NA PRÁTICA

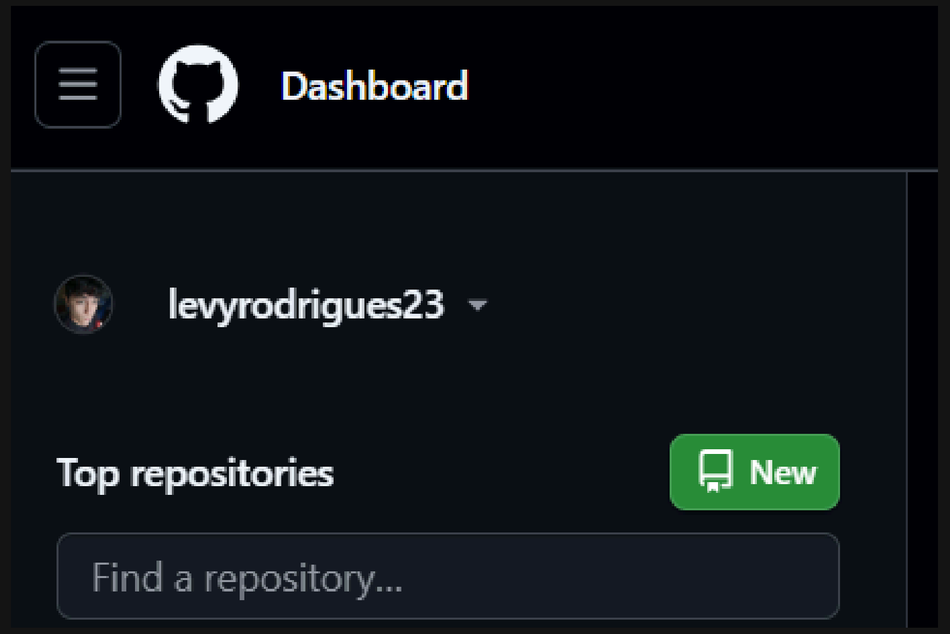
CRIANDO SUA CONTA NO GITHUB

- Acesse: https://github.com
- Clique em "Sign up"
- Escolha um username único (será sua marca!)
- Use o mesmo email configurado no Git
- Senha forte e segura



GITHUB NA PRÁTICA

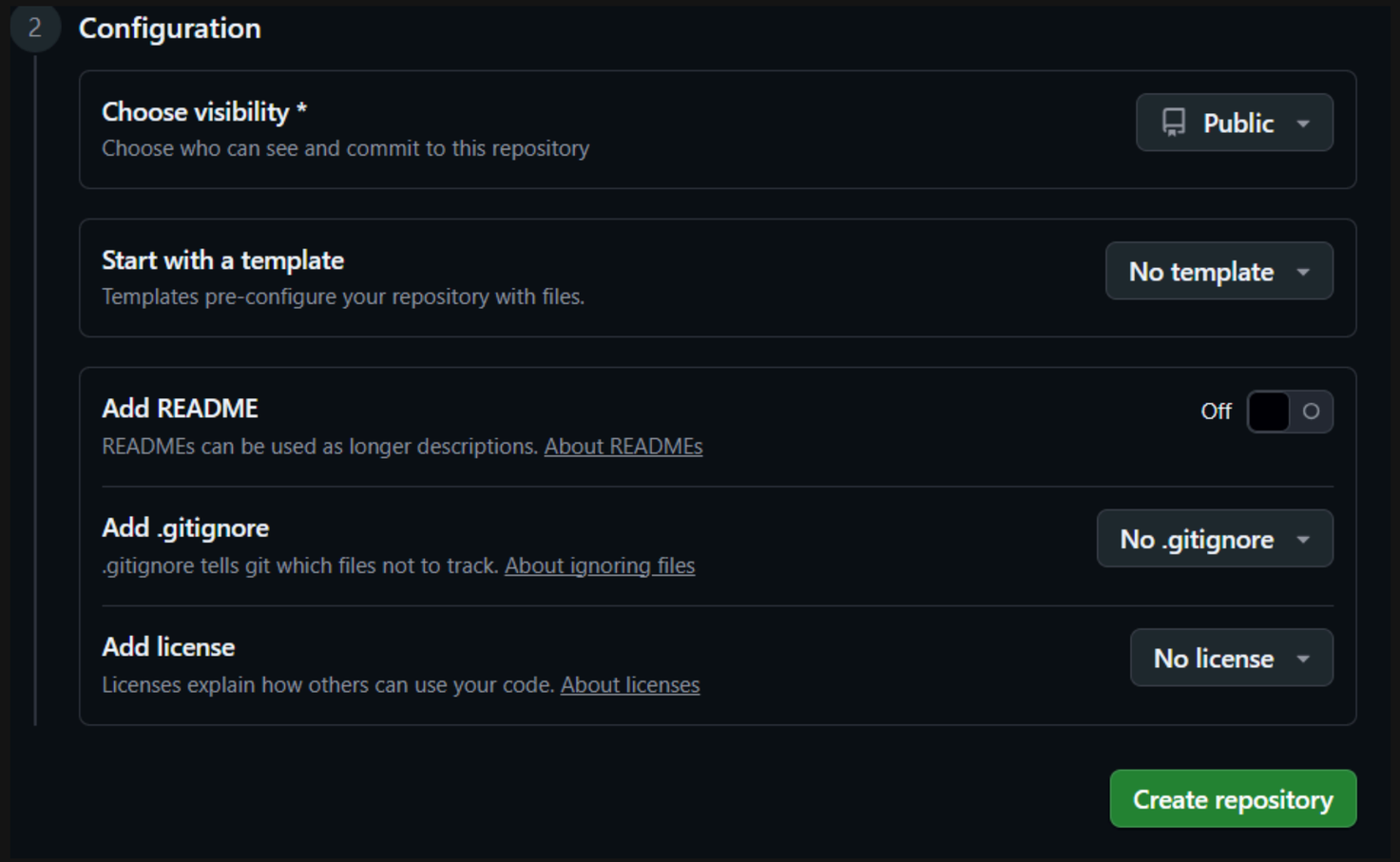
- Criando repositórios no GitHub



- Adicionando um nome

GITHUB NA PRÁTICA

- Configurações do repositório e Criar repositório



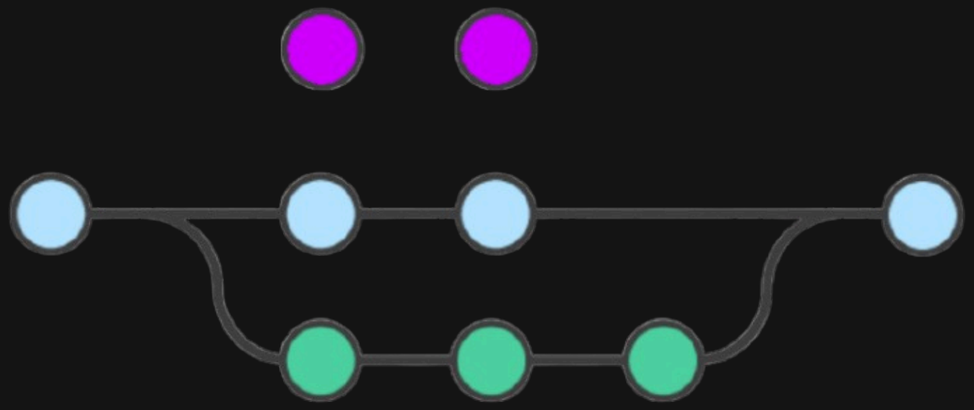
GITHUB NA PRÁTICA

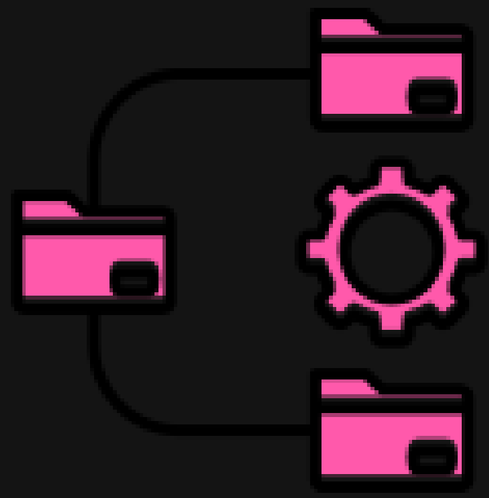
- Configuração do repositório na máquina local

COLABORAÇÃO NO GITHUB

Colaboração no GitHub é quando várias pessoas trabalham juntas em um mesmo projeto de forma organizada, usando ferramentas como branches, pull requests e issues pra propor mudanças, revisar código e acompanhar tarefas.

BRANCHES

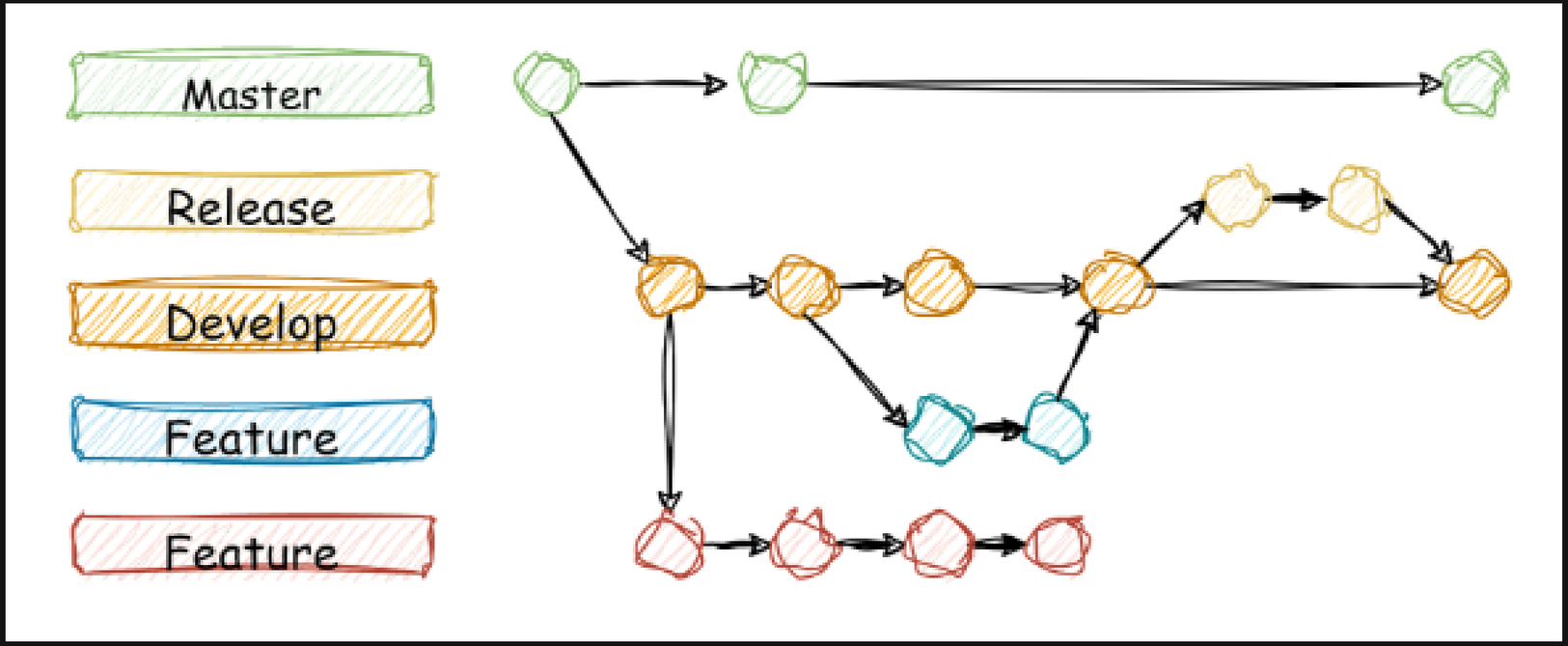




O que são branches

É uma linha de desenvolvimento paralela e independente
Permite trabalhar em diferentes funcionalidades sem afetar o código principal
Como ter "várias versões" do seu projeto ao mesmo tempo

BRANCH = "GALHO" OU "RAMIFICAÇÃO"



Por que usar Branches?

Desenvolvimento Seguro

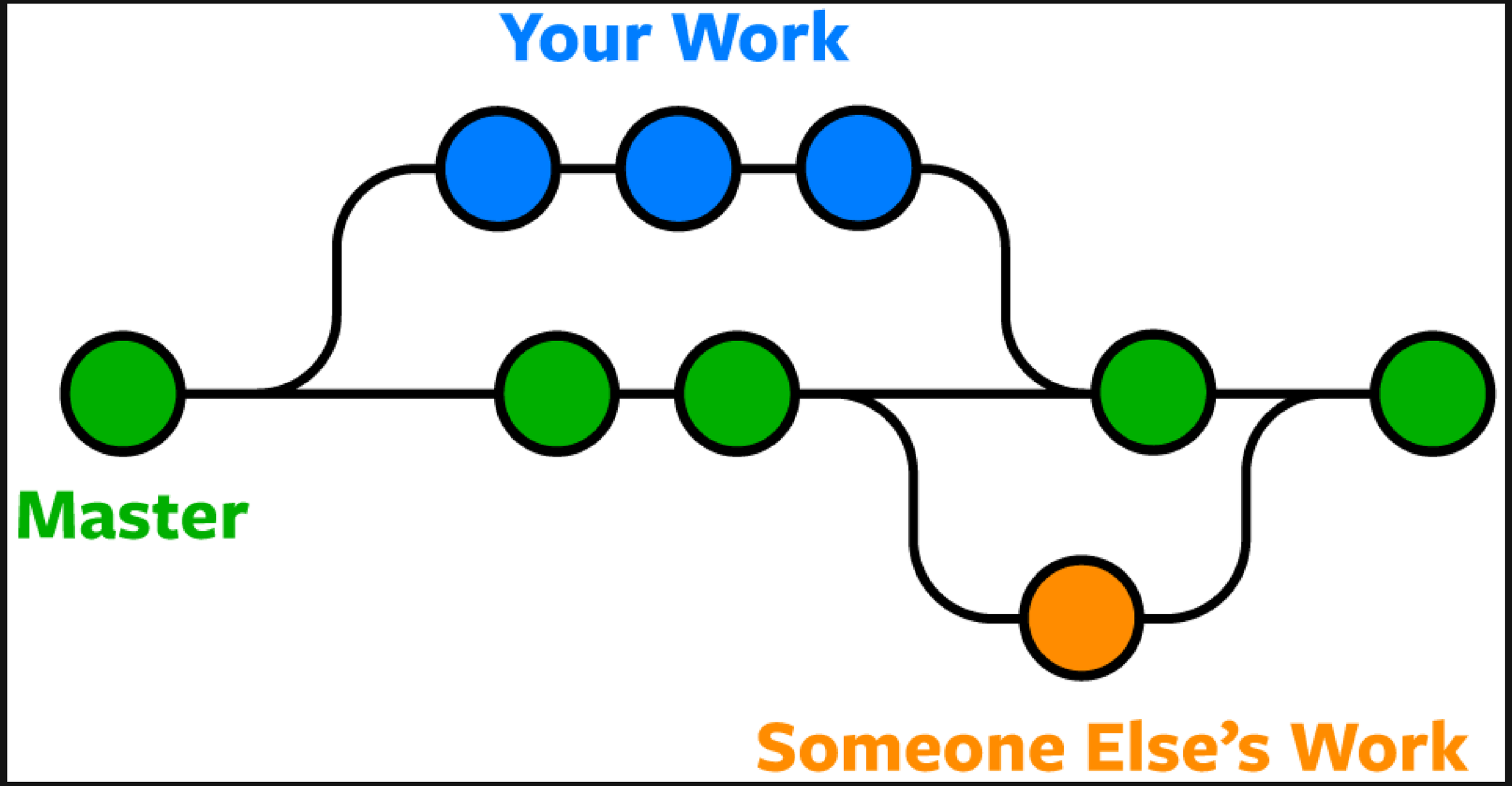
- Teste novas funcionalidades sem quebrar o código que já funciona
- Experimente sem medo - sempre pode voltar ao código estável

Colaboração Eficiente

- Cada desenvolvedor trabalha em sua própria branch
- Evita conflitos entre membros da equipe

Organização Clara

- Uma branch para cada funcionalidade/correção
- Histórico limpo e fácil de entender


Your Work
Master
Someone Else's Work

Comandos práticos

Comandos Essenciais

```
git branch                      # Lista todas as branches
git branch nova-funcionalidade # Cria uma nova branch
git checkout nova-funcionalidade # Muda para a branch
git checkout -b nova-feature   # Cria E muda para a branch (atalho)
git branch -d nome-branch      # Deleta uma branch
```

Comandos práticos

1. Criando uma Nova Funcionalidade:

```
git checkout -b adicionar-login
# Agora você está na branch "adicionar-login"
```

Comandos práticos

2. Trabalhando na Branch:

```
# Faça suas modificações...
git add .
git commit -m "Adiciona tela de login"
git push origin adicionar-login
```

Comandos práticos

3. Voltando para a Branch Principal:

```
git checkout main
# Agora você está de volta na branch principal
```

Comandos práticos

4. Quando sua funcionalidade está pronta:

```
git checkout main                 # Vai para a branch principal
git merge adicionar-login      # Traz as mudanças da sua branch
git branch -d adicionar-login # Deleta a branch (opcional)
```

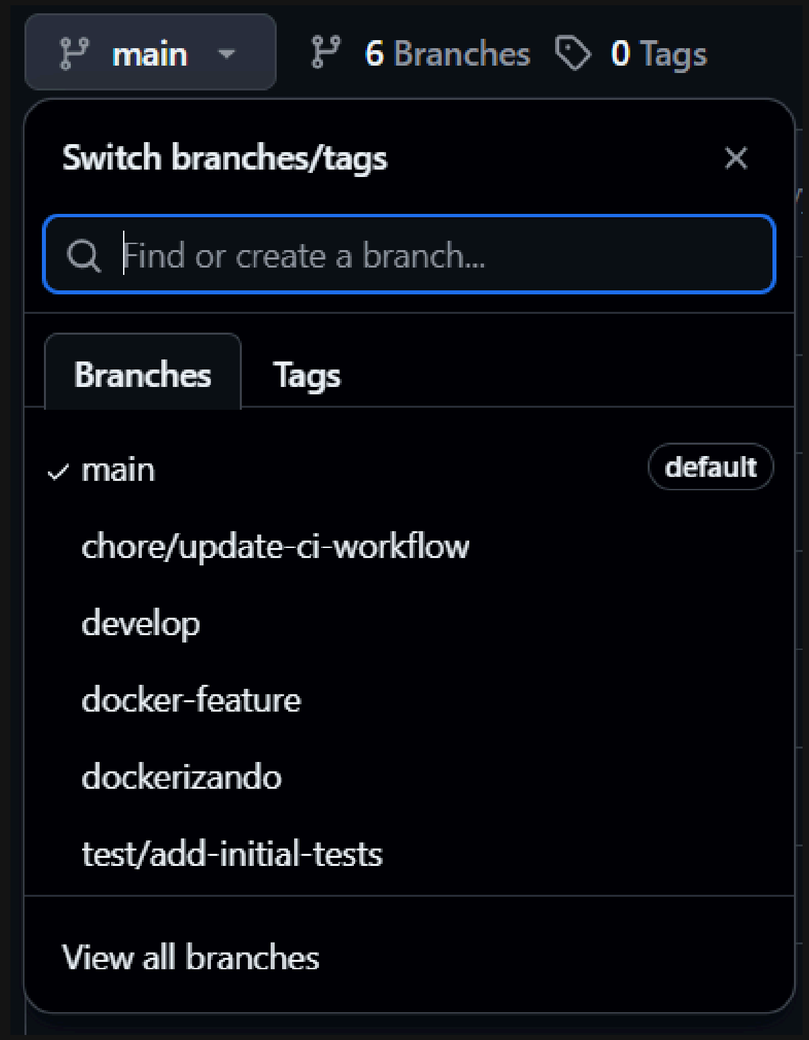
Boas práticas

O ato de usar **nomes descritivos** para as features que serão trabalhadas

```
✅ git checkout -b adicionar-autenticacao-google
❌ git checkout -b nova-coisa
```

Boas práticas

Branches Pequenas e Focadas
Uma funcionalidade por branch
Merge frequente para evitar conflitos grandes



Boas práticas

Uso do .gitignore

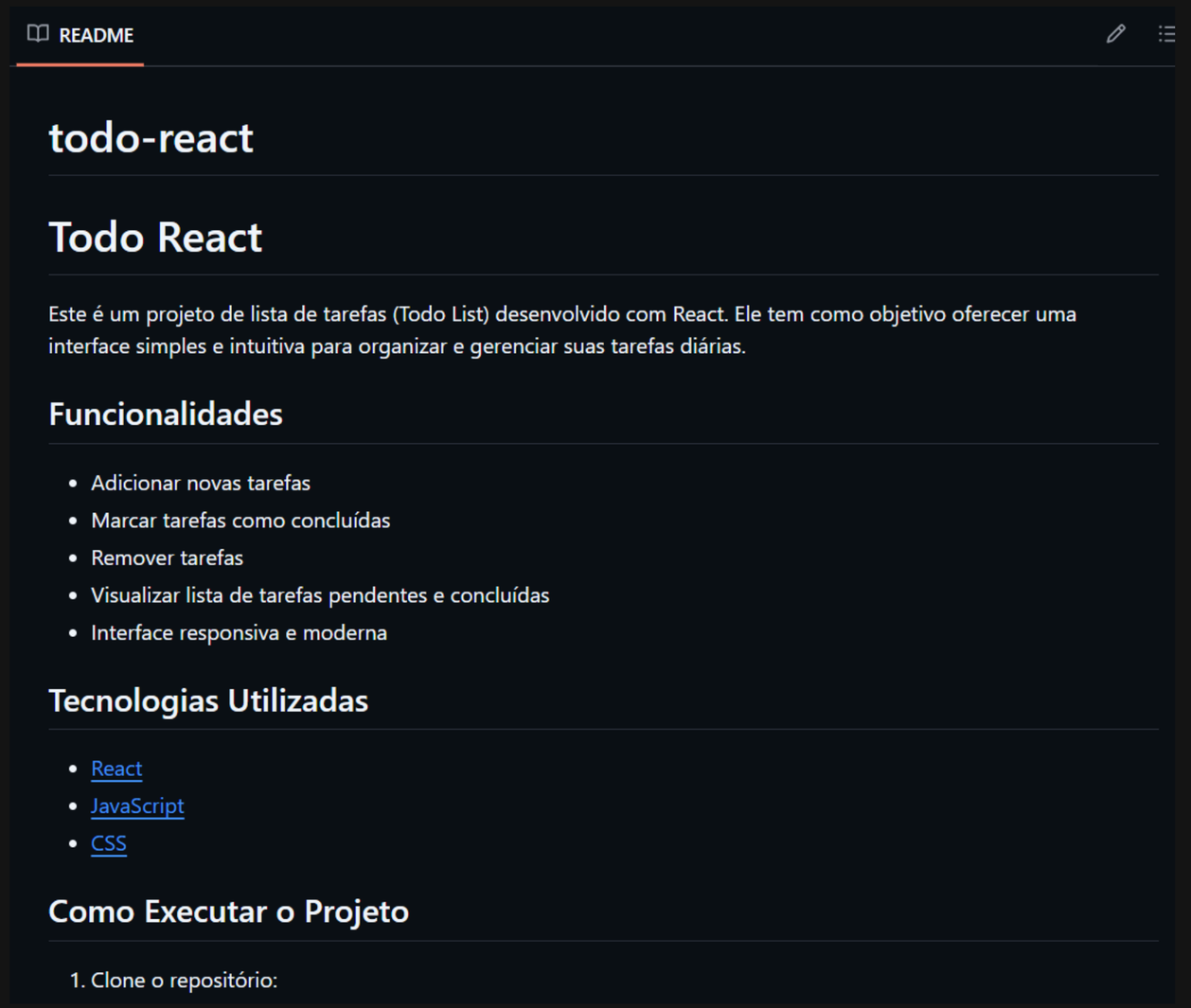
O .gitignore é responsável por guardar os arquivos que não serão expostos no repositório remoto

.gitignore

```
# Logs
logs
*.log
npm-debug.log*
yarn-debug.log*
yarn-error.log*
pnpm-debug.log*
lerna-debug.log*

node_modules
dist
dist-ssr
*.local

# Editor directories and files
.vscode/*
!.vscode/extensions.json
.idea
.DS_Store
*.suo
*.ntvs*
*.njsproj
*.sln
*.sw?

.vercel

```

Boas práticas

README.md bem escrito, destacando todo passo-a-passo do projeto



README

todo-react

Todo React

Este é um projeto de lista de tarefas (Todo List) desenvolvido com React. Ele tem como objetivo oferecer uma interface simples e intuitiva para organizar e gerenciar suas tarefas diárias.

Funcionalidades

- Adicionar novas tarefas
- Marcar tarefas como concluídas
- Remover tarefas
- Visualizar lista de tarefas pendentes e concluídas
- Interface responsiva e moderna

Tecnologias Utilizadas

- React
- JavaScript
- CSS

Como Executar o Projeto

1. Clone o repositório:

GITHUB NA PRÁTICA

- **Issues**

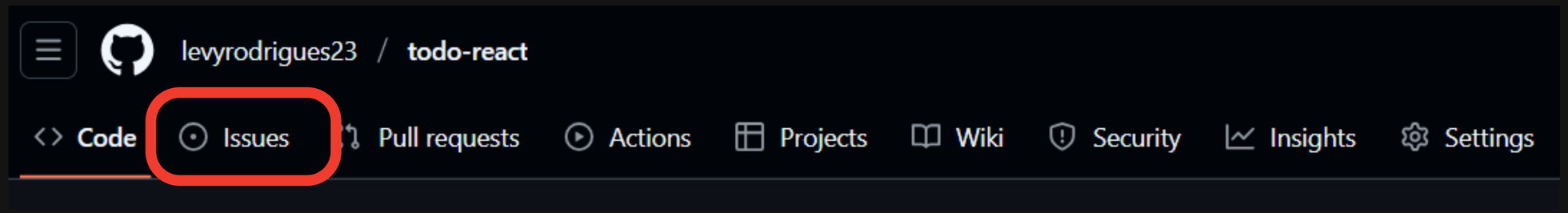
São como chamados/tarefas dentro do repositório. Dá pra usar pra relatar bugs, sugerir melhorias ou organizar o que precisa ser feito.
É tipo um quadro de “pendências” do projeto.

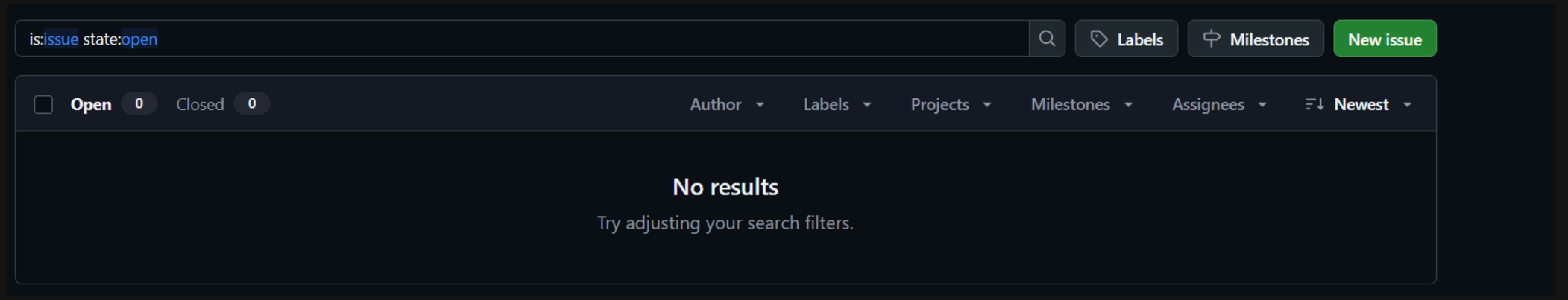
- **Pull Requests**

É quando alguém faz uma mudança no código (numa branch) e pede pra essa mudança ser revisada e juntada (merge) na branch principal. Serve pra discutir, revisar e aprovar código antes de ir pro projeto principal.

GITHUB NA PRÁTICA

Acessando as issues dentro do projeto





GITHUB NA PRÁTICA

Issue criada

Adicionar filtro de tarefas por prioridade #12

Open

levyrodrigues23 opened 3 minutes ago — Owner

Seria interessante implementar uma funcionalidade que permite ao usuário filtrar as tarefas pela prioridade definida (alta, média, baixa). Isso facilitaria a organização e o foco nas tarefas mais importantes.

Sugestão de funcionalidade:

- Adicionar um seletor de prioridade na interface de tarefas
- Permitir filtrar e exibir apenas tarefas de uma determinada prioridade
- Atualizar a documentação e exemplos, se necessário

Benefícios:

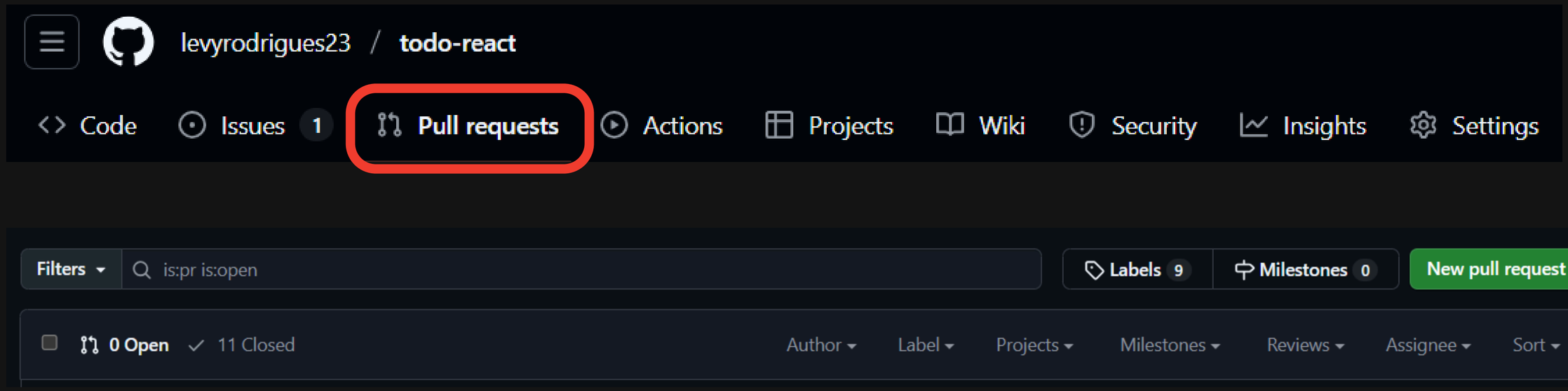
- Melhora a usabilidade
- Facilita a gestão das tarefas

Caso precise de mais detalhes ou exemplos, posso complementar.

Create sub-issue

GITHUB NA PRÁTICA

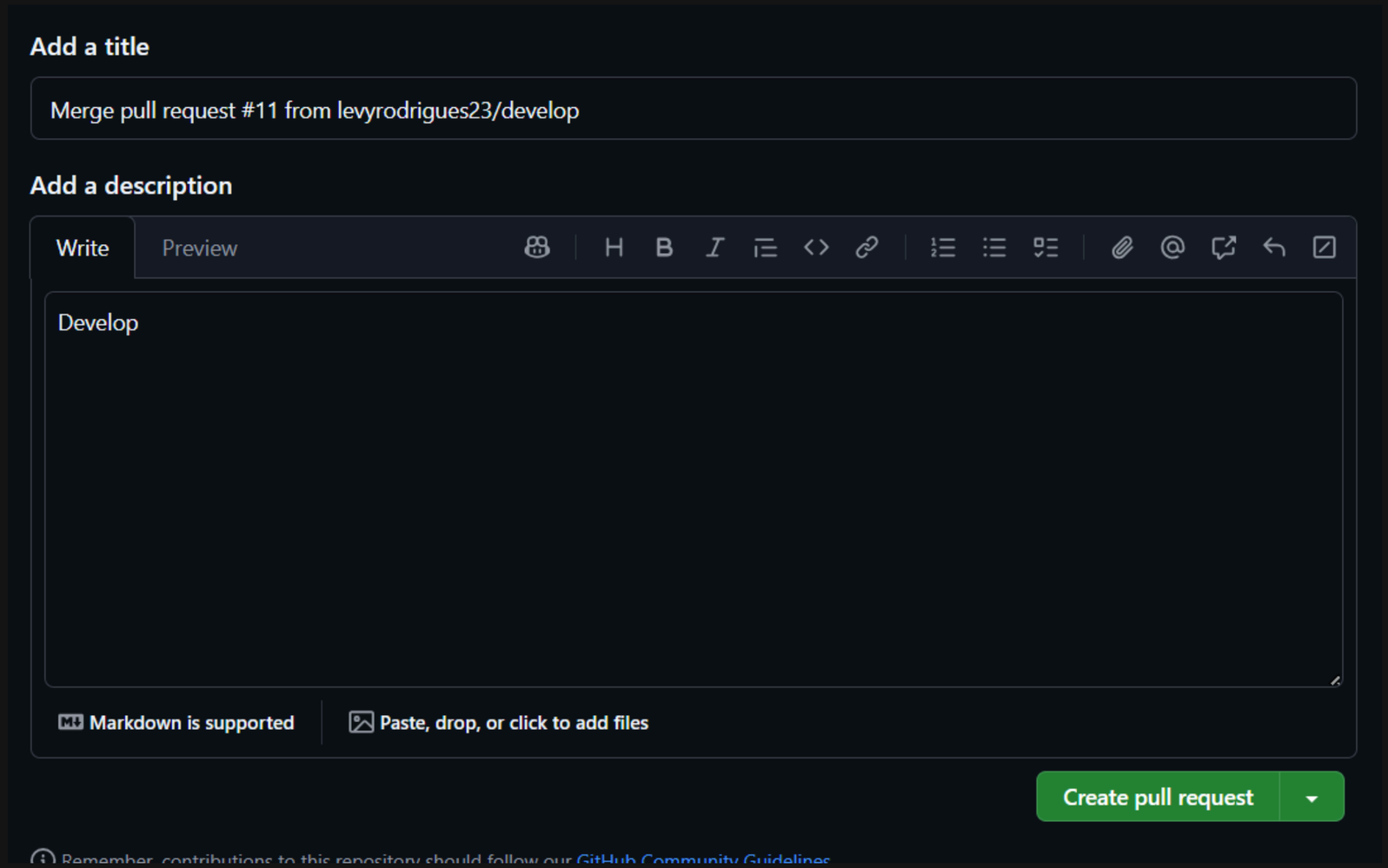
Acessando os pull requests



OBS: Para criar um pull request, voce precisa de outras branches para poder fazer as comparações

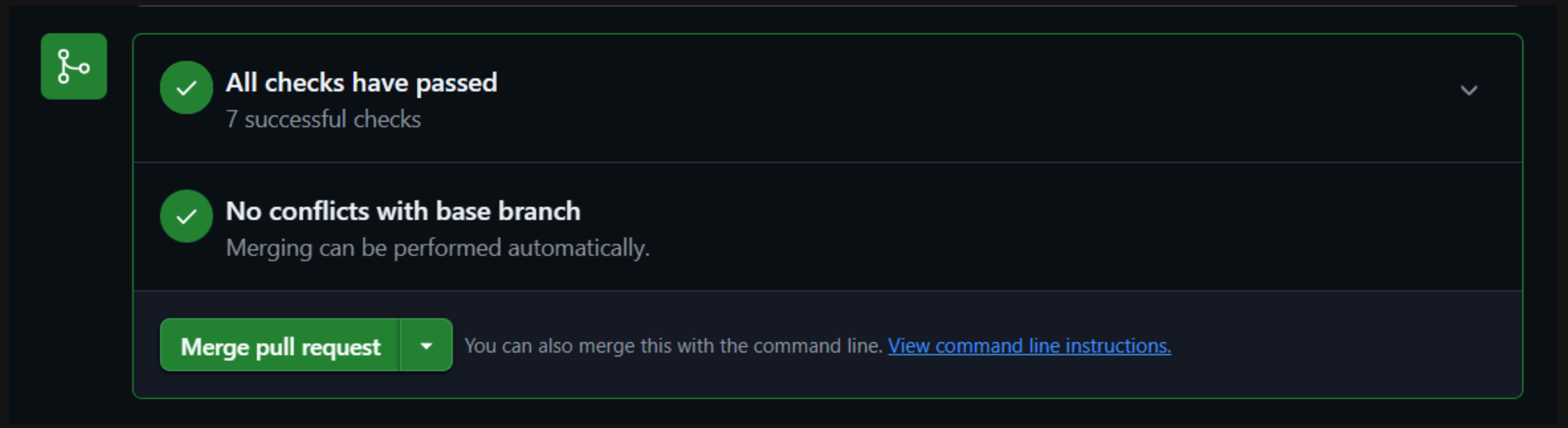
GITHUB NA PRÁTICA

Aqui voce pode criar um pull request antes de fazer o merge das alterações perante a comparação das branches.



GITHUB NA PRÁTICA

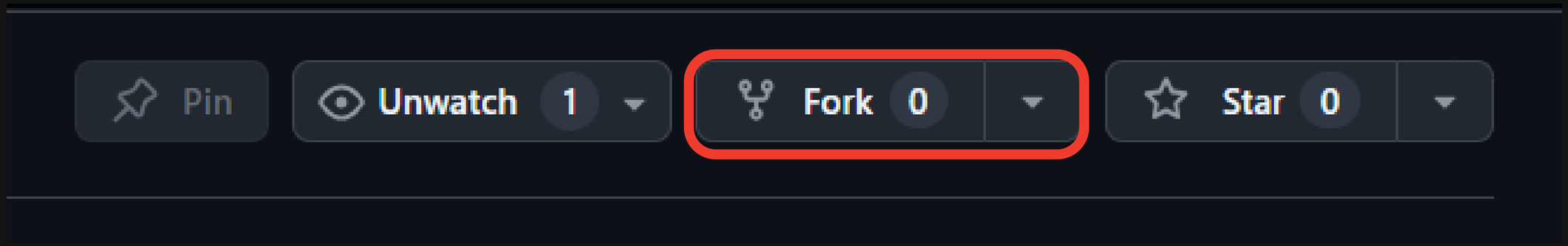
Após o processo anterior, é possível realizar o merge das alterações :)



GITHUB NA PRÁTICA

- **Fork**

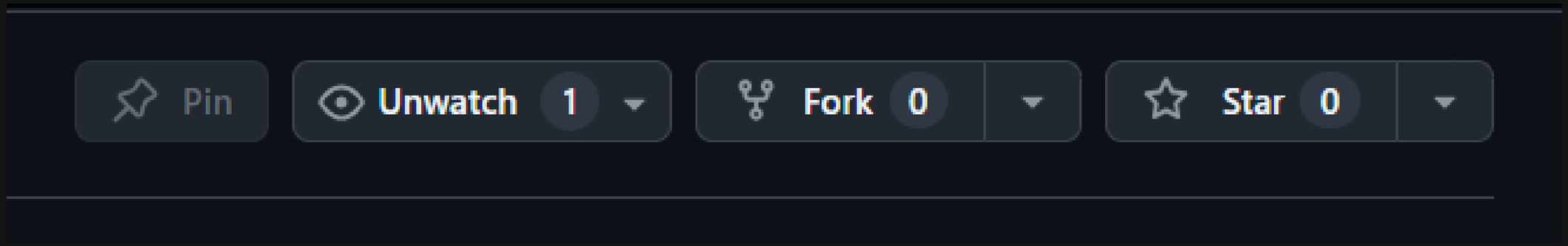
Um fork é essencialmente uma cópia de um repositório de código que permite desenvolvimento independente. Quando você faz um fork, está criando sua própria versão do projeto onde pode fazer modificações sem afetar o código original.



GITHUB NA PRÁTICA

- **Fork**

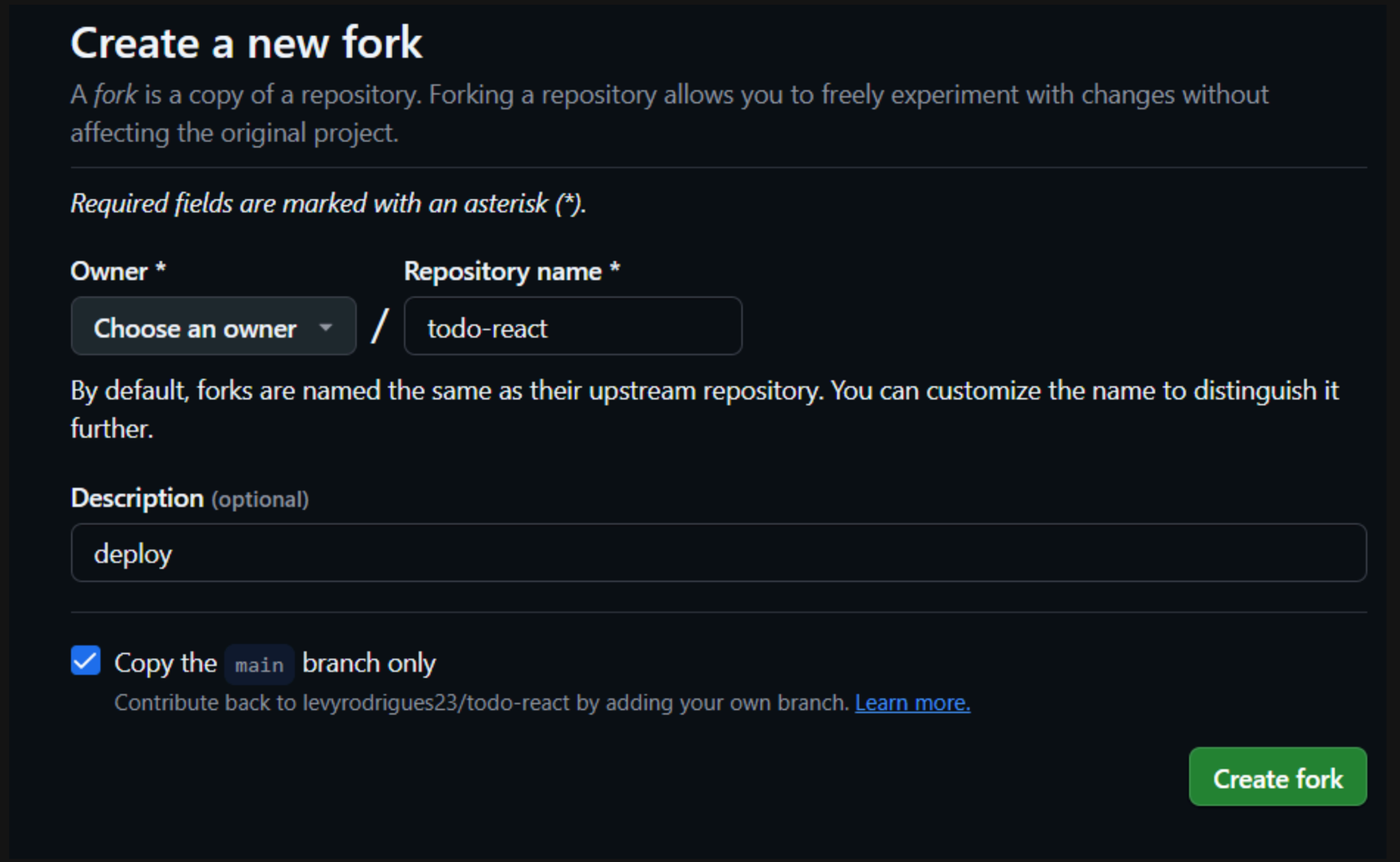
Um fork é essencialmente uma cópia de um repositório de código que permite desenvolvimento independente. Quando você faz um fork, está criando sua própria versão do projeto onde pode fazer modificações sem afetar o código original.



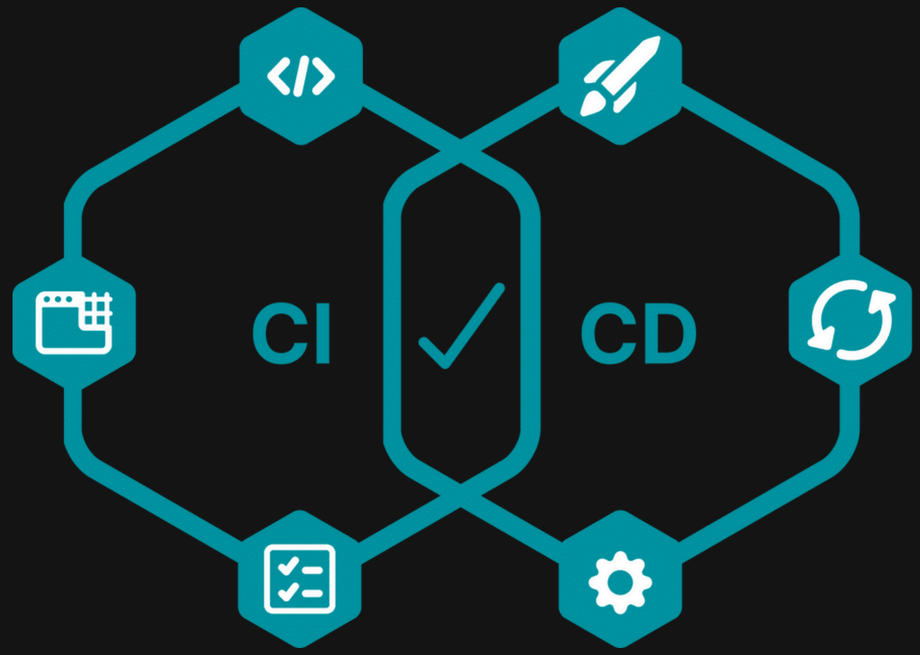
GITHUB NA PRÁTICA

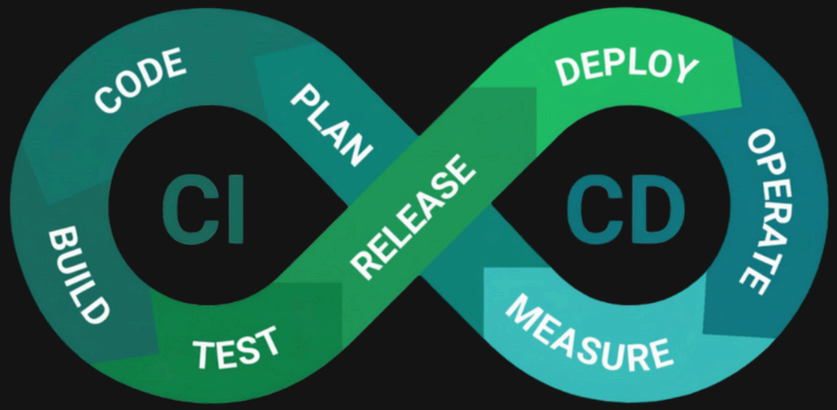
Cria uma cópia do repositório na tua conta do GitHub, que tu pode modificar à vontade e depois propor mudanças pro original via pull request.

GIT CLONE ≠ FORK



INTEGRAÇÃO CONTÍNUA (CI)
&
ENTREGA CONTÍNUA (CD)





O que é CI/CD?

CI/CD é uma prática essencial no desenvolvimento de software moderno que visa automatizar as etapas durante o desenvolvimento de software. Dessa forma, as equipes de desenvolvimento conseguem entregar software de forma mais rápida, segura e confiável.

OBJETIVOS E VANTAGENS

CI

- Detectar problemas mais cedo
- Reduzir conflitos de merge
- Melhorar a qualidade do código

CD

- Lançamentos de baixo risco
- Entregas mais rápidas
- Feedback rápido

INTEGRAÇÃO CONTÍNUA (CI)

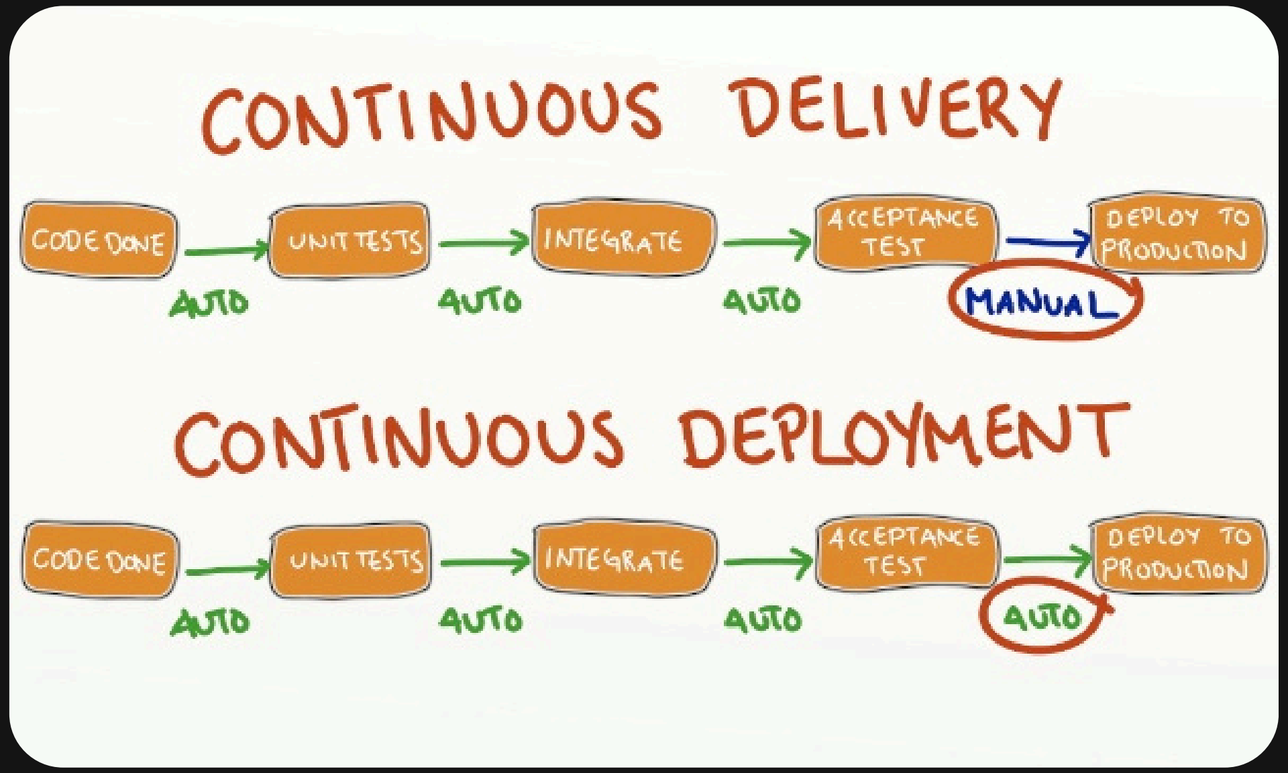
A Integração Contínua é a parte do processo focada nos desenvolvedores e na qualidade do código. A ideia principal é integrar o código de múltiplos desenvolvedores em um repositório central. Ele foca em garantir que o novo código não "quebre" o que já existe e está funcionando.

ENTREGA CONTÍNUA (CD)

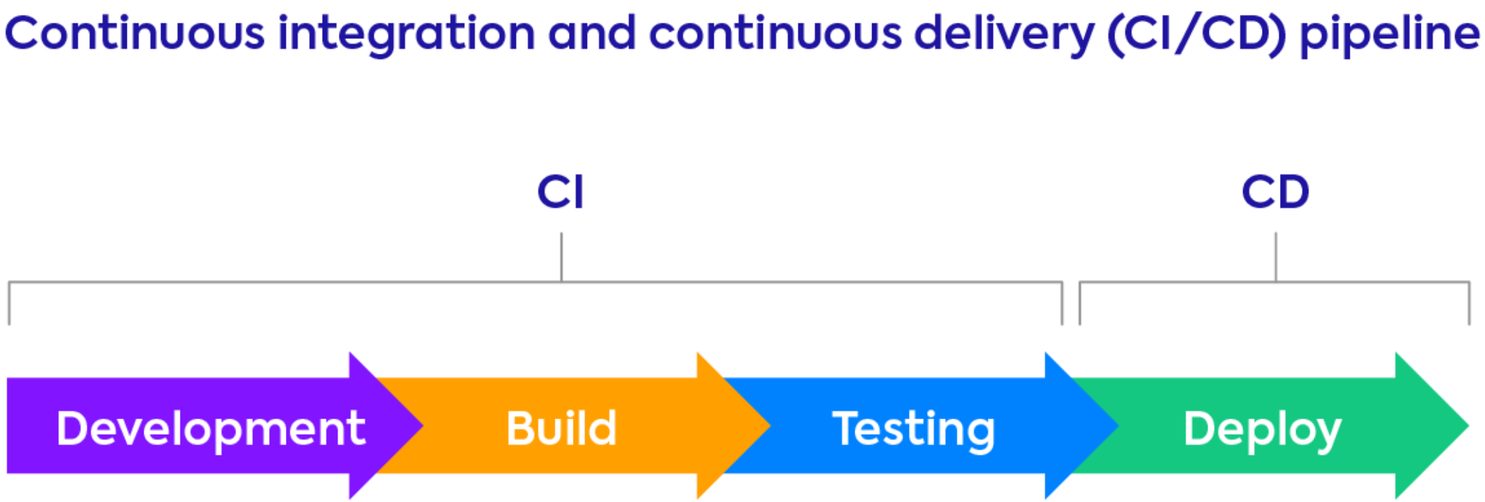
Automatiza o processo de lançamento do software para os ambientes de teste e produção. Isso permite que a equipe de negócios ou de qualidade decida o melhor momento para liberar a nova versão para os usuários finais. o CD ocorrem em 2 partes: Entrega Contínua e Implantação Contínua.
O CD ocorre após o CI, caso tenha passado nas etapas de (build e teste).

- **ENTREGA CONTÍNUA: O PROCESSO É AUTOMATIZADO ATÉ A PRODUÇÃO. O LANÇAMENTO FINAL É MANUAL.**
- **IMPLANTAÇÃO CONTÍNUA: SE O CÓDIGO PASSAR POR TODOS OS TESTES AUTOMATIZADOS, ELE É LANÇADO EM PRODUÇÃO AUTOMATICAMENTE, SEM QUALQUER INTERVENÇÃO HUMANA.**

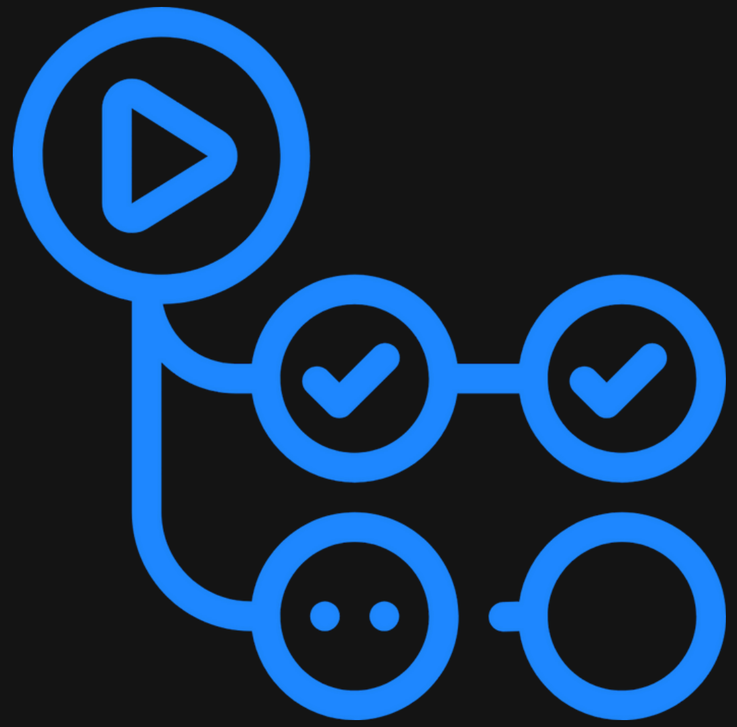
ENTREGA CONTÍNUA (CD)



FLUXO DE CI/CD



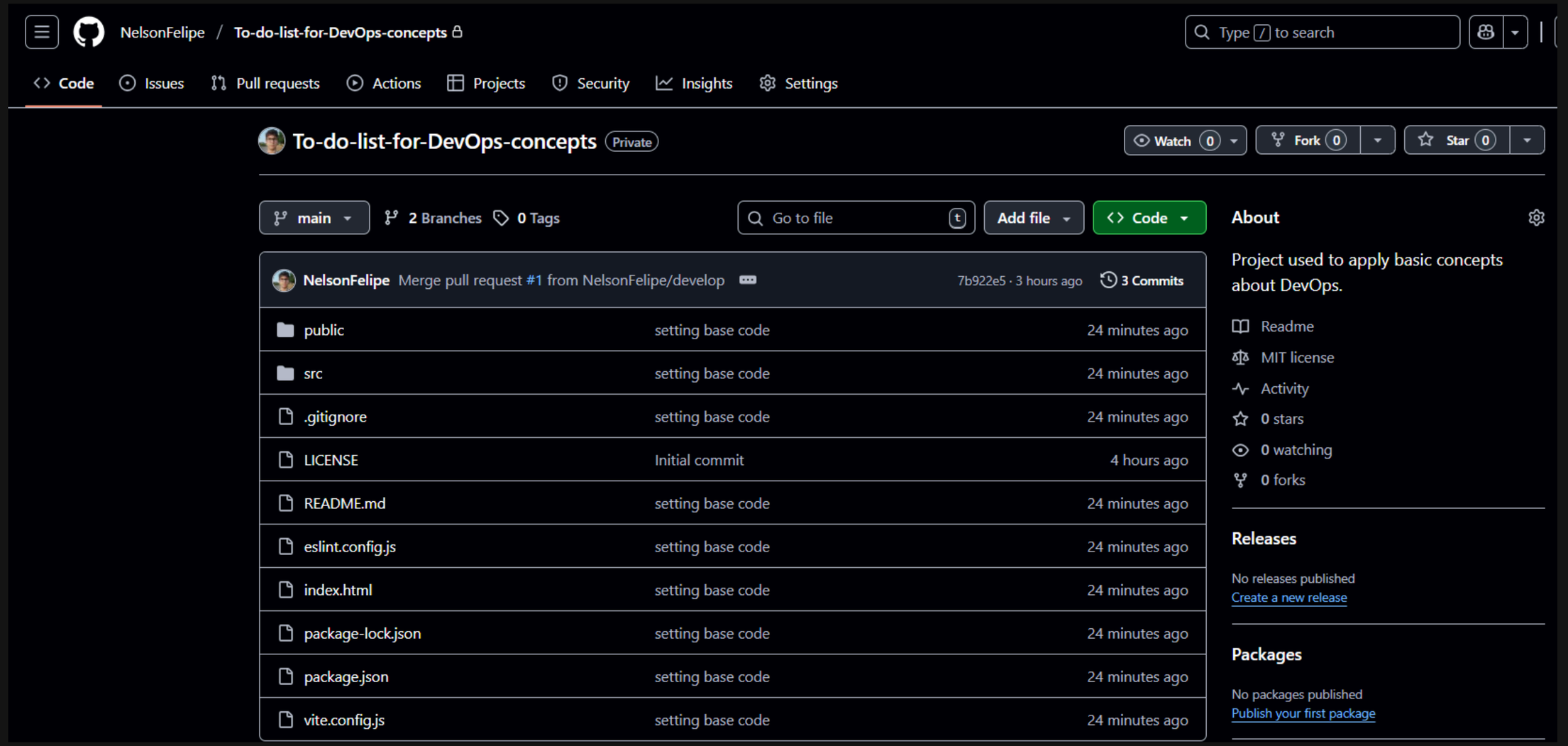
CI/CD COM
GITHUB ACTIONS



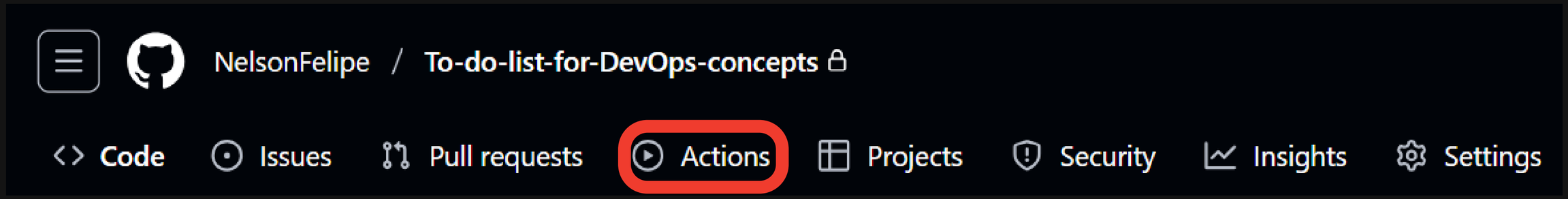
GITHUB ACTIONS

GitHub Actions é uma plataforma de automação integrada ao GitHub, com ela podemos automatizar uma vasta gama de tarefas e outras operações diretamente dos repositórios. Em essência, o GitHub Actions funciona como uma ferramenta de Integração Contínua e Entrega Contínua (CI/CD).

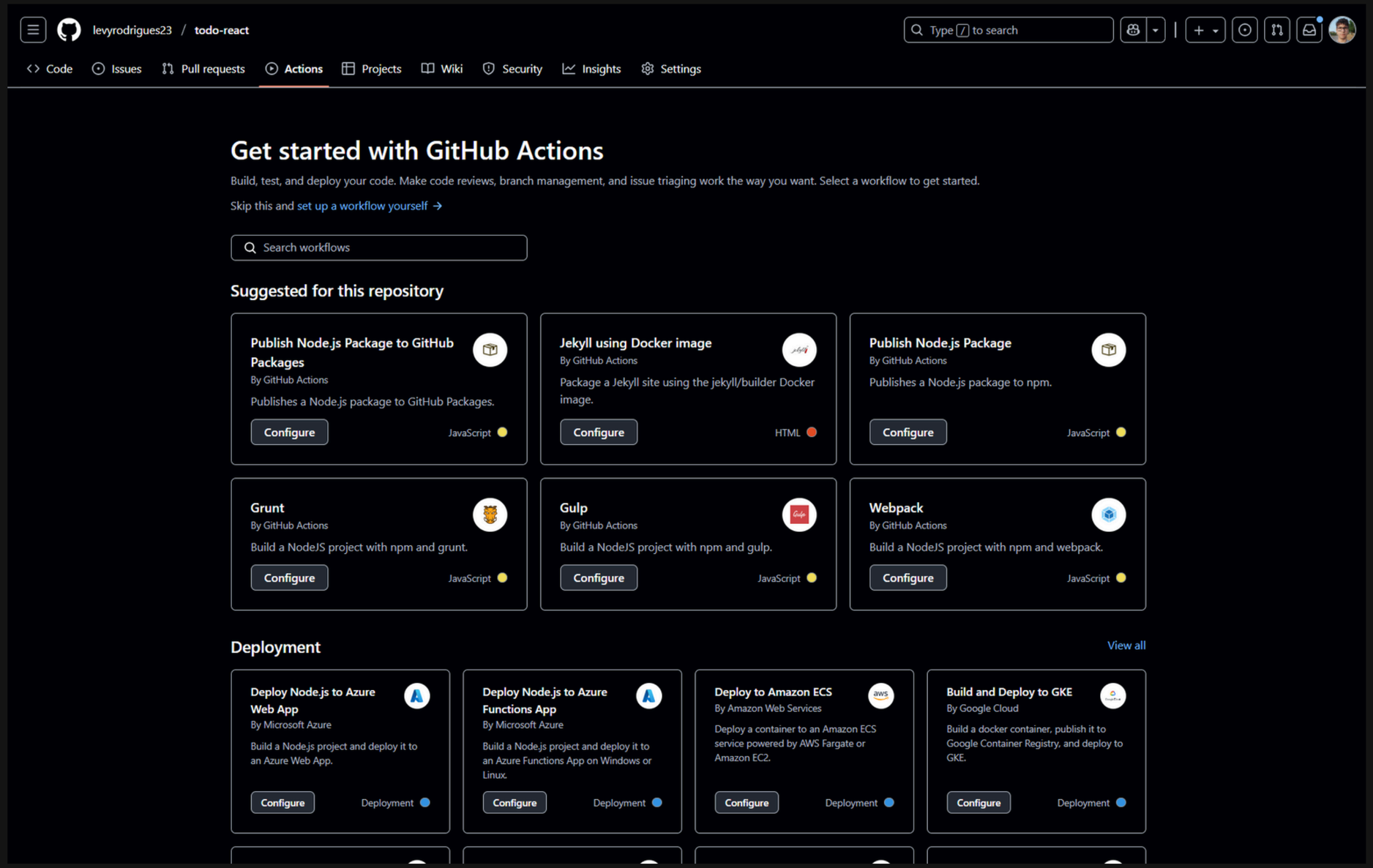
GITHUB ACTIONS



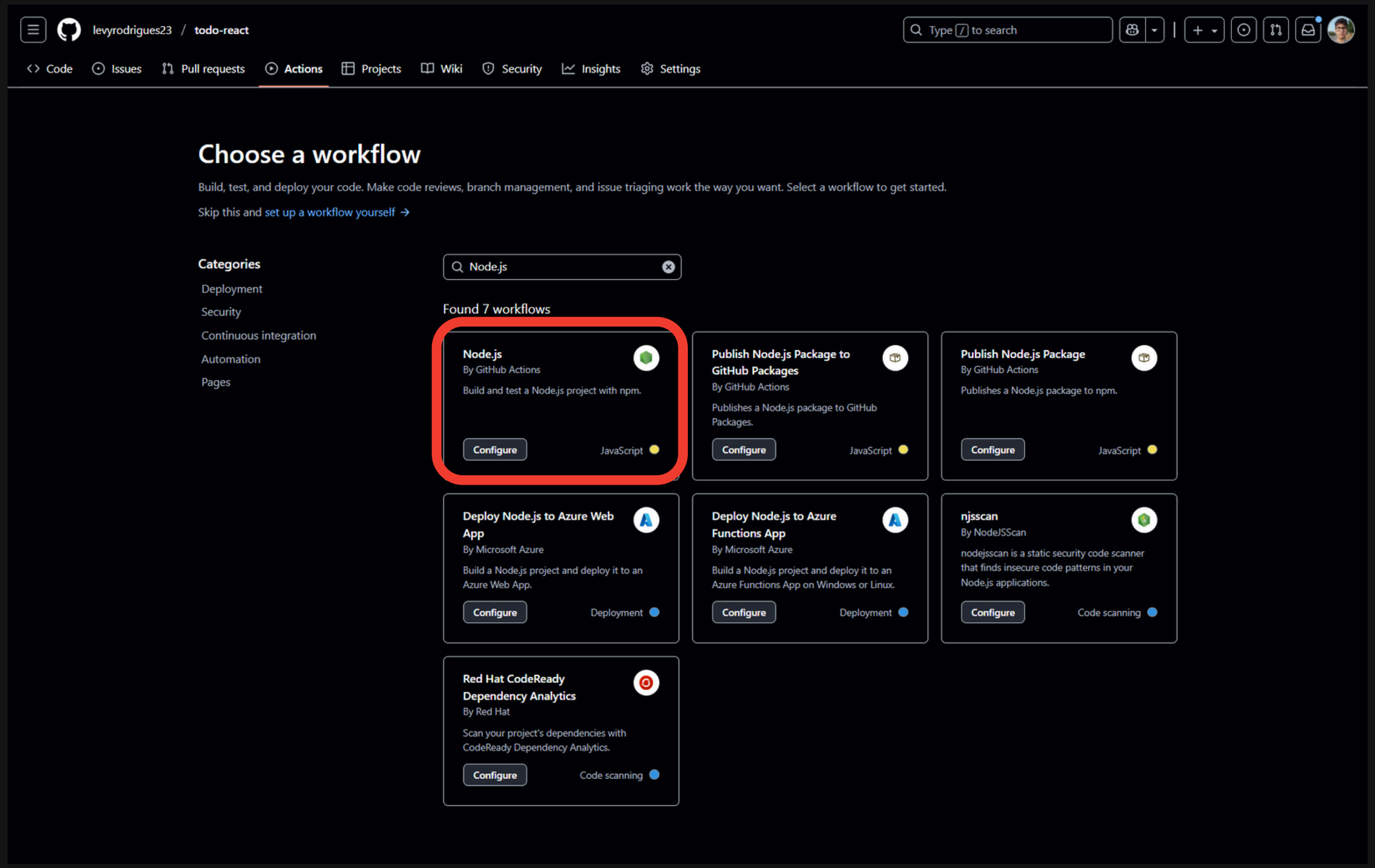
GITHUB ACTIONS



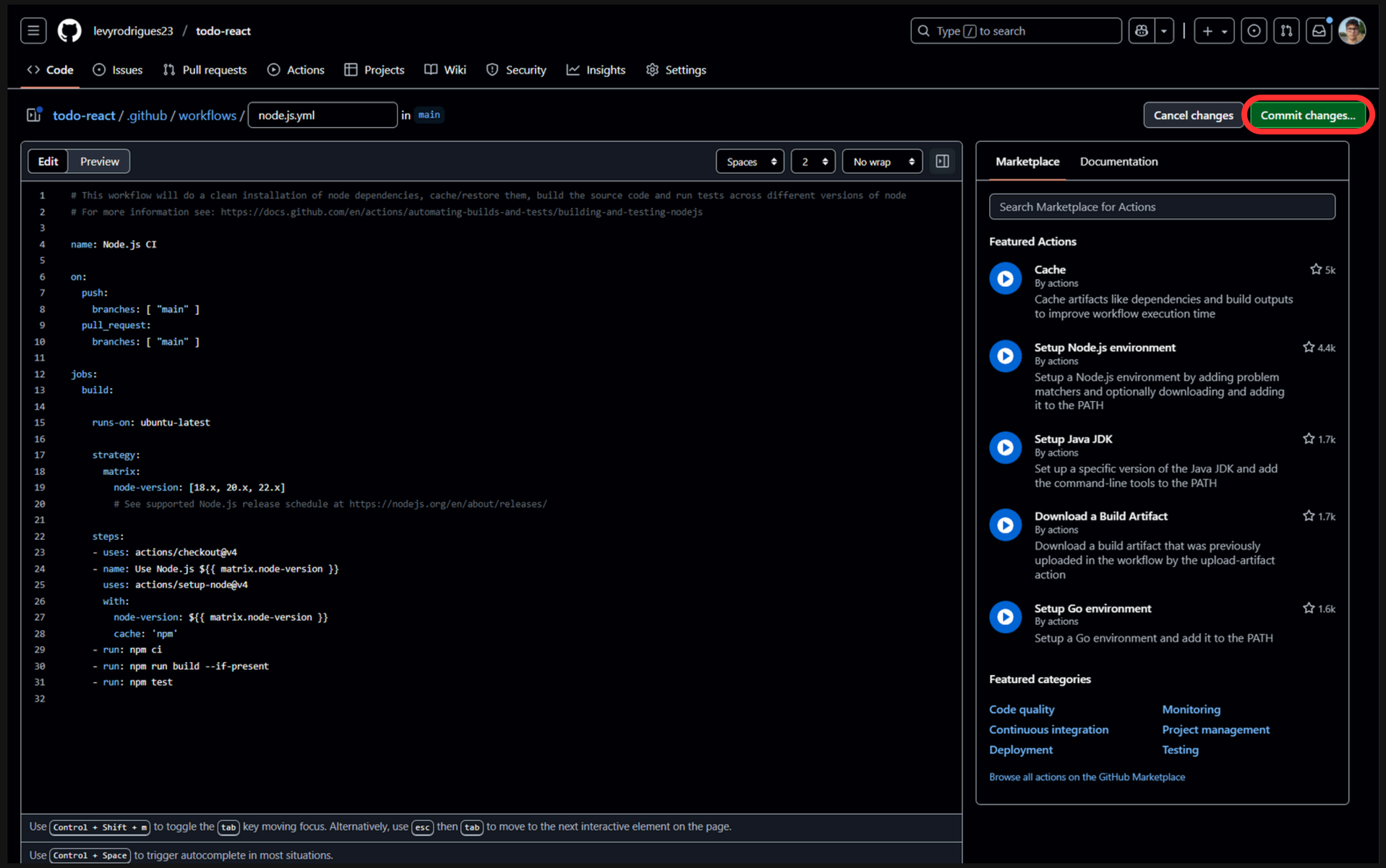
GITHUB ACTIONS



GITHUB ACTIONS



GITHUB ACTIONS



GITHUB ACTIONS

ON: DEFINE OS GATILHOS QUE ACIONAM OS WORFLOWS

JOBS: CONTÉM AS "TAREFAS" QUE SERÃO REALIZADAS PELO WORKFLOW.

RUNS-ON: DEFINE O AMBIENTE VIRTUAL (VM) ONDE OS PASSOS SERÃO EXECUTADOS.

STRATEGY: CONTROLA COMO O JOB SERÁ EXECUTADO EM DIFERENTES VARIAÇÕES, GERALMENTE COM UMA MATRIZ (MATRIX).

STEPS: DEFINEM AS INSTRUÇÕES INDIVIDUAIS DENTRO DE UM JOB. SÃO EXECUTADOS EM ORDEM.

USES: DEFINEM O USO DE UMA ACTION

NAME: DEFINE UM RÓTULO

WITH: DEFINE INPUTS PARA AS ACTIONS.

CACHE: ACELERA EXECUÇÕES FUTURAS.

RUN: EXECUTAR COMANDOS DE SHELL.

todo-react / .github / workflows /

```yaml
name: Node.js CI

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

jobs:
  build:

    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [18.x, 20.x, 22.x]
        # See supported Node.js release schedule at https://nodejs.org/en/about/releases/

    steps:
    - uses: actions/checkout@v4
    - name: Use Node.js ${{ matrix.node-version }}
      uses: actions/setup-node@v4
      with:
        node-version: ${{ matrix.node-version }}
        cache: 'npm'
    - run: npm ci
    - run: npm run build --if-present
    - run: npm test

```

GITHUB ACTIONS

```yaml
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4
      - name: Setup Node.js
        uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'
      - name: Install dependencies
        run: npm ci
      - name: Run ESLint
        run: npm run lint
```

GITHUB ACTIONS

```yaml
  test:
    runs-on: ubuntu-latest
    strategy:
      matrix:
        node-version: [18.x, 20.x, 22.x]
    steps:
      - uses: actions/checkout@v4
      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v4
        with:
          node-version: ${{ matrix.node-version }}
          cache: 'npm'
      - run: npm ci
      - run: npm test
```

GITHUB ACTIONS

```yaml
  build:
    runs-on: ubuntu-latest
    needs: test
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-node@v4
        with:
          node-version: '20'
          cache: 'npm'
      - run: npm ci
      - run: npm run build --if-present
      - name: Upload artifact
        uses: actions/upload-artifact@v4
        with:
          name: production-files
          path: ./dist
```

GITHUB ACTIONS

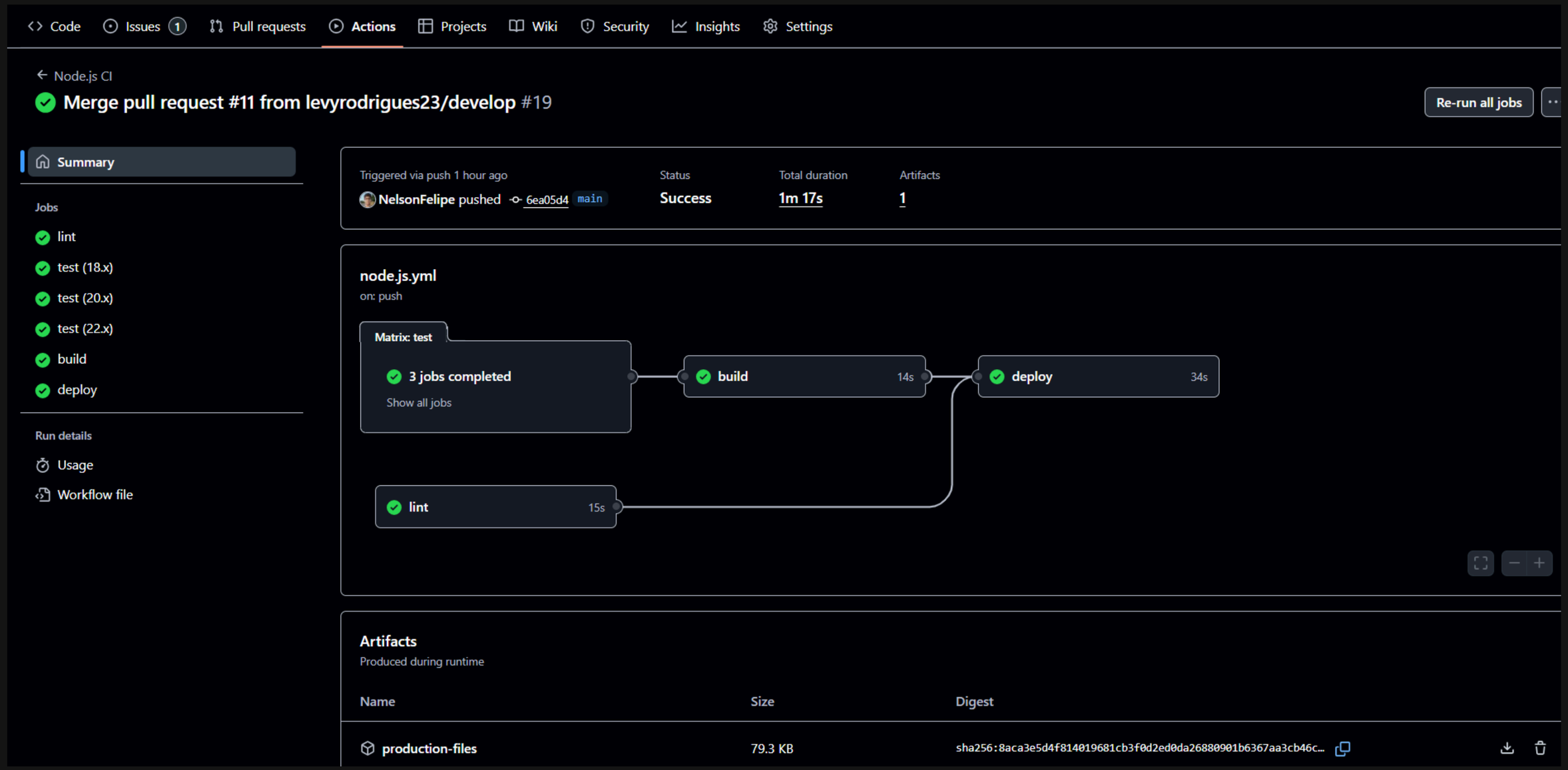
NEEDS: [BUILD, LINT, TEST]

IF: GITHUB.REF == `REFS/HEADS/MAIN`

FIREBASE_TOKEN: ${{ SECRETS.FIREBASE_TOKEN }}

```yaml
deploy:
  runs-on: ubuntu-latest
  needs: [build, lint, test]
  if: github.ref == 'refs/heads/main'
  steps:
    - name: Checkout repository
      uses: actions/checkout@v4
    - name: Setup Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '20'
        cache: 'npm'
    - name: Install dependencies and Build
      run: |
        npm ci
        npm run build --if-present
    - name: Deploy to Firebase Hosting
      uses: w9jds/firebase-action@master
      with:
        args: deploy --only hosting
      env:
        FIREBASE_TOKEN: ${{ secrets.FIREBASE_TOKEN }}
```

GITHUB ACTIONS



GITHUB ACTIONS

todo-react Public

Watch

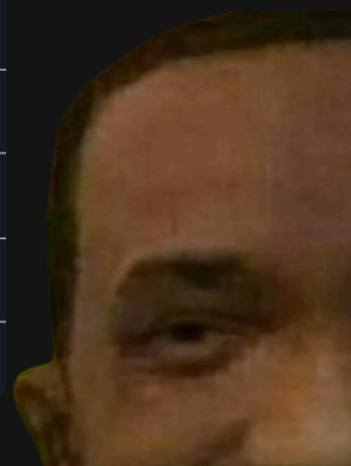
main 8 Branches 0 Tags Go to file d file Code

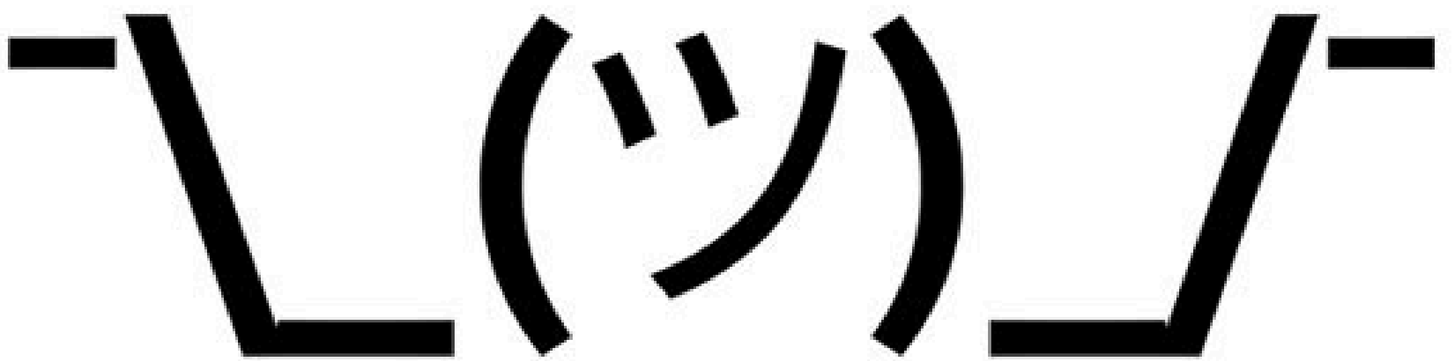
NelsonFelipe Merge pull request #11 from levyrodrigues23/develop 05d4 · 1 hour ago 24 Commits

All checks have passed

7 successful checks

- Node.js CI / lint (push) Successful in 15s Details
- Node.js CI / test (18.x) (push) Successful in 19s Details
- Node.js CI / test (20.x) (push) Successful in 12s Details
- Node.js CI / test (22.x) (push) Successful in 12s Details
- Node.js CI / build (push) Successful in 14s Details
- Node.js CI / deploy (push) Successful in 34s Details
- CodeRabbit - Review skipped Details





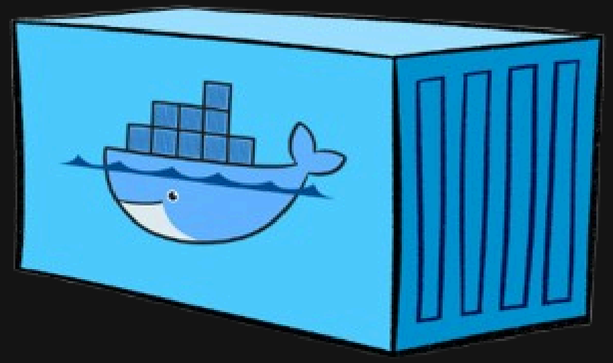
NA MINHA MÁQUINA

FUNCIONA!

INTRODUZINDO DOCKER

O Docker foi criado em 2013 por Solomon Hykes, junto com a equipe da empresa dotCloud (que depois mudou o nome para Docker Inc.).

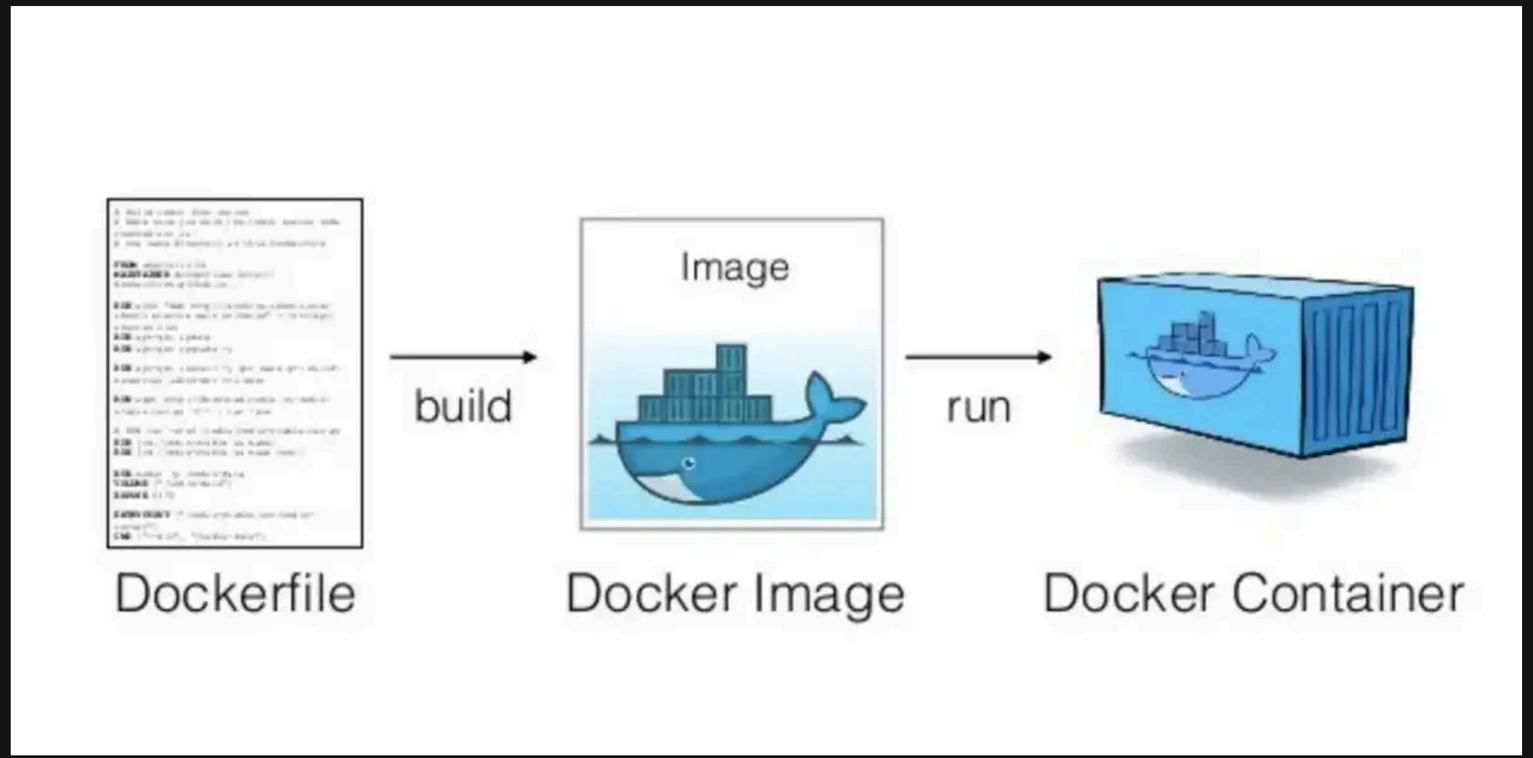
Uma ferramenta open source de virtualização leve



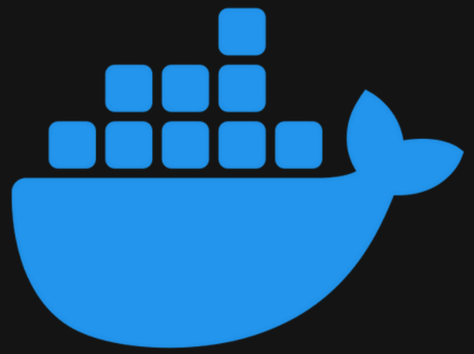
Roda aplicações isoladas em containers

PORQUE USAR DOCKER?

- Pacote completo → junta código, bibliotecas e configurações num só lugar.
- Segurança e organização → variáveis e senhas guardadas direitinho.
- Funciona em qualquer lugar → mesma versão em qualquer computador ou servidor.

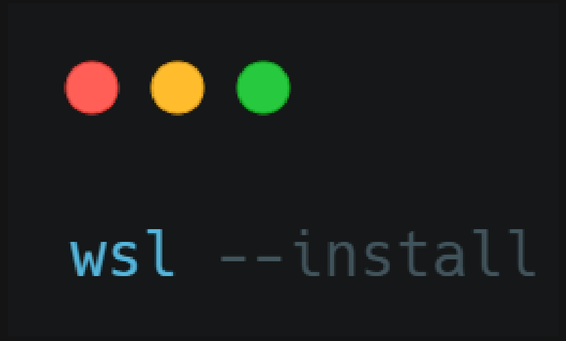


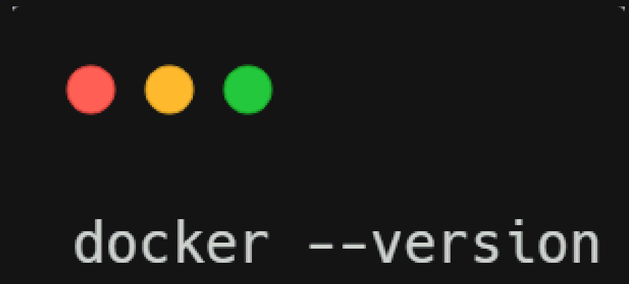
INSTALANDO DOCKER



Windows / macOS

- Baixar e instalar o Docker Desktop:
- https://www.docker.com/products/docker-desktop
- Seguir os passos do instalador.
- Reiniciar a máquina (se pedir).
- Verificar no terminal:

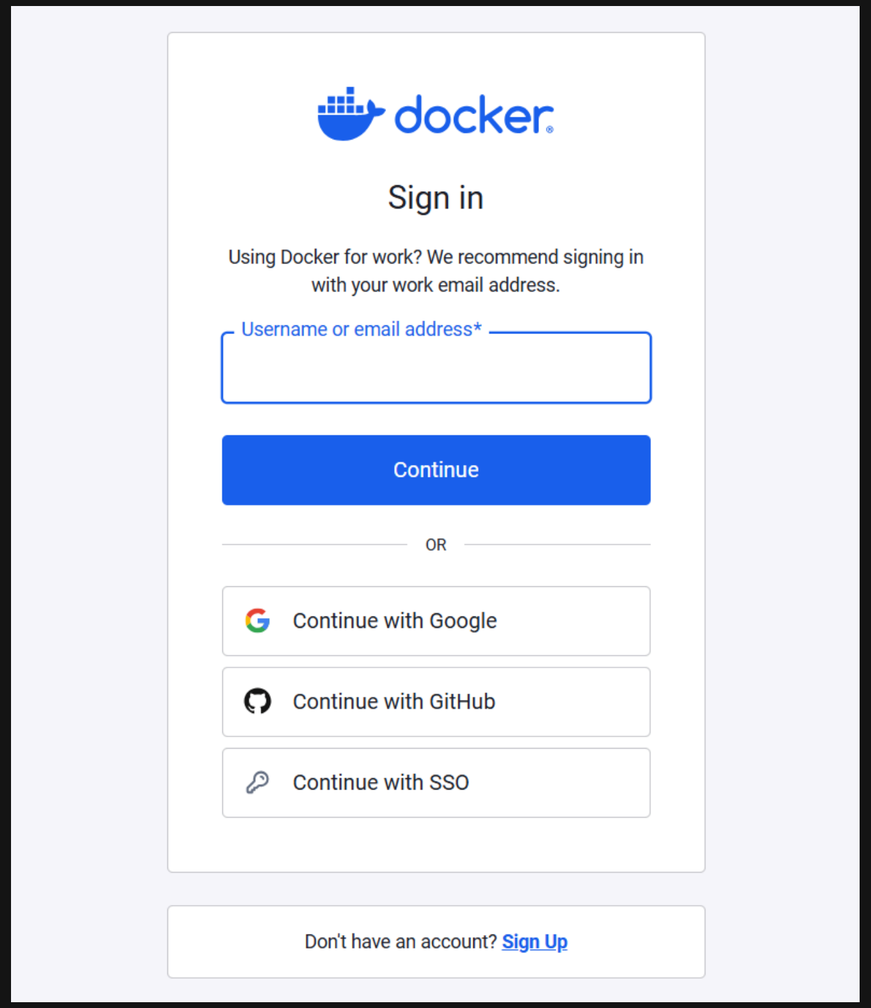




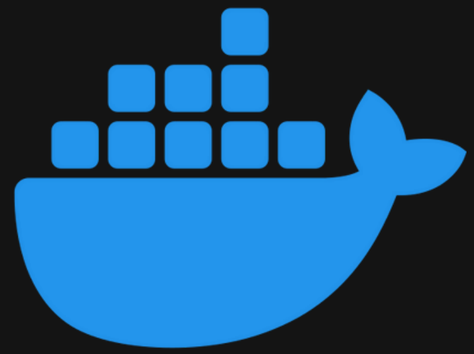
No Windows, o Docker Desktop usa o WSL2 (Subsistema Linux).
Após instalar, já é possível rodar containers, criar imagens e usar o docker-compose.

INSTALANDO DOCKER

1. Clique em Sign Up
2. Escolha Continue with GitHub
3. Autorize o acesso e pronto!



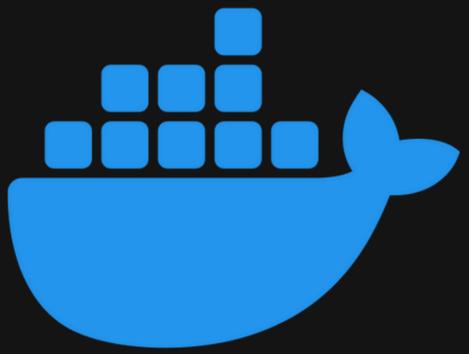
CONCEITOS BÁSICOS



Container: execução isolada do app.
Imagem: definição de como o container vai funcionar.
Porta: comunicação do container com o mundo externo.
Volume: pasta compartilhada, persiste dados mesmo que o container seja destruído.
Registro / Registry: armazena imagens (Docker Hub)

Detalhamento de cada conceito a seguir ⟶

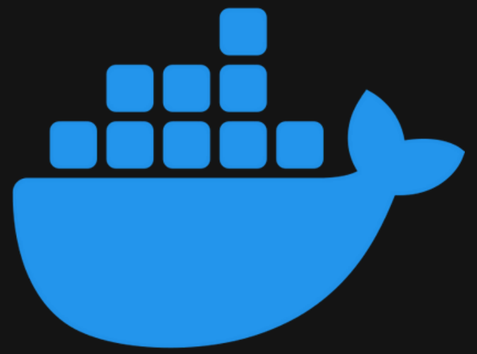
CONTAINER



É como se fosse uma caixinha onde roda a aplicação. Dentro dele vai estar tudo que a aplicação precisa (bibliotecas, dependências, configs), separado do resto do sistema.

Exemplo: um container só do Postgres, outro só do Node.js.

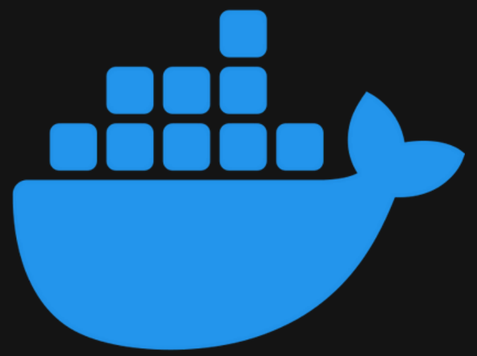
IMAGEM



É o “molde” usado pra criar containers. Pensa como uma receita: a imagem diz como o container deve ser montado.

Exemplo: node:18-alpine é uma imagem oficial do Node mais leve.

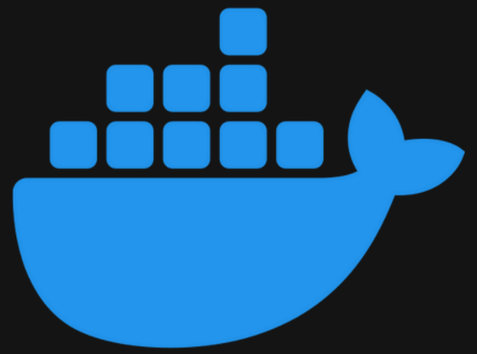
PORTA



É a forma de expor o container pro mundo externo. Como ele é isolado, precisa de uma “porta de entrada/saída” pra comunicação.

Exemplo: **-p 3000:3000** mapeia a porta do container pra máquina.

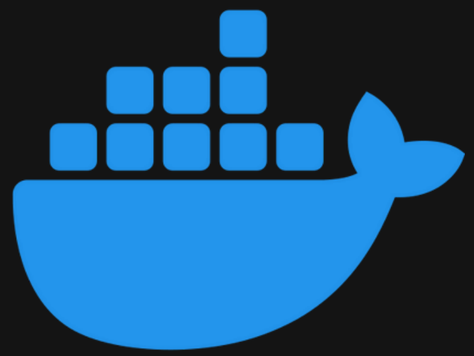
VOLUME



Área de armazenamento compartilhada. Mesmo que o container morra, os dados ficam salvos.

Exemplo: banco de dados em Docker usa volume pra não perder as tabelas ao reiniciar.

REGISTRO / REGISTRY



É onde ficam guardadas as imagens (tipo um repositório).

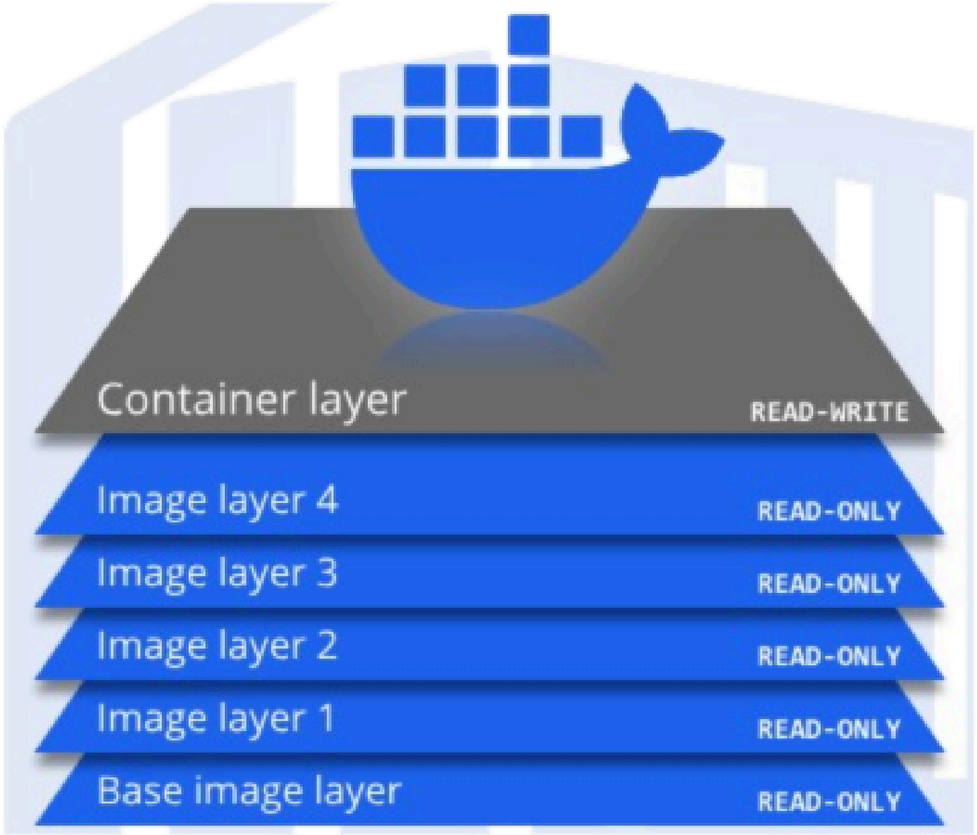
Exemplo: Docker Hub (público) ou um registry privado da empresa

DIFERENÇA DE IMAGEM E CONTAINER

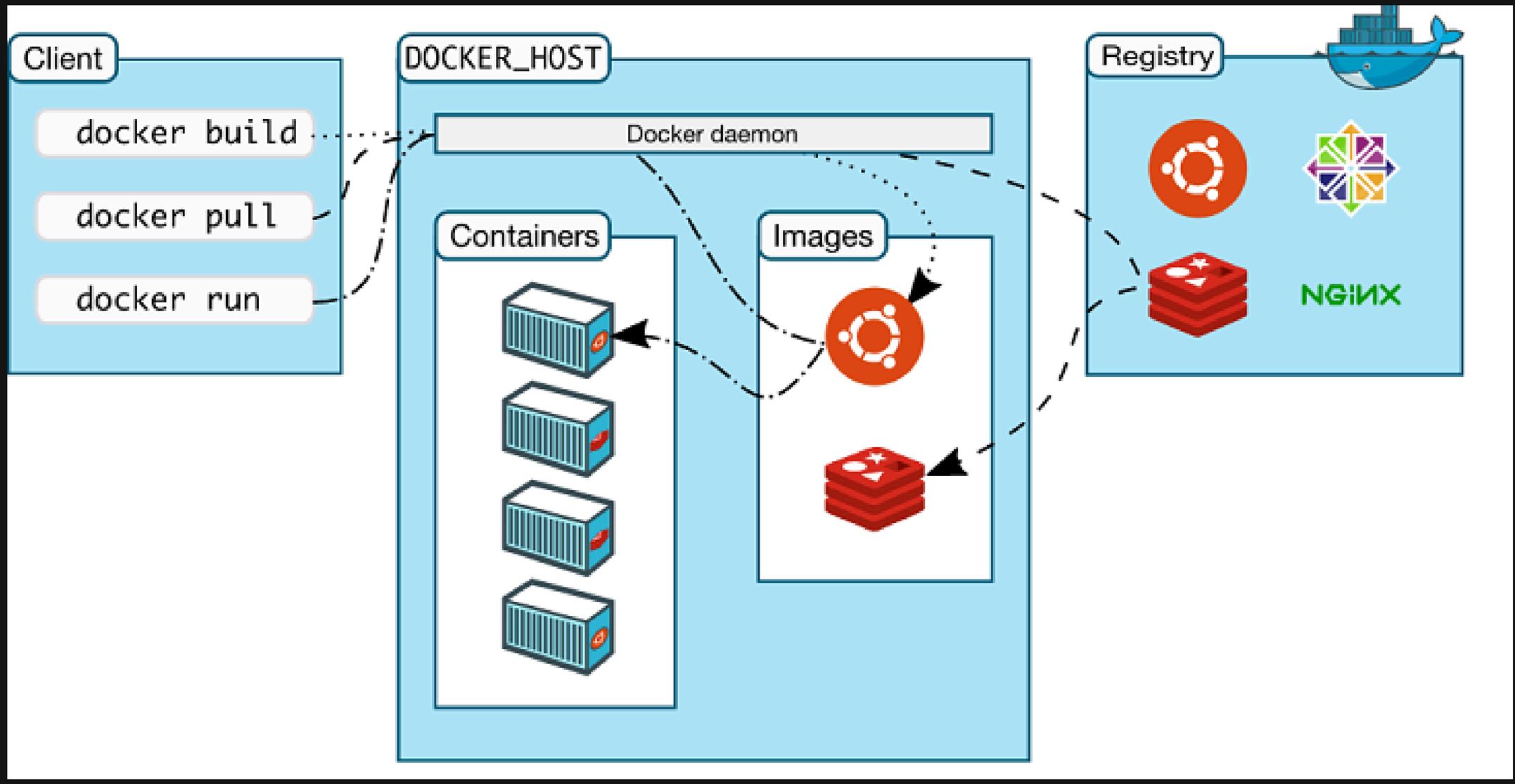
Imagem → camadas read-only com código + dependências

Container → adiciona camada read-write sobre a imagem

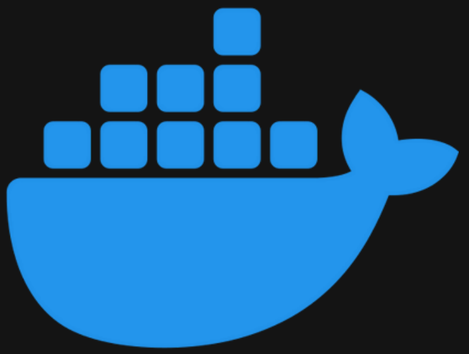
Vários containers podem ser criados da mesma imagem



ARQUITETURA DO DOCKER



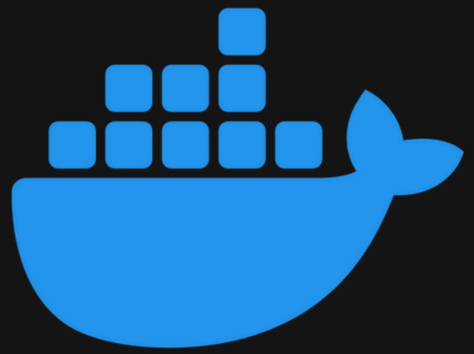
COMANDOS BÁSICOS



Imagens

```
docker images                    # listar todas as imagens
docker image ls                  # mesmo comando acima
docker pull nginx                # baixar imagem do registry
docker build .                   # criar imagem do Dockerfile atual
```

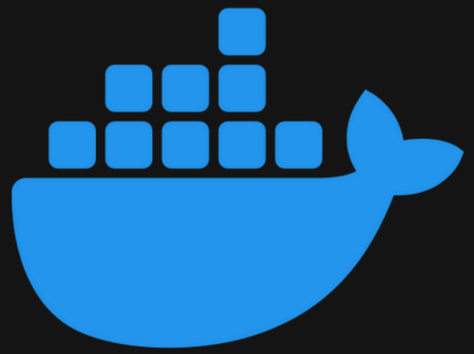
COMANDOS BÁSICOS



Executando containers

```
docker run nginx                    # executar container básico
docker run -d nginx                 # executar em background (detached)
docker run -d -p 8080:80 nginx      # mapear porta host:container
docker run -d -P nginx              # mapear porta aleatória
```

COMANDOS BÁSICOS



http://localhost:8080

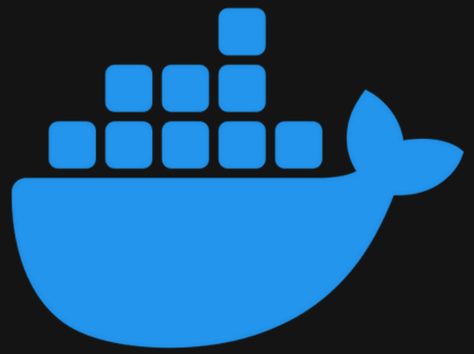
Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

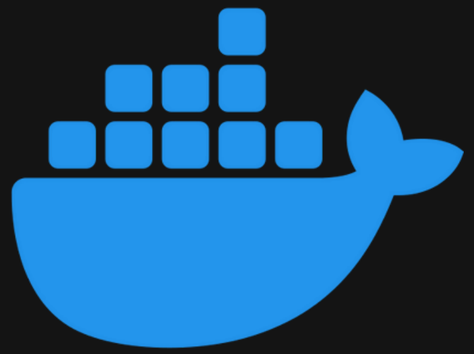
Thank you for using nginx.

COMANDOS BÁSICOS



```
# Executa um comando dentro do container
docker exec -it <id_container> <comando>

# Pausa e volta o container
docker pause <id_container>
docker unpause <id_container>

# Remove um container parado
docker rm <id_container>

# Força a remoção de um container ativo
docker rm -f <id_container>

# Remove todos os containers parados
docker container prune
```

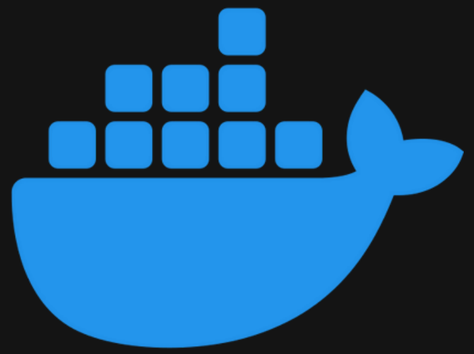
COMANDOS BÁSICOS



Monitorar Containers

```
docker ps                    # containers rodando
docker ps -a                 # todos os containers (rodando + parados)
docker inspect <id>          # informações detalhadas do container
docker stats <id>            # uso de CPU, memória, rede em tempo real
docker attach <id>           # conectar ao stdout do container
```

EXEMPLO PRÁTICO



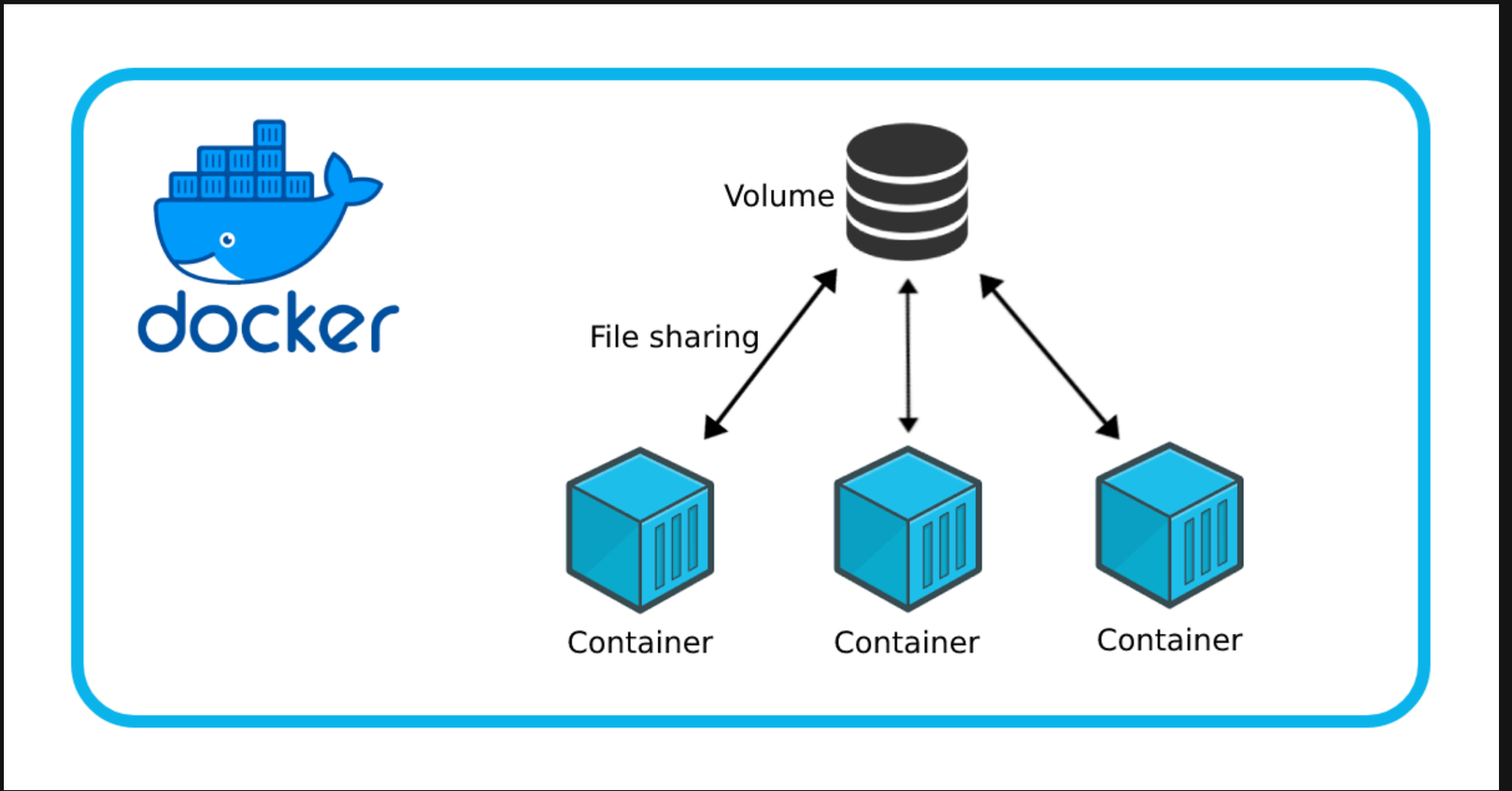
Consumindo a imagem do nginx

```
# 1. Executar nginx em background na porta 8080
docker run -d -p 8080:80 nginx

# 2. Testar se está funcionando
curl localhost:8080                    # retorna HTML do nginx

# 3. Acessar pelo navegador
# http://localhost:8080 ou http://SEU_IP:8080

# 4. Ver containers rodando
docker ps

# 5. Parar o container
docker stop <container_id>
```

VOLUMES NO DOCKER

O que são volumes no docker?

É um espaço de armazenamento **fora da camada** de escrita do container.
Dados **não se perdem** se o container for removido.

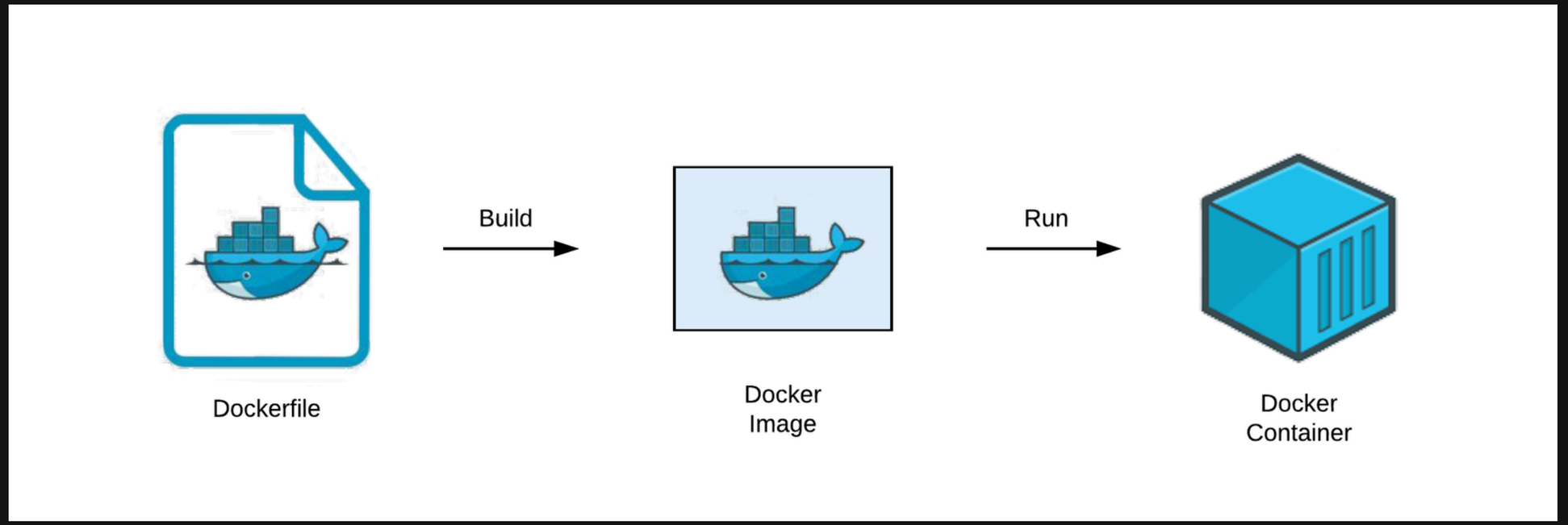


O QUE É UMA DOCKERFILE?

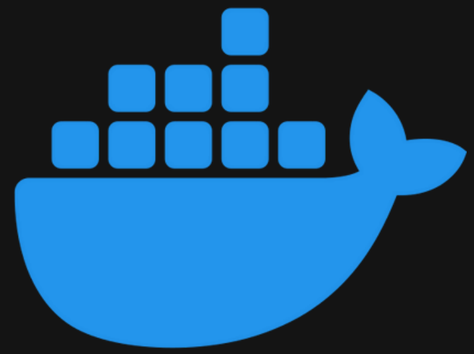
É uma sequência de comandos que vai ditar do que a minha aplicação precisa.

Essa sequência de passos é definida da seguinte forma:

- FROM
- RUN
- COPY
- ENV
- WORKDIR
- EXPOSE
- CMD



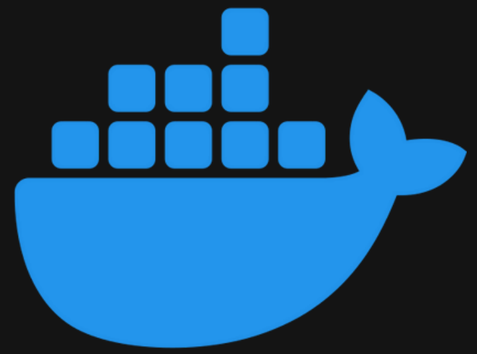
FROM



É o ponto de partida. Tu diz de qual imagem base vai herdar.

```
FROM node:18
```

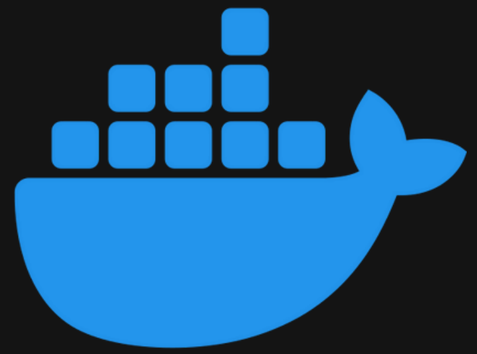
RUN



Executa comandos durante a construção da imagem.

```
RUN npm install
```

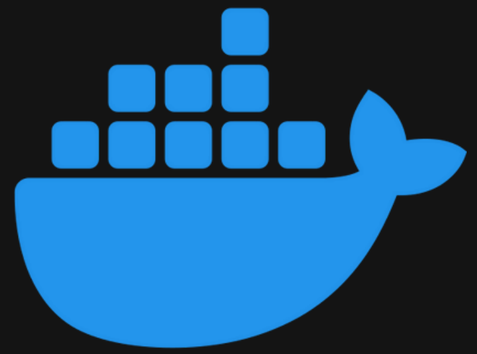
COPY



Copia arquivos do computador para dentro do container.

```
COPY . .
```

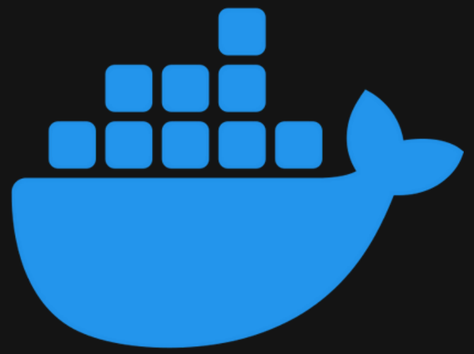
ENV



Define variáveis de ambiente dentro do container.

```
ENV NODE_ENV=production
```

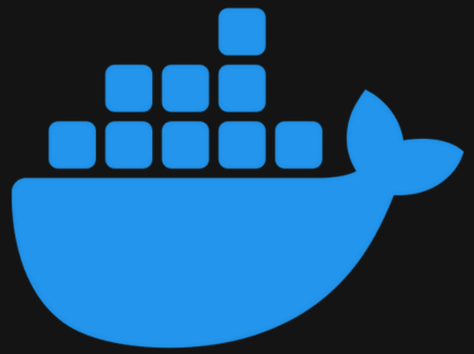
WORKDIR



Define qual pasta será o "diretório atual" dentro do container.

```
WORKDIR /app
```

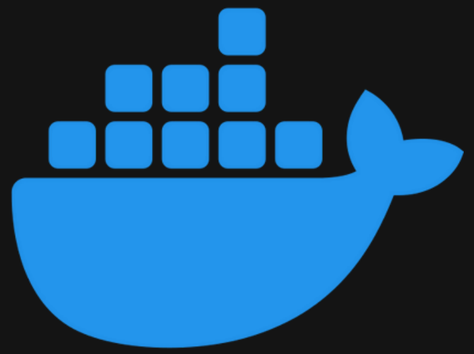
EXPOSE



Informa que a aplicação vai usar a porta a ser definida internamente.

```
EXPOSE 3000
```

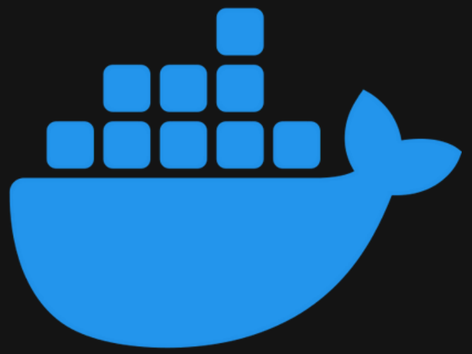
CMD



Comando final que roda quando o container é iniciado.

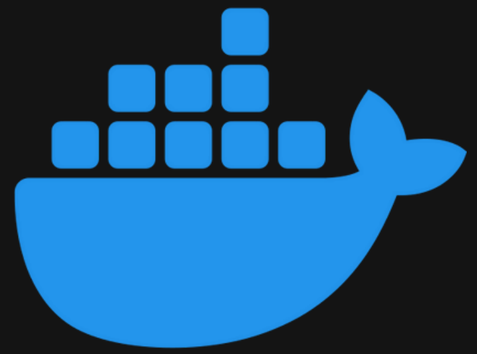
```
CMD ["npm", "start"]
```

CACHE EM DOCKER



- Cada comando que eu executo é uma nova camada.
- temos uma hierarquia entre as camadas, ou seja, a ordem importa.
- se eu modificar o diretório do projeto, o comando **COPY** copia todo o projeto novamente porque houve alteração.
- poderíamos adaptar o build apenas para o código principal, ou seja, fazendo um COPY apenas para o código em si.
- ao invés de cada COPY que for executado

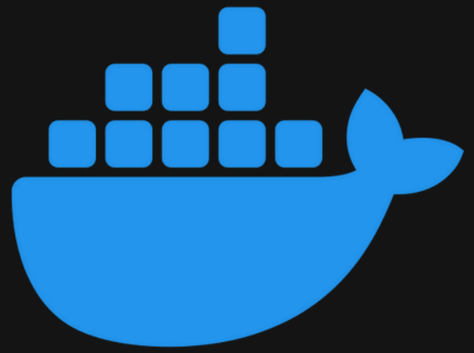
EXEMPLO DE UMA DOCKERFILE



Exemplo ao lado
mostra uma Dockerfile
para um app Node.js.

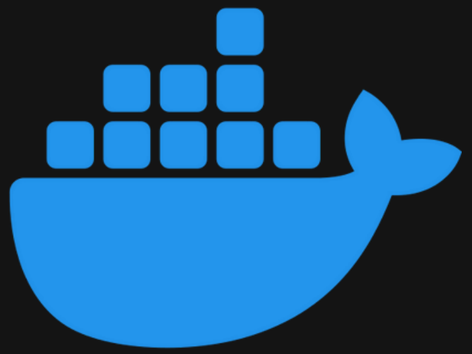
```
FROM node:18-alpine

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 3000
CMD ["npm", "start"]
```

RODANDO O CONTAINER



```
# 1️⃣ Construir a imagem e definir o nome
docker build -t meu-app .

# 2️⃣ Rodar um container a partir da imagem
docker run -d -p 3000:3000 meu-app
```

BOAS PRÁTICAS EM DOCKER



- Usar imagens leves (ex.: node:18-alpine).
- Minimizar camadas no Dockerfile.
- Usar .dockerignore (mesmo conceito do .gitignore).
- Sempre versionar o Dockerfile junto com o código.