



SÃO
PAULO
TECH
SCHOOL

Computação e Sistemas Distribuídos

Processo de CI e CD:

Mudança de Mentalidade e
Benefícios a Longo Prazo

Eduardo Verri

eduardo.verri@sptech.school

Continuous Integration (CI)

Continuous Integration [CI]

Continuous Integration (CI) ou Integração Contínua é uma prática de desenvolvimento de software onde as alterações de código são automaticamente testadas e mescladas em um repositório comum com frequência. Isso facilita a detecção precoce de problemas e incompatibilidades, promovendo uma entrega de software mais rápida e confiável.

Ao automatizar a compilação e os testes, o CI diminui significativamente o risco de erros humanos, aumentando assim a eficiência e a segurança no ciclo de vida do desenvolvimento de software.

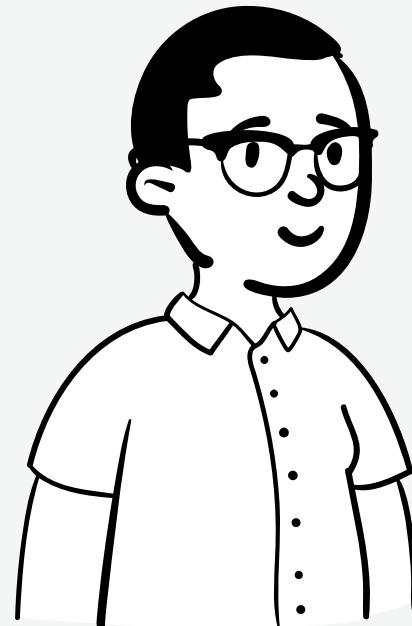


Principais processos

- Execução de Testes (unidade, integração e funcionalidade);
- Validação de padrões de código (Linter);
- Qualidade (SonarQube, Qodana e etc);
- Segurança (Senhas, tokens e etc);
- Geração de artefatos (imagem, código intermediário, binário executável);
- Identificação da próxima versão que será gerada no software;
- Tags e Releases;

Exemplo:

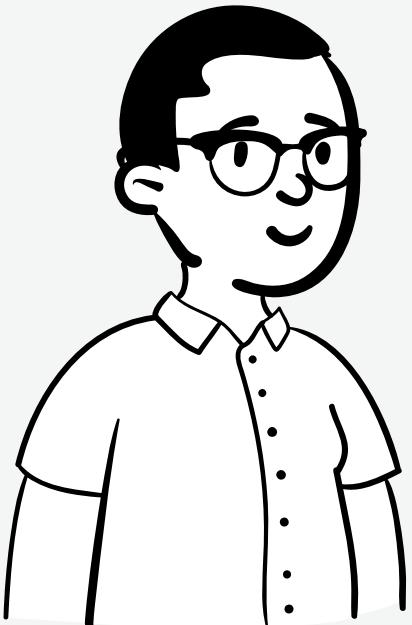
Imagine que Paulo está trabalhando em um novo recurso para o projeto de software de Manoel. Paulo finaliza sua parte do trabalho e cria um **Pull Request (PR)** para integrar suas mudanças no repositório principal.



Exemplo:

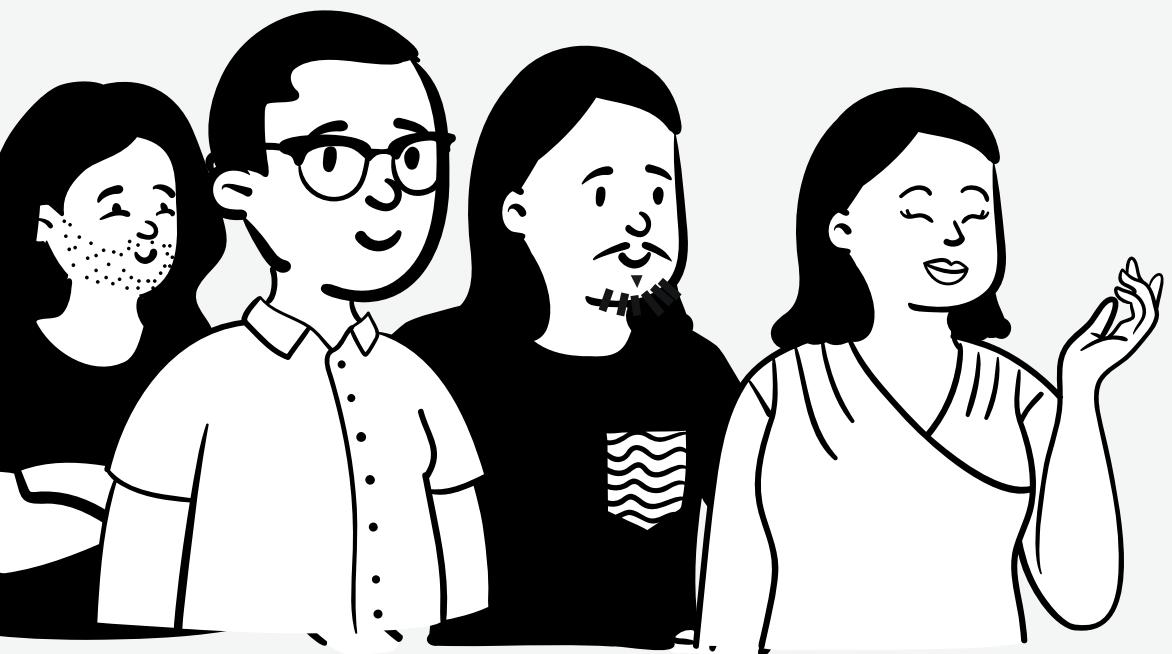
Manoel por sua vez terá que fazer o seguinte antes de aceitar o PR do Paulo:

- Execução de testes;
- Análise de segurança;
- Verificação de estilo de código (Code linter);
- Outros passos afim de manter a qualidade do software;



Exemplo:

Embora possa parecer um processo direto, a complexidade surge do fato de que Paulo é apenas um dos muitos desenvolvedores que contribuem para o projeto liderado por Manoel.

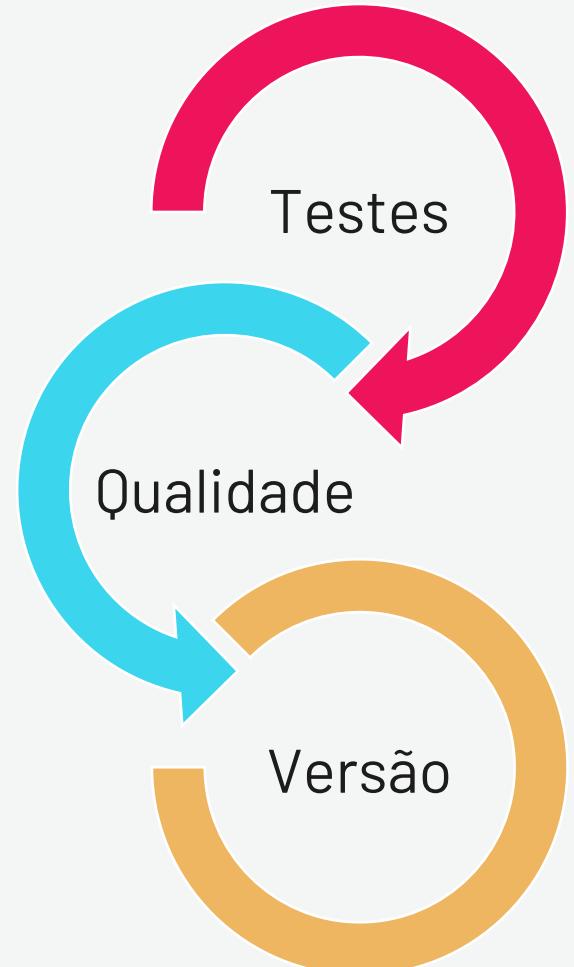


Automação de processos:

Imagine passar o código-fonte por um "funil" de testes automatizados para criar uma nova versão. Esse processo ocorre comumente durante o versionamento em plataformas como **GitHub, GitLab ou BitBucket**.

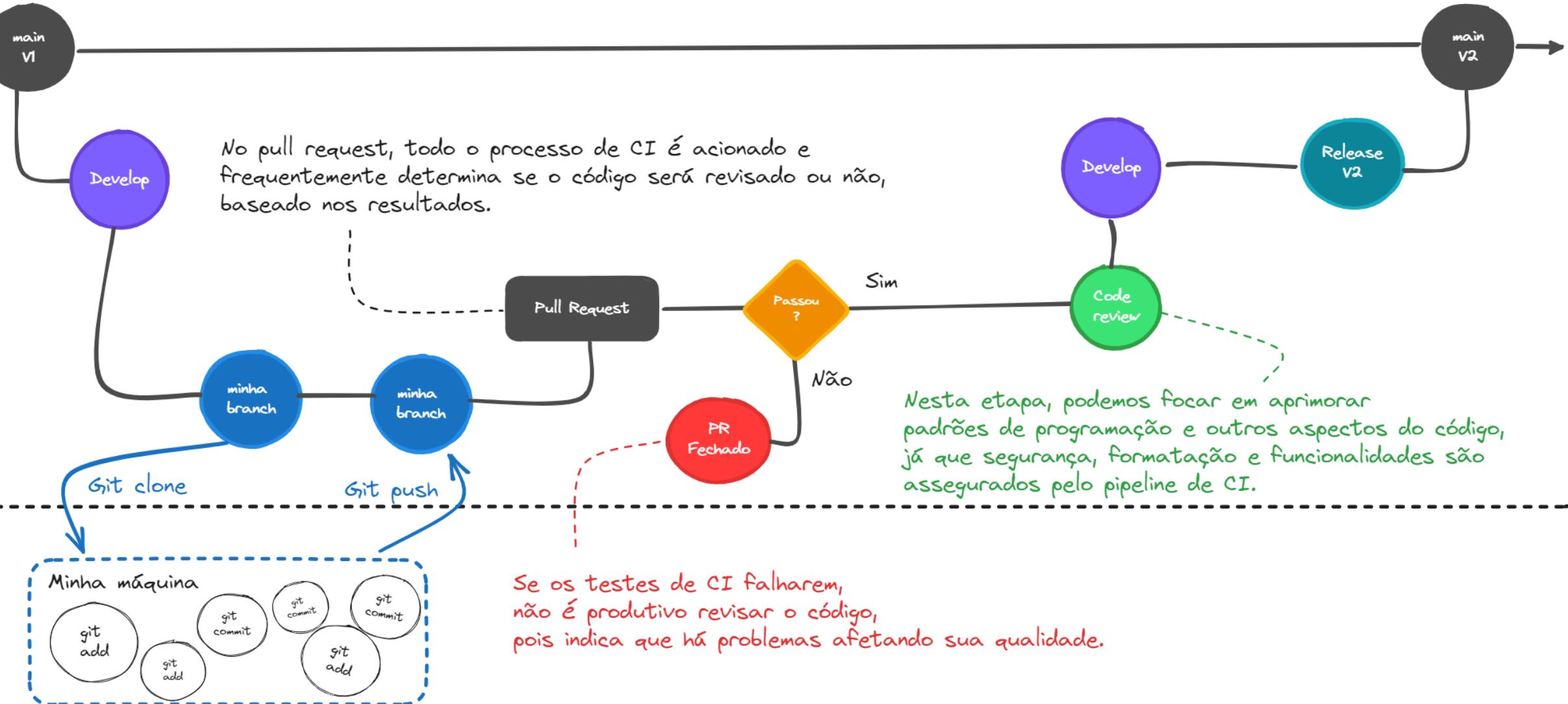
Assim, cada contribuição passa por uma avaliação sistemática antes de ser integrada a Branch principal.

Importante: Isso complementa, mas não substitui, revisões de código e outras análises manuais.



Flow de desenvolvimento:

Github (Code Base)



Exemplo:

Com a automação dessas etapas, Manoel ganha mais tempo para se dedicar a outras tarefas importantes e a probabilidade de erros ou omissões humanas cai drasticamente, já que a máquina realiza o processo com consistência e precisão.



Continuous Delivery (CD)

Continuous Delivery [CD]

A **Entrega Contínua (CD)** representa uma evolução significativa da Integração Contínua (CI), ampliando o foco da automação para além da integração de código, visando garantir que cada **versão gerada** esteja não apenas integrada, **mas totalmente pronta para ser lançada em ambiente de produção.**

A entrega contínua automatiza a implantação de uma versão em um **ambiente para preparação ou teste.**

Existe uma extensão da Entrega Contínua, **a Implantação Contínua.**

Continuous Deployment [CD]

A **Implantação Contínua** é uma extensão da Entrega Contínua, caracterizando-se por uma automação avançada que permite a implantação automática de cada nova versão do software.

Essa versão, uma vez que passa por todo o pipeline de desenvolvimento – incluindo etapas rigorosas de testes –, é **automaticamente movida para o ambiente de produção**.

Este processo assegura que as atualizações sejam feitas de forma contínua e **sem intervenção manual**.

Principais Processos: **Continuous Delivery**

- A entrega contínua implanta automaticamente versões em um **ambiente de teste ou preparação**.
- A entrega contínua **requer intervenção humana** para implantar uma versão da preparação para a produção .
- A entrega contínua **não implanta automaticamente alterações de código na produção** .



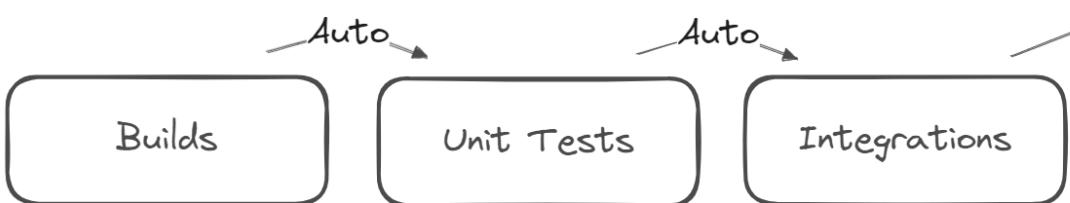
Principais Processos: **Continuous Deployment**

- A **implantação contínua** implanta automaticamente versões desde a construção até o teste e a **produção**.
- A implantação contínua **não requer intervenção humana** .
- A implantação contínua não garante que sua cultura e protocolo de testes estejam à altura . Ele simplesmente **procura uma caixa marcada** e, se encontrar que a caixa está marcada, implanta a **versão na produção**.

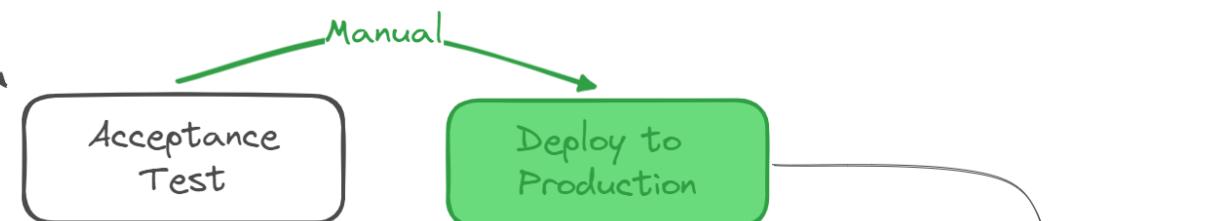


Fluxo Completo:

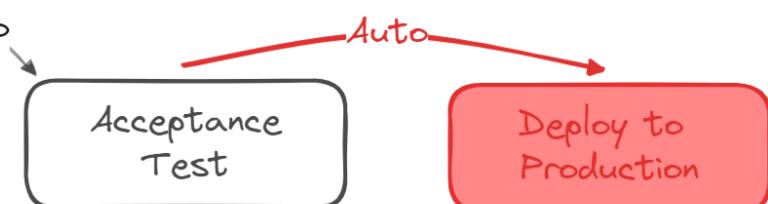
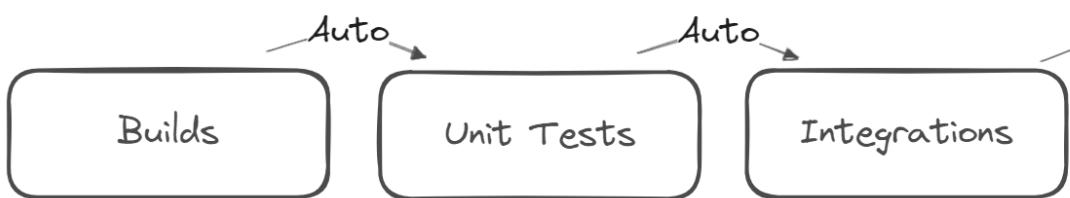
Continuous Integration



Continuous Delivery



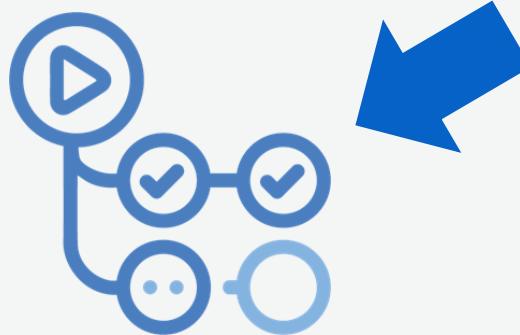
Continuous Deployment



Pontos Cruciais a Considerar:

- ❑ Para uma implementação eficaz da **Integração Contínua (CI)** e da **Entrega Contínua (CD)**, é essencial uma **mudança de mentalidade na equipe**. Isso inclui enfatizar a importância de fazer entregas que foram rigorosamente **testadas**, configurar corretamente o **ambiente de desenvolvimento**, e promover uma **colaboração eficiente** entre todos os membros do time.
- ❑ Os processos de **CI/CD simplificam a automação de verificações críticas** que, sob métodos tradicionais, exigiriam uma revisão manual, garantindo assim a **manutenção da qualidade do código**. Para empresas que ainda não possuem essa cultura, é crucial **desenvolver esses processos** antes de poder **automatizá-los**.
- ❑ A implementação de CI/CD, apesar de demandar um **investimento inicial significativo**, oferece retornos valiosos: **reduz erros humanos, aumenta a produtividade do time** e **diminui a frequência de falhas em produção**, resultando em uma **economia substancial de custos no longo prazo**.

Principais Ferramentas:



GitHub Actions



AWS Code Build



GitHub Actions:

Uma Ferramenta para Automação

de Workflow

GitHub Actions é uma plataforma de automação de **workflow integrada ao GitHub**, que permite aos desenvolvedores e às equipes automatizar seus pipelines de software diretamente dentro de seus repositórios. Essa ferramenta **robusta transforma o GitHub em um ambiente de CI/CD (Integração Contínua e Entrega Contínua)**, facilitando a execução de uma série de operações automaticamente quando eventos específicos ocorrem no repositório.

Funcionalidades e Benefícios:

- ❑ **Eventos Disparadores (Triggers):** GitHub Actions permite acionamento por eventos como **push, pull requests e criação de tags**, oferecendo flexibilidade para adaptar workflows às necessidades do projeto.
- ❑ **Workflows Customizáveis:** Possibilidade de criar workflows personalizados com um arquivo **YAML**, incluindo múltiplos jobs que podem operar sequencialmente ou paralelamente, em diferentes sistemas operacionais e ambientes (máquinas virtuais ou contêineres Docker).

Funcionalidades e Benefícios:

- ❑ **Integração e Entrega Contínuas:** Facilita a automação de testes, construção e deploy, acelerando o desenvolvimento e aumentando a estabilidade e confiabilidade das versões.
- ❑ **Ambiente Rico em Recursos:** Oferece integrações com diversas ferramentas e serviços, ampliando funcionalidades para além de CI/CD, como gerenciamento de dependências e notificações.
- ❑ **Mercado de Actions(Marketplace):** Acesso ao Marketplace do GitHub para utilizar Actions prontas, economizando tempo e permitindo a reutilização de automações desenvolvidas globalmente.

Hands On: Continuous Integration [CI]

Objetivos:

Nossos objetivos nesta seção incluem:

1. Configurar um projeto de exemplo;
2. Estabelecer um pipeline de CI que contém:
 - Uma etapa de compilação;
 - Uma etapa de teste unitário;
 - Uma etapa de geração de imagem Docker;
 - Uma etapa de publicação de imagem no Dockerhub;
3. Implementar um processo de governança no repositório;
4. Aprender a gerar chaves de acesso no Dockerhub;
5. Aprender a registrar variáveis de ambiente no GitHub Secrets.

Por onde começar:

Para este exemplo, iremos desenvolver um **projeto Spring**.

Necessitaremos apenas da seguinte configuração:

The screenshot shows the Spring Initializr web interface for creating a new Spring Boot project. The configuration is as follows:

- Project:** Maven (selected)
- Language:** Java (selected)
- Spring Boot:** 3.2.5 (selected)
- Project Metadata:**
 - Group: sptech.school
 - Artifact: exemplo-actions
 - Name: exemplo-actions
 - Description: Demo project for Spring Boot
 - Package name: sptech.school.exemplo-actions
 - Packaging: Jar (selected)
 - Java: 22 (selected)
 - Java: 21 (selected)
 - Java: 17 (selected)
- Dependencies:**
 - Spring Web** (WEB): Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
 - Spring Boot Actuator** (OPS): Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.
- ADD DEPENDENCIES... CTRL + B** button

Código para teste

The image shows a screenshot of the Visual Studio Code (VS Code) interface. On the left is the Explorer sidebar, which displays the project structure for 'EXEMPLO-ACTIONS [WSL: UBUNTU]'. The 'src' folder contains 'main', 'java', 'resources', 'test', 'target', and configuration files like '.gitignore' and 'pom.xml'. The 'java' folder has subfolders 'sptech' and 'school', with 'exemploactions' containing 'controller' (with 'CalculadoraController.java') and 'service' (with 'CalculadoraService.java'). The 'ExemploActionsApplication.java' file is also listed under 'school'. The main editor area shows two Java files: 'CalculadoraController.java' and 'CalculadoraService.java'. 'CalculadoraController.java' defines a REST endpoint for division. 'CalculadoraService.java' implements the division logic, throwing an exception if the divisor is zero. A screenshot of a browser or API client at the bottom right shows a POST request to 'localhost:8080/divisao/10/2.75' returning a 200 OK response with the value '3.6363636363636362'.

```
10 @RestController
11 public class CalculadoraController {
12
13     private final CalculadoraService calculadoraService;
14
15     public CalculadoraController(CalculadoraService calculadoraService) {
16         this.calculadoraService = calculadoraService;
17     }
18
19     @PostMapping("/divisao/{a}/{b}")
20     public ResponseEntity<Double> divisao(@PathVariable double a, @PathVariable double b) {
21         return ResponseEntity.ok(calculadoraService.dividir(a, b));
22     }
23 }
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100 }
```

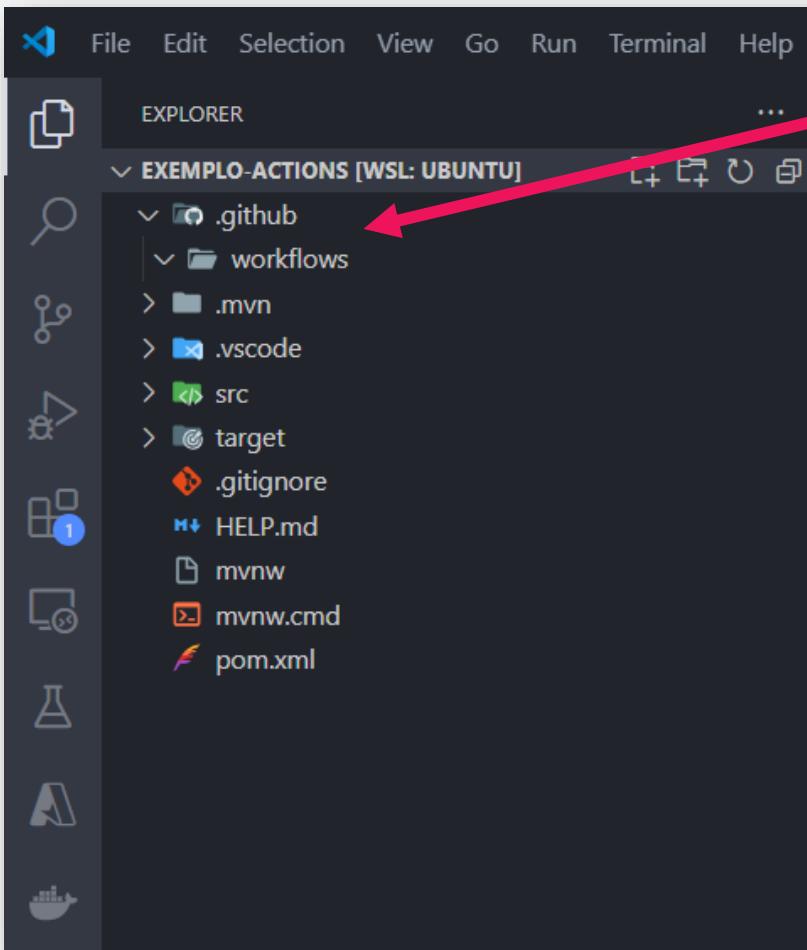
POST localhost:8080/divisao/10/2.75 Send 200 OK 3.6363636363636362 3.78 ms 18 B

Body Auth Query Headers 1 Docs Preview Headers 4 Code

1 3.6363636363636362

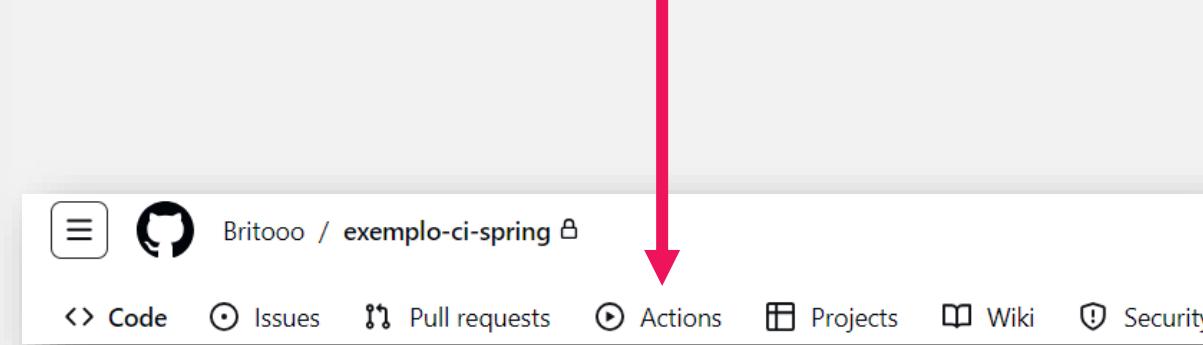
T 3.6363636363636362

Configuração Pipelines



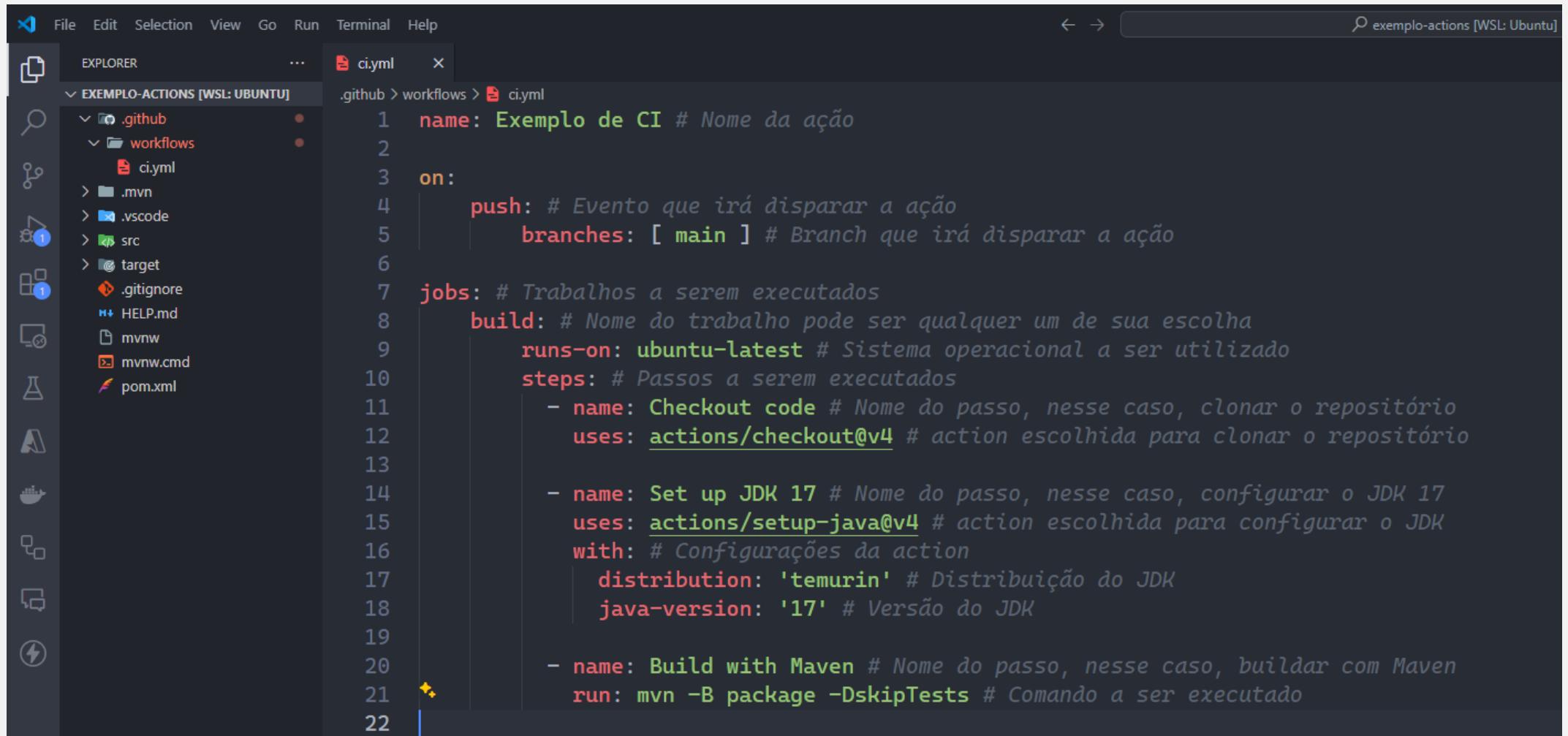
Crie um diretório chamado **.github** e, dentro deste, um subdiretório chamado **workflows**.
É neste local que configuraremos nossos pipelines.

Embora seja possível configurar isso através da interface gráfica, este tutorial guiará você pelo processo manual. Se estiver interessado em explorar a configuração via interface, simplesmente acesse o menu **Actions** no GitHub e siga as instruções detalhadas disponíveis lá.



Arquivo ci.yml

Vamos criar um arquivo chamado `ci.yml` no diretório `workflows`. Neste arquivo, definiremos os passos do nosso pipeline. O primeiro passo é assegurar que o código compila corretamente.



The screenshot shows a dark-themed instance of VS Code. In the top right corner, there's a status bar with the text "exemplo-actions [WSL: Ubuntu]". The left sidebar has a "EXPLORER" tab open, showing a tree view of the repository structure under ".github": ".github", "workflows", and "ci.yml". The "ci.yml" file is selected and shown in the main editor area. The code content is as follows:

```
1 name: Exemplo de CI # Nome da ação
2
3 on:
4   push: # Evento que irá disparar a ação
5     branches: [ main ] # Branch que irá disparar a ação
6
7 jobs: # Trabalhos a serem executados
8   build: # Nome do trabalho pode ser qualquer um de sua escolha
9     runs-on: ubuntu-latest # Sistema operacional a ser utilizado
10    steps: # Passos a serem executados
11      - name: Checkout code # Nome do passo, nesse caso, clonar o repositório
12        uses: actions/checkout@v4 # action escolhida para clonar o repositório
13
14      - name: Set up JDK 17 # Nome do passo, nesse caso, configurar o JDK 17
15        uses: actions/setup-java@v4 # action escolhida para configurar o JDK
16        with: # Configurações da action
17          distribution: 'temurin' # Distribuição do JDK
18          java-version: '17' # Versão do JDK
19
20      - name: Build with Maven # Nome do passo, nesse caso, buildar com Maven
21        run: mvn -B package -DskipTests # Comando a ser executado
22
```

Arquivo ci.yml

A **sintaxe do GitHub Actions** é declarativa, permitindo diversas configurações. Este é apenas um exemplo simples. Para mais informações, acesse:

Sintaxe de fluxo de trabalho para o GitHub Actions

O termo "**uses**" indica a utilização de uma '**action**' específica dentro do workflow.

Uma 'action' pode ser entendida como um **repositório Git** que contém lógicas desenvolvidas para facilitar e automatizar etapas específicas de um **pipeline**.

Há uma vasta gama de '**actions**' disponíveis, que podem ser encontradas em:

Marketplace

The screenshot shows the GitHub Marketplace interface. At the top, there's a search bar with the query "sort:popularity-desc" and a dropdown menu set to "Sort: Most installed/starred". Below the search bar, the results are filtered by the "Actions" category. The results list includes several actions:

- TruffleHog OSS** (Creator verified by GitHub): An infographics generator with 40+ plugins and 300+ options to display stats about your GitHub account. 12.7k stars.
- yq - portable yaml processor** (Creator verified by GitHub): A tool for creating, reading, updating, deleting, merging, validating, and doing more with yaml. 10.9k stars.
- Gosec Security Checker** (By securego): Runs the gosec security checker. 7.5k stars.
- Metrics embed** (By lowlighter): An infographics generator with 40+ plugins and 300+ options to display stats about your GitHub account. 12.7k stars.
- Super-Linter** (By super-linter): It is a simple combination of various linters, written in bash, to help validate your source code. 9.2k stars.
- OpenCommit — improve commits with AI** (By di-sukharev): Replaces lame commit messages with meaningful AI-generated messages when you push to remote.

On the left side of the main content area, there's a sidebar titled "About YAML" which provides basic information about YAML files used in GitHub Actions. The sidebar includes sections for "Workflow files use YAML", "About GitHub Actions", "About GitHub", and "GitHub Help".

Configurando o repositório

Acesse o **GitHub** e crie um novo **repositório**, optando por **não** incluir um arquivo **README** inicialmente.

Em seguida, dentro do diretório do seu projeto Java Spring, execute os comandos fornecidos na página inicial do repositório recém-criado.

The screenshot shows a GitHub repository page for 'exemplo-actions'. At the top, there are buttons for 'Unwatch' (1), 'Fork' (0), and 'Star' (0). Below the header, there's a 'Start coding with Codespaces' section with a 'Create a codespace' button. To the right, there's a 'Add collaborators to this repository' section with a 'Invite collaborators' button. A blue arrow points from the bottom left towards the 'Quick setup' section. The 'Quick setup' section contains instructions: 'if you've done this kind of thing before', a code input field with 'git@github.com:Britooo/exemplo-actions.git', and a note: 'Get started by creating a new file or uploading an existing file. We recommend every repository include a README, LICENSE, and .gitignore.' Below this, another section says '...or create a new repository on the command line' with the following terminal commands:

```
echo "# exemplo-actions" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:Britooo/exemplo-actions.git
git push -u origin main
```

Troque por:
git add .

Isso garantirá que **todos os arquivos** sejam incluídos no próximo commit para o repositório remoto.

Primeira execução do Pipeline

Quando você verificar o commit no repositório remoto, **notará um ícone amarelo** ao lado da mensagem do commit. Isso indica que o nosso pipeline foi ativado, uma vez que configuramos o **evento de push na branch main**. Para acompanhar os detalhes da execução, **clique em "Actions"**.

The screenshot illustrates the GitHub Actions interface for the repository `exemplo-actions`.

Left Panel (Repository Overview): Shows the repository name, private status, branches (main), tags (0), and a recent commit by `Britooo` titled "Initial commit". A blue arrow labeled "observe" points to this commit.

Middle Panel (Actions Tab): Shows the "Actions" tab selected. It displays "All workflows" and "1 workflow run" for "Initial commit". A blue arrow labeled "clique" points to the workflow run card.

Right Panel (Workflow Run Details): Shows the details for "Initial commit #1". It includes:

- Summary:** Triggered via push 4 minutes ago, Status Success, Total duration 26s.
- ci.yml:** on: push
- build:** succeeded 4 minutes ago in 16s
- Log:** Search logs (Shows log entries for each step: Set up job, Checkout code, Set up JDK 17, Build with Maven, Post Set up JDK 17, Post Checkout code, Complete job). A blue arrow labeled "clique" points to the "build" step.

Annotations:

- A blue arrow labeled "observe" points to the "Initial commit" message in the repository overview.
- A blue arrow labeled "clique" points to the "Actions" tab in the repository overview.
- A blue arrow labeled "clique" points to the workflow run card in the "Actions" tab.
- A blue arrow labeled "clique" points to the "build" step in the log details.

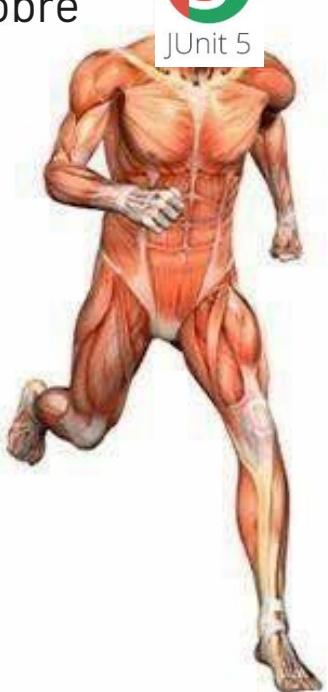
Text Labels:

- Nesta seção, todos os passos definidos em nosso pipeline são listados.
- Se houvesse algum erro, ele seria claramente visível aqui.
- No entanto, neste caso, o projeto foi compilado com sucesso.

Upgrade no Code Base e no pipeline

Vamos aprimorar nosso pipeline para adicionar a **execução de testes na aplicação**.
E sim, acredite ou não, não há truques aqui; vamos escrever **testes unitários** mais uma vez.

Bora escrever
Testes meu nobre

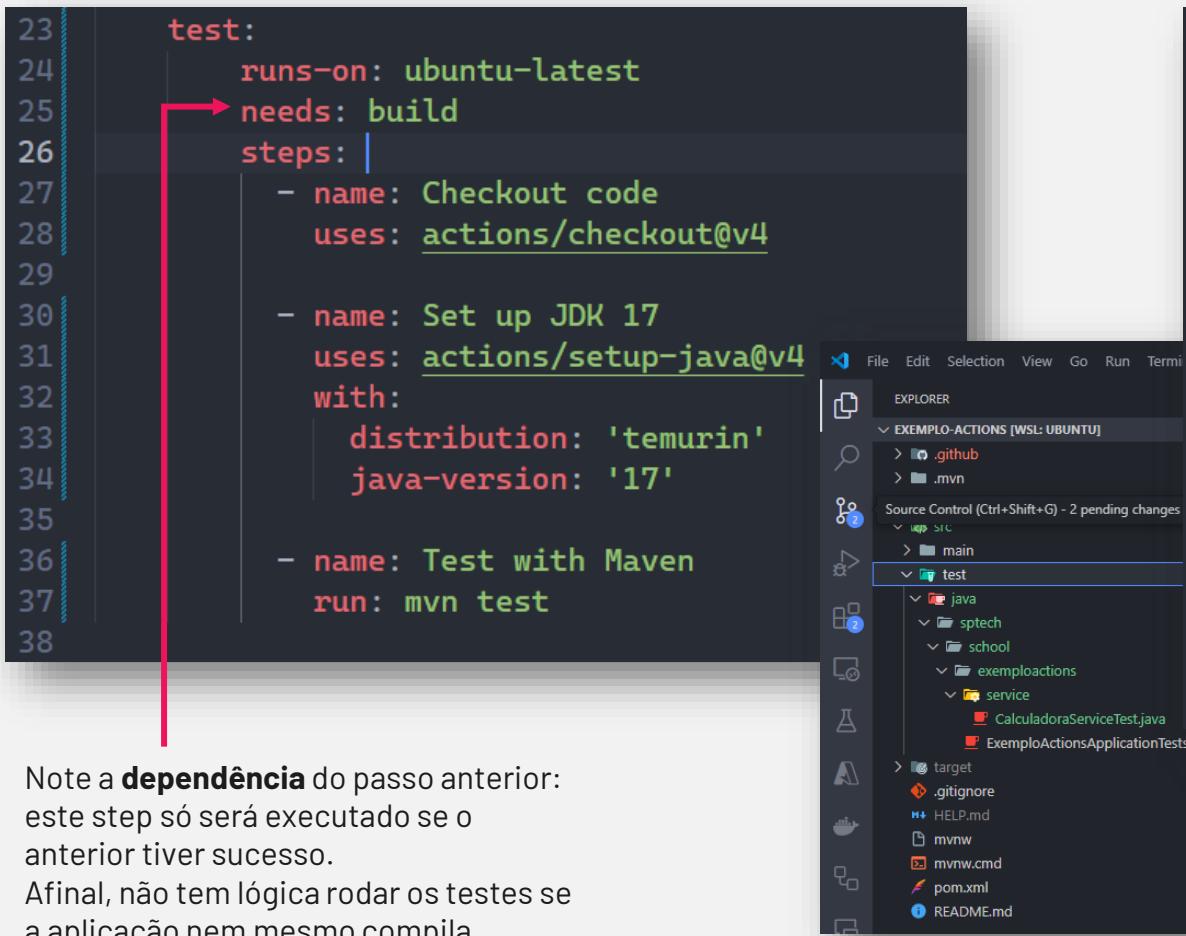


Sai fora maluco
todo dia isso

Upgrade no Code Base e no pipeline

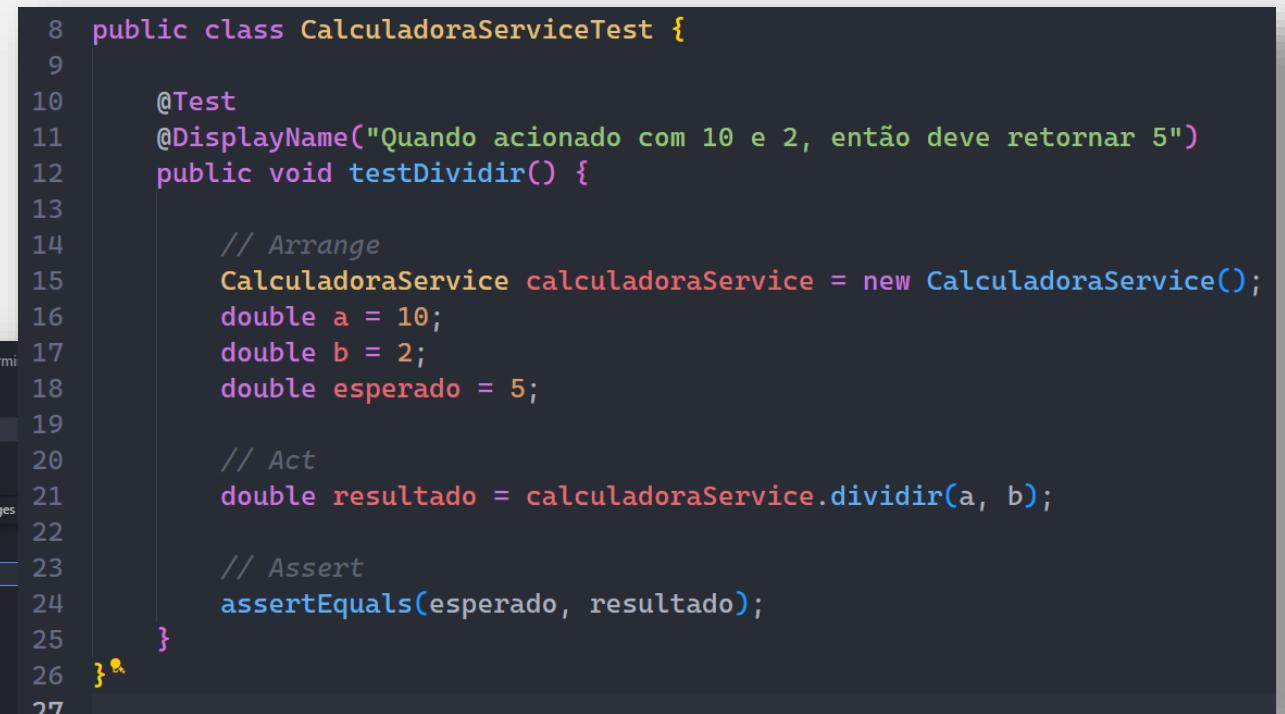
Vamos adicionar mais um Job ao nosso pipeline, desta vez dedicado exclusivamente a executar os testes da aplicação. Poderíamos combinar este passo com o de build, mas preferimos isolá-lo para identificar claramente qualquer problema.

```
23 test:  
24   runs-on: ubuntu-latest  
25   needs: build  
26   steps:  
27     - name: Checkout code  
28       uses: actions/checkout@v4  
29  
30     - name: Set up JDK 17  
31       uses: actions/setup-java@v4  
32       with:  
33         distribution: 'temurin'  
34         java-version: '17'  
35  
36     - name: Test with Maven  
37       run: mvn test
```



Note a **dependência** do passo anterior:
este step só será executado se o
anterior tiver sucesso.
Afinal, não tem lógica rodar os testes se
a aplicação nem mesmo compila.

```
8 public class CalculadoraServiceTest {  
9  
10    @Test  
11    @DisplayName("Quando acionado com 10 e 2, então deve retornar 5")  
12    public void testDividir() {  
13  
14        // Arrange  
15        CalculadoraService calculadoraService = new CalculadoraService();  
16        double a = 10;  
17        double b = 2;  
18        double esperado = 5;  
19  
20        // Act  
21        double resultado = calculadoraService.dividir(a, b);  
22  
23        // Assert  
24        assertEquals(esperado, resultado);  
25    }  
26}  
27
```



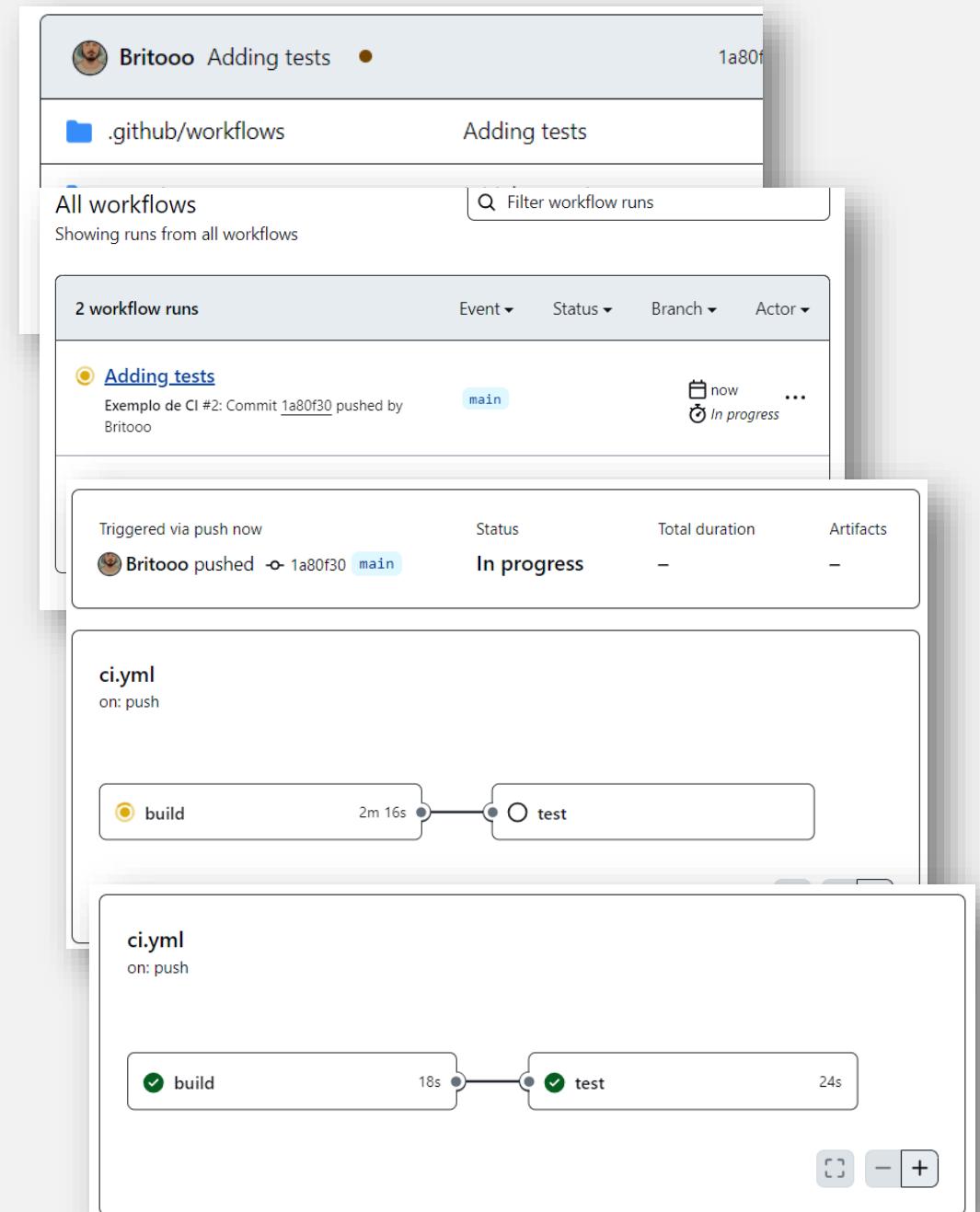
Use atalhos para gerar os testes em sua IDE, isso evitará problemas de configuração de diretórios/arquivos.

Segunda execução do Pipeline

Crie um novo commit com as alterações e envie-o ao repositório remoto. Ao verificar a aba "**Actions**" novamente, você observará nosso novo passo de validação de testes. Se tudo foi feito corretamente, notará que o passo de teste só será executado após o sucesso do build. Com o nosso pipeline configurado dessa maneira, fica fácil identificar se as mudanças enviadas estão corretas ou não. **No entanto, permitir que os desenvolvedores enviem mudanças diretamente para a branch principal pode ser arriscado.**

Imagine se alguém envia um código que **não compila**?

Podemos ajustar nosso pipeline para que o gatilho de ativação seja a abertura de **um Pull Request (PR)**.

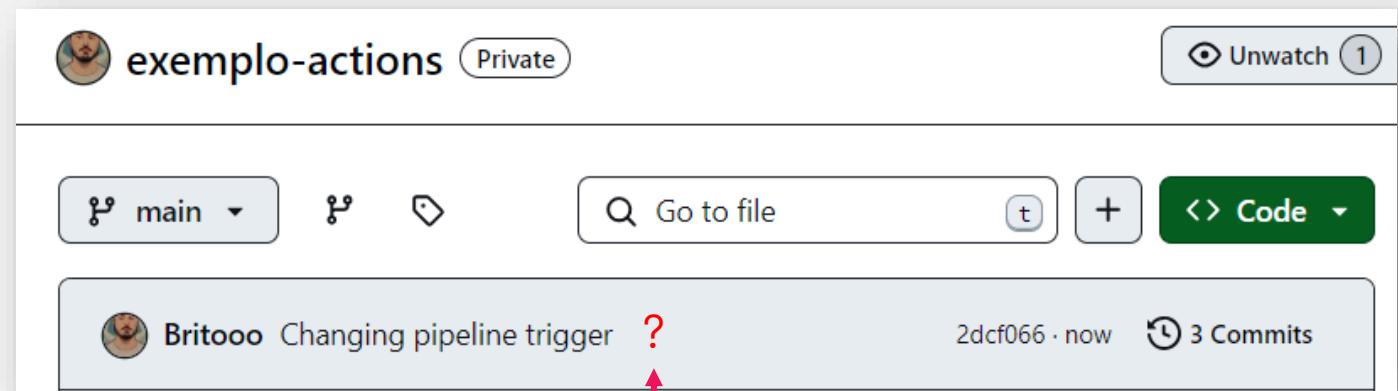


CI somente via Pull Request

Vamos modificar nosso pipeline, alterando o gatilho de acionamento para pull request:

```
3 on:  
4   pull_request:  
5     branches:  
6       - main  
7       - develop  
8     types: [ opened, reopened ]  
9
```

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS AZURE  
  
brito in exemplo-actions  
› git add .  
brito in exemplo-actions  
› git commit -m "Changing pipeline trigger"  
[main 2dcf066] Changing pipeline trigger  
 1 file changed, 6 insertions(+), 3 deletions(-)  
brito in exemplo-actions  
› git push  
Enter passphrase for key '/home/brito/.ssh/github-key':  
Enumerating objects: 9, done.  
Counting objects: 100% (9/9), done.  
Delta compression using up to 8 threads  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (5/5), 503 bytes | 503.00 KiB/s, done.  
Total 5 (delta 2), reused 0 (delta 0), pack-reused 0  
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.  
To github.com:Britooo/exemplo-actions.git  
 1a80f30..2dcf066 main -> main  
brito in exemplo-actions took 5.5s  
›
```



Note que ao enviar para o repositório, nosso pipeline não foi ativado desta vez, porque alteramos o gatilho que o dispara.

Está tudo certo até aqui!

Branch Develop

Vamos criar uma branch intermediária chamada **develop**. Essa branch servirá como um ponto central para todas as alterações de código, **mantendo a main, nossa fonte da verdade**, intacta e somente sendo alterada após uma revisão cuidadosa.

Lembre-se de que a **main** é o nó principal na nossa base de código, e **NUNCA** é recomendado fazer commits diretamente nela.

Embora seja possível criar a branch através da interface do GitHub, vocês já sabem como eu prefiro fazer as coisas, então vamos usar o **terminal**:

Ao concluir, você notará que existem **duas branches** no repositório remoto.

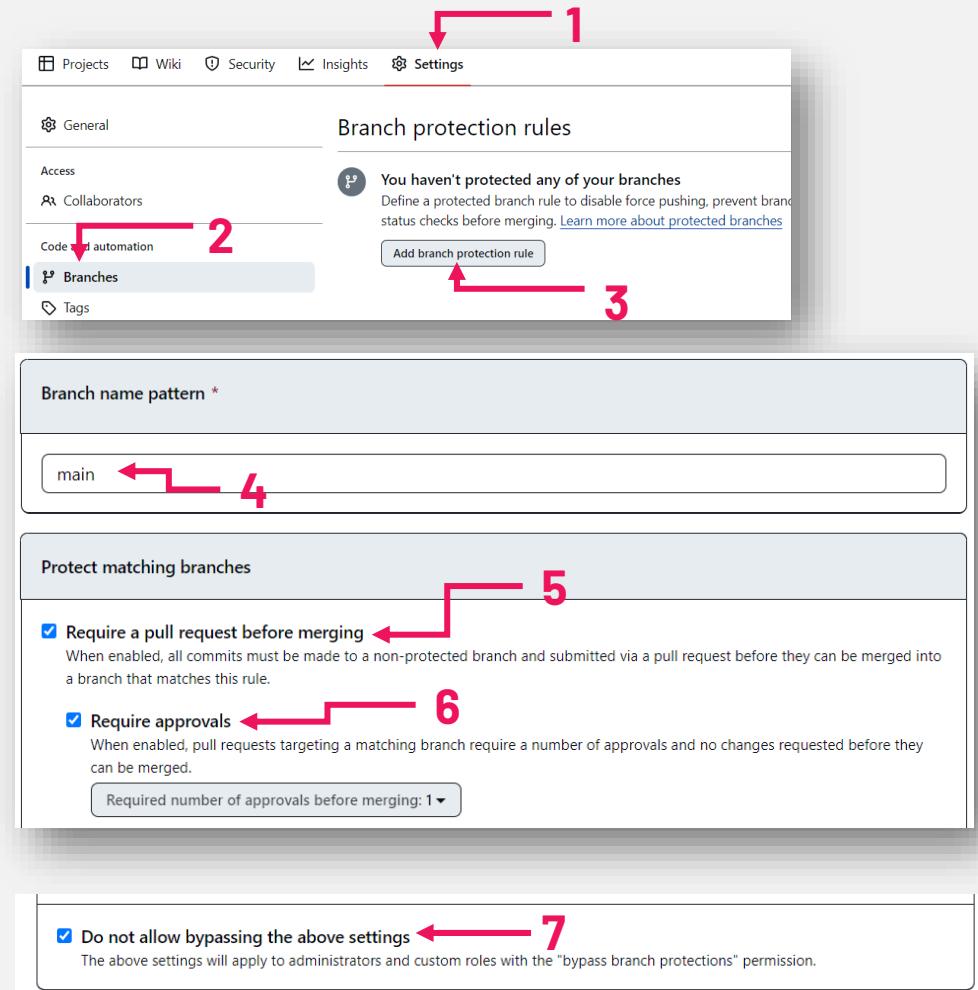
The screenshot shows a terminal session in a dark-themed code editor. The user runs the command `git checkout -b develop` to switch to a new branch named 'develop'. They then run `git push -u origin HEAD` to push this branch to the remote repository. The output shows the branch was created and tracked by 'origin'. A red arrow points from the text 'Execute os comandos abaixo na branch main' to the first line of the terminal output (`git status`). Another red arrow points from the GitHub repository interface at the bottom right to the 'main' branch selection in the terminal's dropdown menu.

```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS AZURE  
brito in exemplo-actions  
› git status  
On branch main  
Your branch is up to date with 'origin/main'.  
nothing to commit, working tree clean  
brito in exemplo-actions  
› |  
  
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS AZURE  
brito in exemplo-actions  
› git checkout -b develop  
Switched to a new branch 'develop'  
brito in exemplo-actions  
› git push -u origin HEAD  
Enter passphrase for key '/home/brito/.ssh/github-key':  
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0  
remote:  
remote: Create a pull request for 'develop' on GitHub by visiting:  
remote: https://github.com/Britooo/exemplo-actions/pull/new/develop  
remote:  
To github.com:Britooo/exemplo-actions.git  
 * [new branch]      HEAD -> develop  
Branch 'develop' set up to track remote  
brito in exemplo-actions took 5.6s  
› |  
  
exemplo-actions (Private)  
main ▾ 2 Branches 0 Tags  
Britooo Changing pipeline trigger
```

Bloqueando acesso direto

Vamos implementar uma regra para bloquear o acesso direto à branch `main`, estabelecendo um processo de aprovação no repositório do GitHub:

1. Acesse as **configurações** do repositório clicando em **Settings**;
2. Dentro de Settings, selecione a opção **Branches**;
3. Clique em **Add Branch protection rule** para adicionar uma nova regra de proteção;
4. No campo **Branch name pattern**, insira o nome da branch que deseja proteger, neste caso, a **main**;
5. Marque a opção **Require a pull request before merging** na seção **Protect matching branches**;
6. Marque também o checkbox **Require approvals** e mantenha o número padrão de 1.
7. Marque a opção **Do not allow bypassing the above settings**, isso impedirá alguém passar por cima da regra;
8. Clique em criar;



Repita esse procedimento para a Branch develop

Bloqueando acesso direto

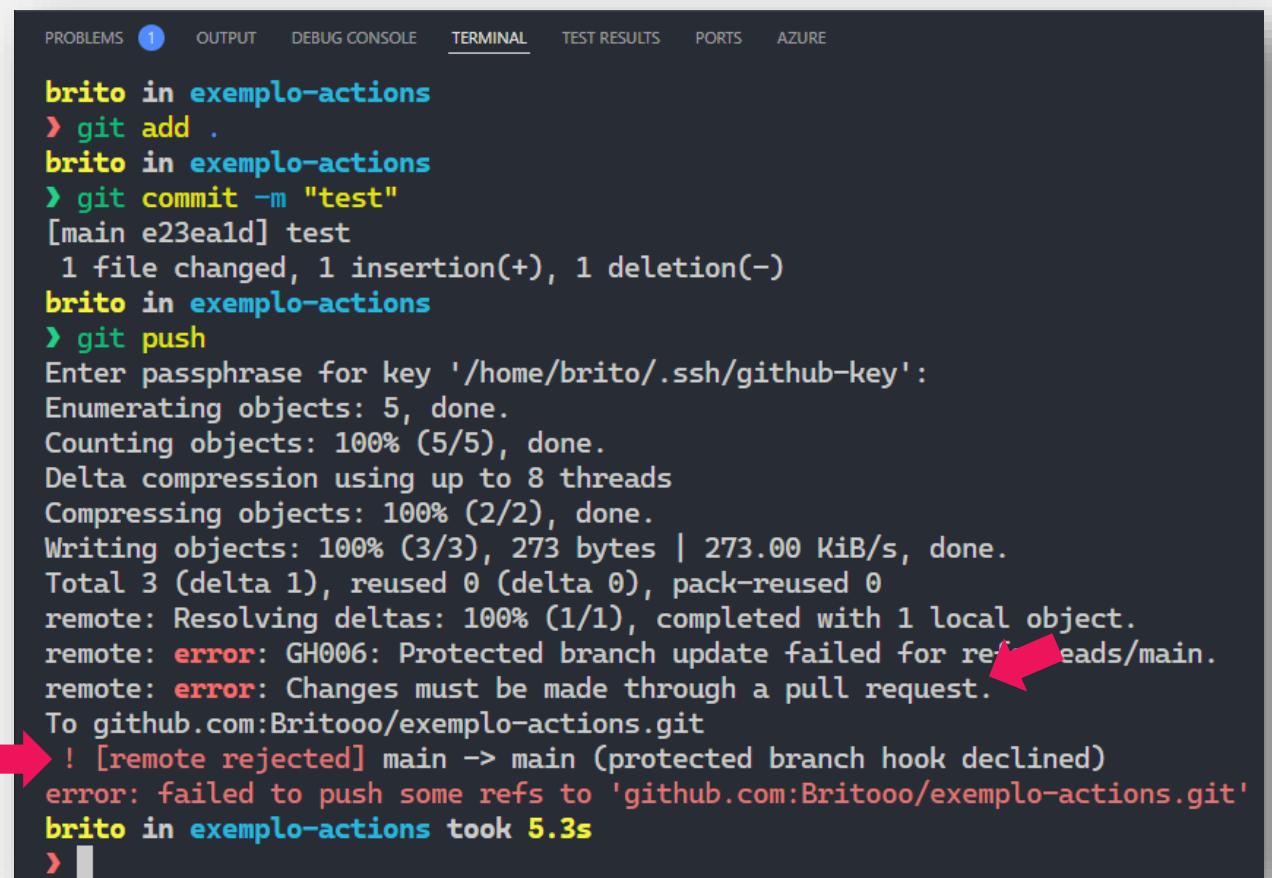
Ao concluir essa configuração, se alguém tentar comprometer sua sexta-feira enviando algo inadequado diretamente à branch **main**, que frequentemente ativa o pipeline de CD, será impedido.

Essa medida é para garantir que não ocorram desastres nas sextas-feiras.

- Mas e se alguém abrir um PR e ele for aprovado, isso não poderia arruinar a sexta-feira da mesma forma?!

- Benjamin Franklin oferece sabedoria que se aplica aqui: **"Diga-me e eu esqueço, ensine-me e eu posso lembrar, envolva-me e eu aprendo."**

Se isso ocorrer, simplesmente peça para a pessoa resolver o problema. Garanto que não se repetirá, seja por escolha própria ou por falta de oportunidade.



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL TEST RESULTS PORTS AZURE

brito in exemplo-actions
> git add .
brito in exemplo-actions
> git commit -m "test"
[main e23ea1d] test
  1 file changed, 1 insertion(+), 1 deletion(-)
brito in exemplo-actions
> git push
Enter passphrase for key '/home/brito/.ssh/github-key':
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 273 bytes | 273.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote: error: GH006: Protected branch update failed for refs/heads/main.
remote: error: Changes must be made through a pull request.
To github.com:Britooo/exemplo-actions.git
 ! [remote rejected] main -> main (protected branch hook declined)
error: failed to push some refs to 'github.com:Britooo/exemplo-actions.git'
brito in exemplo-actions took 5.3s
>
```

The terminal window shows a sequence of git commands: adding a file, committing it with a message, and then pushing to the 'main' branch. The push fails due to a 'Protected branch update failed' error, specifically 'GH006: Changes must be made through a pull request'. Two red arrows point to the error message and the final status line 'brito in exemplo-actions took 5.3s'.

Novo teste e Conventional Commits

O processo no repositório, supondo a criação de um novo teste, seguirá estas etapas:

- Criação de uma branch:** Esta geralmente utiliza um prefixo que indica o conteúdo, como **feat/, fix/, docs/, test/**, etc., seguido do nome. Utilizar [Conventional Commits](#) para melhor organização é recomendado.
- Desenvolvimento na branch:** Realização de adições e commits conforme necessário.
- Abertura de um Pull Request:** Envio da branch para a `develop`.
- Revisão de código:** O Pull Request é submetido ao pipeline de revisão de código.
- Falhas no CI:** Se algum teste do CI falhar, a Pull Request pode ser automaticamente fechada, embora essa decisão, por enquanto, seja deixada a critério do revisor.

```
brito in exemplo-actions
> git status
On branch develop
Your branch is up to date with 'origin/develop'.

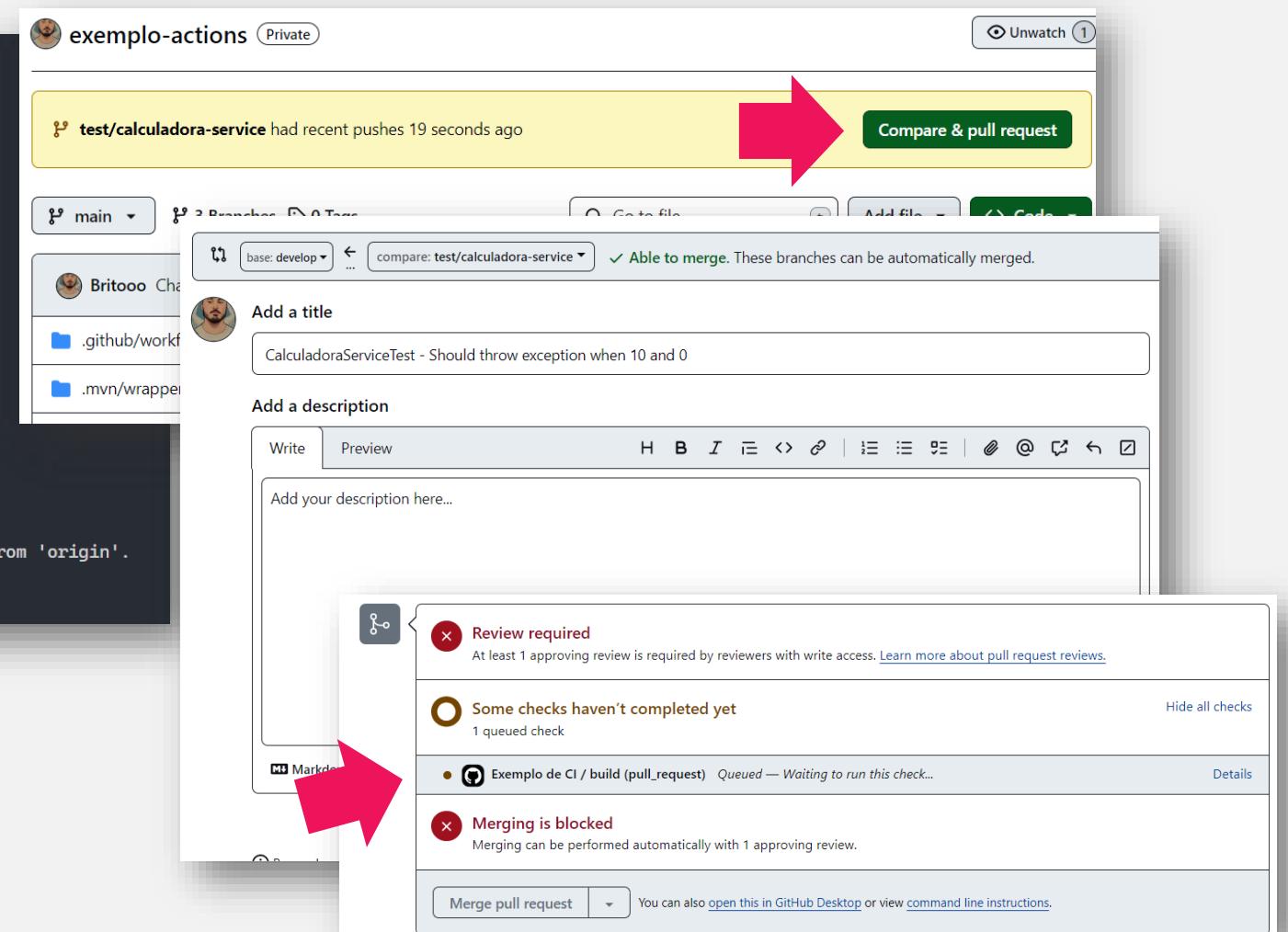
nothing to commit, working tree clean
brito in exemplo-actions
> git checkout -b test/calculadora-service
Switched to a new branch 'test/calculadora-service'
brito in exemplo-actions
>
```

```
29 |     @Test
30 |     @DisplayName("Quando acionado com 10 e 0, então deve lançar uma exceção")
31 |     public void testDividirPorZero() {
32 |
33 |         // Arrange
34 |         CalculadoraService calculadoraService = new CalculadoraService();
35 |         double a = 10;
36 |         double b = 0;
37 |         var expectedMessage = "400 BAD_REQUEST \"Divisão por zero não permitida\"";
38 |
39 |         // Assert
40 |         ResponseStatusException exception = assertThrows(
41 |             ResponseStatusException.class, () -> {
42 |                 // Act
43 |                 calculadoraService.dividir(a, b);
44 |             });
45 |
46 |         // Assert
47 |         assertEquals(expectedMessage, exception.getMessage());
48 |
49 |     }
```

Abrindo PR

Agora, basta abrir o PR para ativar o pipeline. Você pode clicar no link assim que a branch for enviada ou ir diretamente ao GitHub e abrir o PR através da interface:

```
brito in exemplo-actions
> git add .
brito in exemplo-actions
> git commit -m "Should throw exception"
[test/calculadora-service 299f823] Should throw exception
 1 file changed, 24 insertions(+), 1 deletion(-)
brito in exemplo-actions
> git push -u origin HEAD
Enter passphrase for key '/home/brito/.ssh/github-key':
Enumerating objects: 19, done.
Counting objects: 100% (19/19), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (10/10), 1.11 KiB | 1.11 MiB/s, done.
Total 10 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: Create a pull request for 'test/calculadora-service' on GitHub by visiting:
remote:   https://github.com/Britooo/exemplo-actions/pull/new/test/calculadora-service
remote:
To github.com:Britooo/exemplo-actions.git
 * [new branch]      HEAD -> test/calculadora-service
Branch 'test/calculadora-service' set up to track remote branch 'test/calculadora-service' from 'origin'.
brito in exemplo-actions took 5.6s
>
```



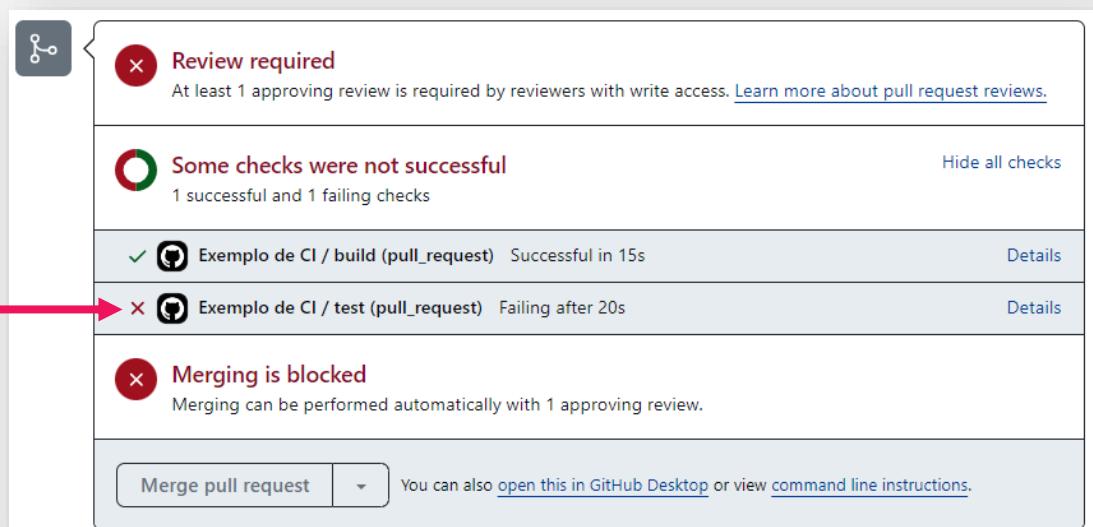
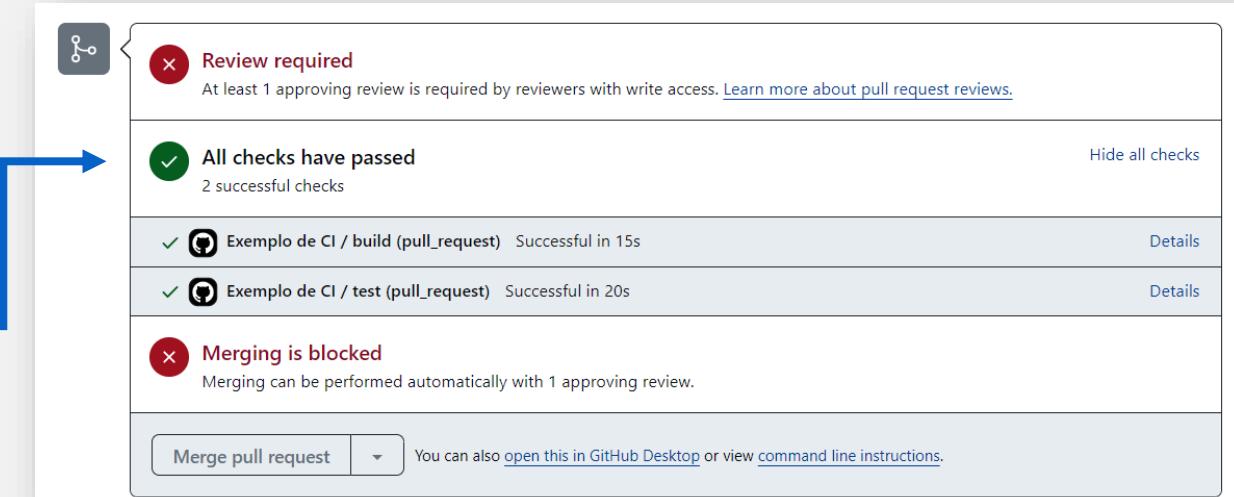
Só correr para o abraço!

Agora é a etapa em que um membro da sua equipe revisará seu código para identificar possíveis gambiarras ou questões relacionadas ao estilo do código.

A tranquilidade vem do fato de que o projeto foi compilado e os testes foram aprovados.

Uma vez aprovado, o botão de merge será liberado para atualizar a branch main.

Se o projeto for enviado com falhas de compilação ou de teste, **o processo será interrompido**, indicando claramente que o commit contém erros, famoso PR bomba, exemplo:



Restringindo ainda mais...

Se desejar, você pode configurar para que apenas os pull requests que passarem em determinados steps sejam revisados. Para isso utilize a **verificação de status**:

1. Clique em **Require status check to pass before merging**;
2. Em seguida, selecione **Require branches to be up to date before merging**;
3. No campo disponível, insira o nome do step desejado e selecione-o;

Você pode adicionar quantos steps considerar necessário, ajuste conforme sua necessidade.

Dessa forma, mesmo que alguém aprove, se o processo mencionado falhar, não será possível realizar o merge.

Protect matching branches

Require a pull request before merging
When enabled, all commits must be made to a non-protected branch and submitted via a pull request before they can be merged into a branch that matches this rule.

Require approvals
When enabled, pull requests targeting a matching branch require a number of approvals and no changes requested before they can be merged.
Required number of approvals before merging: 1 ▾

Dismiss stale pull request approvals when new commits are pushed
New reviewable commits pushed to a matching branch will dismiss pull request review approvals.

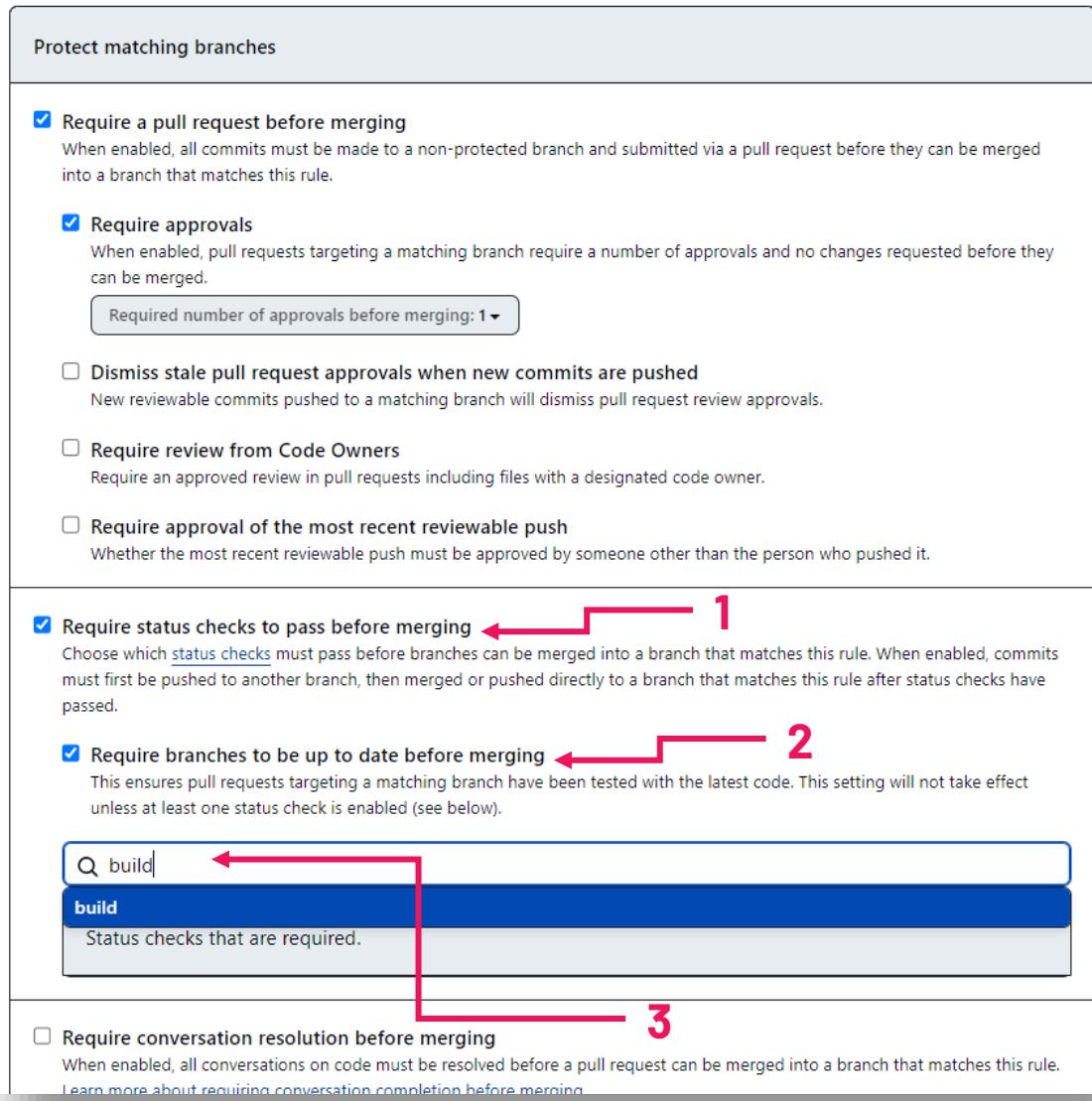
Require review from Code Owners
Require an approved review in pull requests including files with a designated code owner.

Require approval of the most recent reviewable push
Whether the most recent reviewable push must be approved by someone other than the person who pushed it.

Require status checks to pass before merging 1
Choose which status checks must pass before branches can be merged into a branch that matches this rule. When enabled, commits must first be pushed to another branch, then merged or pushed directly to a branch that matches this rule after status checks have passed.

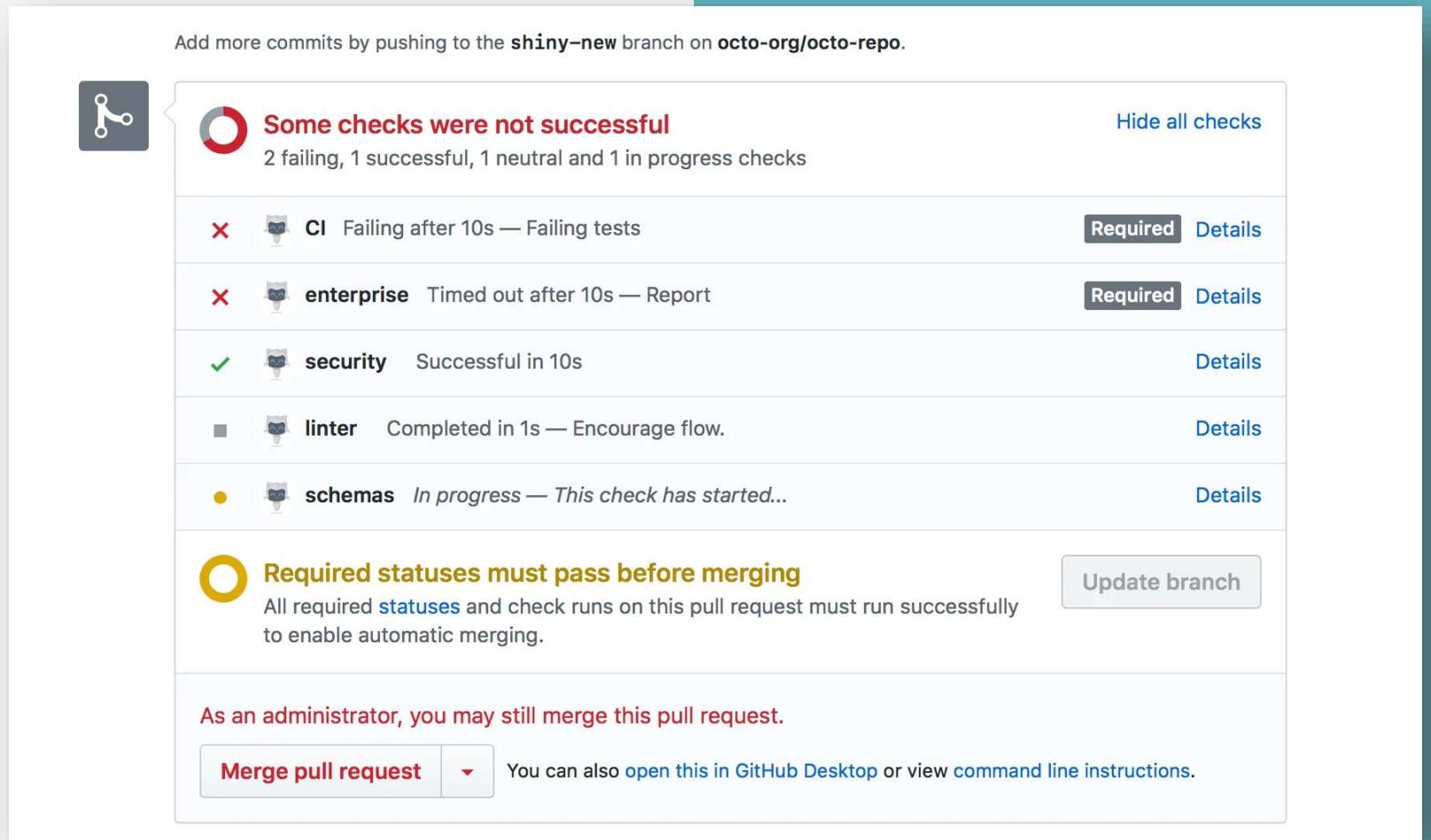
Require branches to be up to date before merging 2
This ensures pull requests targeting a matching branch have been tested with the latest code. This setting will not take effect unless at least one status check is enabled (see below).

Require conversation resolution before merging 3
When enabled, all conversations on code must be resolved before a pull request can be merged into a branch that matches this rule.
[Learn more about requiring conversation completion before merging](#)



Verificação de status

Assegura que uma **Pull Request** não possa ser incorporada ao repositório antes de passar pelo processo de **Integração Contínua** ou pelo processo de revisão de código (Code Review).



Add more commits by pushing to the **shiny-new** branch on **octo-org/octo-repo**.

Some checks were not successful
2 failing, 1 successful, 1 neutral and 1 in progress checks

Check Status	Check Name	Description	Required	Details
X	CI	Failing after 10s — Failing tests	Required	Details
X	enterprise	Timed out after 10s — Report	Required	Details
✓	security	Successful in 10s		Details
■	linter	Completed in 1s — Encourage flow.		Details
●	schemas	In progress — This check has started...		Details

Required statuses must pass before merging
All required [statuses](#) and check runs on this pull request must run successfully to enable automatic merging.

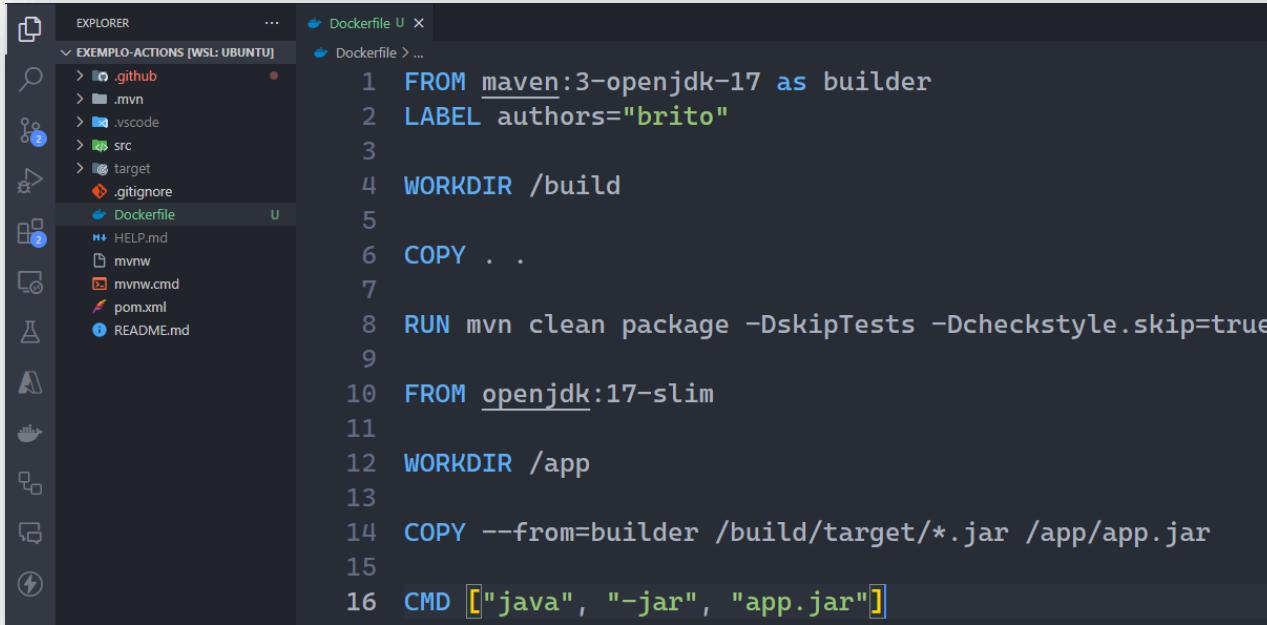
As an administrator, you may still merge this pull request.

[Merge pull request](#) ▾ You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

[Update branch](#)

CI com chave de ouro:

Vamos inserir um step no nosso pipeline para construir a imagem Docker e enviá-la para nossa conta no DockerHub. Serão necessários alguns ajustes para realizar isso...

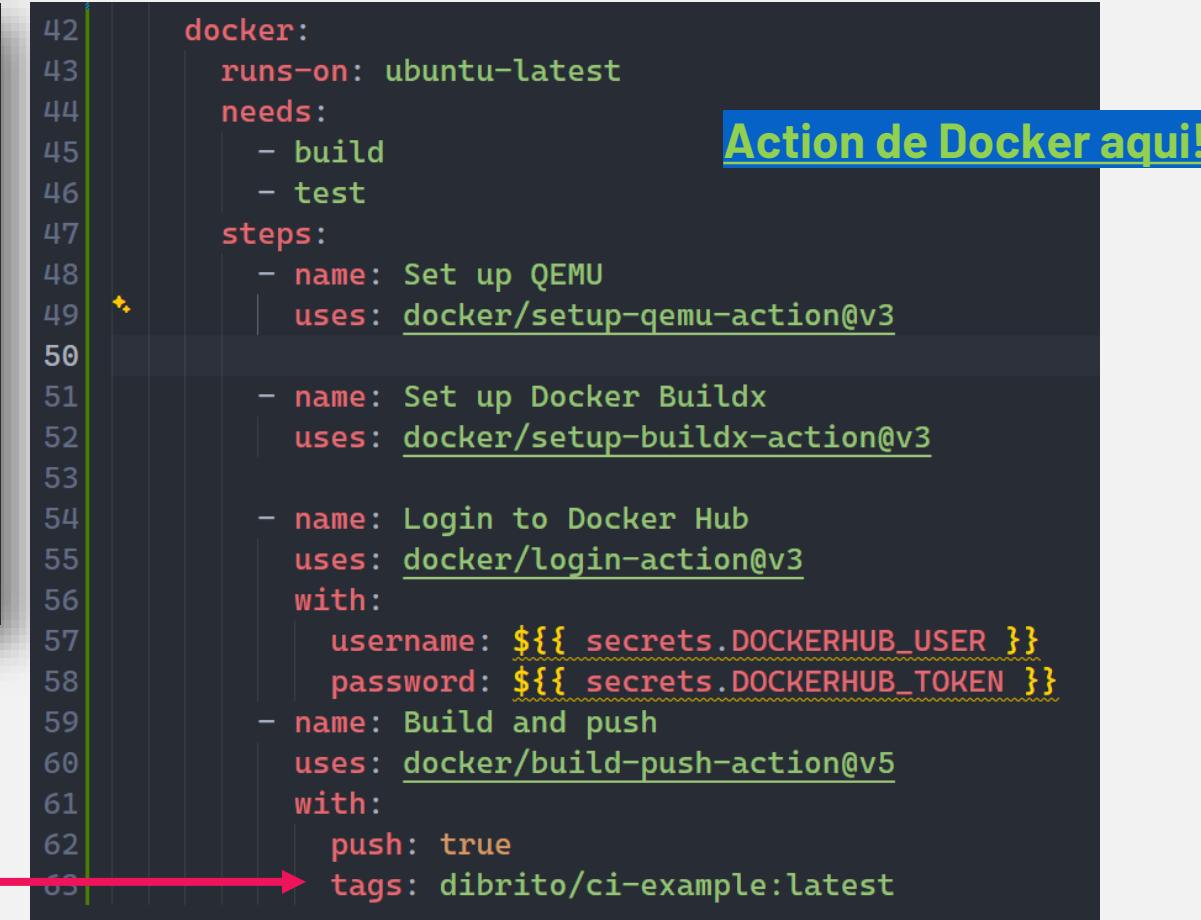


```
FROM maven:3-openjdk-17 as builder
LABEL authors="brito"
WORKDIR /build
COPY . .
RUN mvn clean package -DskipTests -Dcheckstyle.skip=true
FROM openjdk:17-slim
WORKDIR /app
COPY --from=builder /build/target/*.jar /app/app.jar
CMD ["java", "-jar", "app.jar"]
```

1. Criar um **Dockerfile** para nossa aplicação.

Altere para:

usuario/nome-imagem:versao



```
42 docker:
43   runs-on: ubuntu-latest
44   needs:
45     - build
46     - test
47   steps:
48     - name: Set up QEMU
49       uses: docker/setup-qemu-action@v3
50
51     - name: Set up Docker Buildx
52       uses: docker/setup-buildx-action@v3
53
54     - name: Login to Docker Hub
55       uses: docker/login-action@v3
56       with:
57         username: ${{ secrets.DOCKERHUB_USER }}
58         password: ${{ secrets.DOCKERHUB_TOKEN }}
59
60     - name: Build and push
61       uses: docker/build-push-action@v5
62       with:
63         push: true
64         tags: dibrito/ci-example:latest
```

Action de Docker aqui!

2. Criar o step de **build** e **push** no pipeline.

Token Docker Hub

Precisaremos gerar chaves no Dockerhub para utilizarmos em nosso pipeline e logo em seguida adicionar no github secrets, acesse o Dockerhub:

The screenshot shows the Docker Hub dashboard for the user 'dibrito'. At the top, there are tabs for 'Explore', 'Repositories' (which is selected), and 'Organizations'. A search bar says 'Search Docker Hub' with 'ctrl+K' as a keyboard shortcut. On the right, there's a user icon with a 'D' and a dropdown menu. Below the header, there are two repository cards: 'dibrito / ci-example' and 'dibrito / proxy-api'. The 'ci-example' card shows it contains an image, was last pushed 4 days ago, and has 19 pull requests. The 'proxy-api' card shows it contains an image, was last pushed 11 days ago, and has 6 pull requests. The user menu dropdown includes options like 'What's New', 'My Profile', 'My Account', 'Billing', and 'Sign out'. A red arrow labeled '1' points to the user icon in the top right.

The screenshot shows the 'New Access Token' dialog box. It starts with the heading 'New Access Token' and a brief description: 'A personal access token is similar to a password except you can have many tokens and revoke access to each one at any time.' Below this is a field for 'Access Token Description *' containing 'ci-actions'. Under 'Access permissions', it says 'Read, Write, Delete'. A note below states: 'Read, Write, Delete tokens allow you to manage your repositories.' At the bottom are 'Cancel' and 'Generate' buttons. A red arrow labeled '5' points to the 'Access Token Description' field, and another red arrow labeled '6' points to the 'Generate' button.

The screenshot shows the 'Security' tab of the user profile. On the left, there are navigation links for 'General' (selected) and 'Security'. Below this are sections for 'Default Privacy', 'Notifications', 'Convert Account', and 'Deactivate Account'. The main area is titled 'Access Tokens' and lists one token: 'example-ci-spring' (MANUAL, Read, Write, Delete scope, last used May 08, 2024, created May 05, 2024). A red arrow labeled '3' points to the 'Security' tab link, and a red arrow labeled '4' points to the 'New Access Token' button.

Após gerar a chave, anote-a em um local seguro. Em seguida, precisaremos adicioná-la no GitHub, especificamente na seção **"GitHub Secrets"**.

Token Docker Hub

Retorne às configurações do repositório e localize a seção de secrets. Aqui, o objetivo é criar duas variáveis de ambiente, **DOCKERHUB_USER** e **DOCKERHUB_TOKEN**, que serão utilizadas no pipeline.

The screenshot shows the GitHub repository settings for 'exemplo-ci-spring'. A red arrow labeled '1' points to the 'Settings' tab in the top navigation bar. A red bracket labeled '2' covers the sidebar menu, which includes 'General', 'Access', 'Collaborators', 'Code and automation' (with 'Branches', 'Tags', 'Rules', 'Actions', 'Webhooks', 'Environments', 'Codespaces', and 'Pages' listed), 'Security' (with 'Code security and analysis' and 'Deploy keys'), and 'Secrets and variables'. A red arrow labeled '3' points to the 'Secrets and variables' link in the sidebar. A red bracket labeled '4' points to the 'New repository secret' button in the main content area.

The screenshot shows the 'Repository secrets' page for the same repository. It displays a message stating 'This repository has no secrets.' and features a 'New repository secret' button. A red arrow labeled '4' points to this button.

The screenshot shows the 'Repository secrets' page after creating two secrets. The table lists them as follows:

Name	Last updated
DOCKERHUB_TOKEN	now
DOCKERHUB_USER	now

- **DOCKERHUB_USERNAME**: Seu nome de usuário no Dockerhub;
- **DOCKERHUB_TOKEN**: O token gerado no passo anterior;

Em seguida, vamos subir as mudanças pro **Github**.

Só alegria!

Com esse passo configurado, concluímos nosso CI de teste. Agora, seu time pode operar nesse fluxo, sempre produzindo artefatos confiáveis que compilam, passam nos testes, e geram e publicam imagens no seu repositório Docker.

The screenshot shows a GitHub pull request merge interface. On the left, a green checkmark icon indicates "All checks have passed" with 3 successful checks: "Exemplo de CI / build (pull_request) Successful in 16s", "Exemplo de CI / test (pull_request) Successful in 17s", and "Exemplo de CI / docker (pull_request) Successful in 56s". It also states "This branch has no conflicts with the base branch" and "Merging can be performed automatically". A large green "Merge pull request" button is visible. To the right, the GitHub repository page for "dibrito/ci-example" is shown. The repository has one tag, "latest", which was pushed 2 minutes ago. A "Docker commands" section contains the command "docker push dibrito/ci-example:tagname". The "Automated Builds" section explains that manually pushing images to Hub? can be automated by connecting accounts to GitHub or Bitbucket. Both sections mention Pro, Team, and Business subscriptions.

All checks have passed
3 successful checks

✓ Exemplo de CI / build (pull_request) Successful in 16s

✓ Exemplo de CI / test (pull_request) Successful in 17s

✓ Exemplo de CI / docker (pull_request) Successful in 56s

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request

You can also [open this in GitHub Desktop](#)

dibrito/ci-example

Updated 1 minute ago

This repository does not have a description

This repository does not have a category

Docker commands

To push a new tag to this repository:

```
docker push dibrito/ci-example:tagname
```

Tags

This repository contains 1 tag(s).

Tag	OS	Type	Pulled	Pushed
latest		Image	---	2 minutes ago

See all

Automated Builds

Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

Available with Pro, Team and Business subscriptions. [Read more about automated builds](#).

Envie essas alterações para a branch main para que possamos prosseguir com a próxima e crucial etapa: **o processo de CD**.

Hands On: Continuous Delivery [CD]

Mas antes:

Existem diversas formas de realizar um deploy em uma instância EC2. O exemplo a seguir destaca o uso do **GitHub Runner**, um agente disponibilizado pelo próprio GitHub. Fique à vontade para explorar outras abordagens se desejar.

Este método é ideal para projetos de pequeno e médio porte. Para ambientes mais críticos, geralmente se recorre a outras soluções, incluindo serviços como Kubernetes (K8S), Infraestrutura como Código (IAC), entre outras estratégias avançadas.

[GitHub Runner Docs](#)

Bônus: **WatchTower** é uma ferramenta que automatiza o deploy ao monitorar seu DockerHub e atualizar containers automaticamente quando novas imagens são disponibilizadas.

Objetivos:

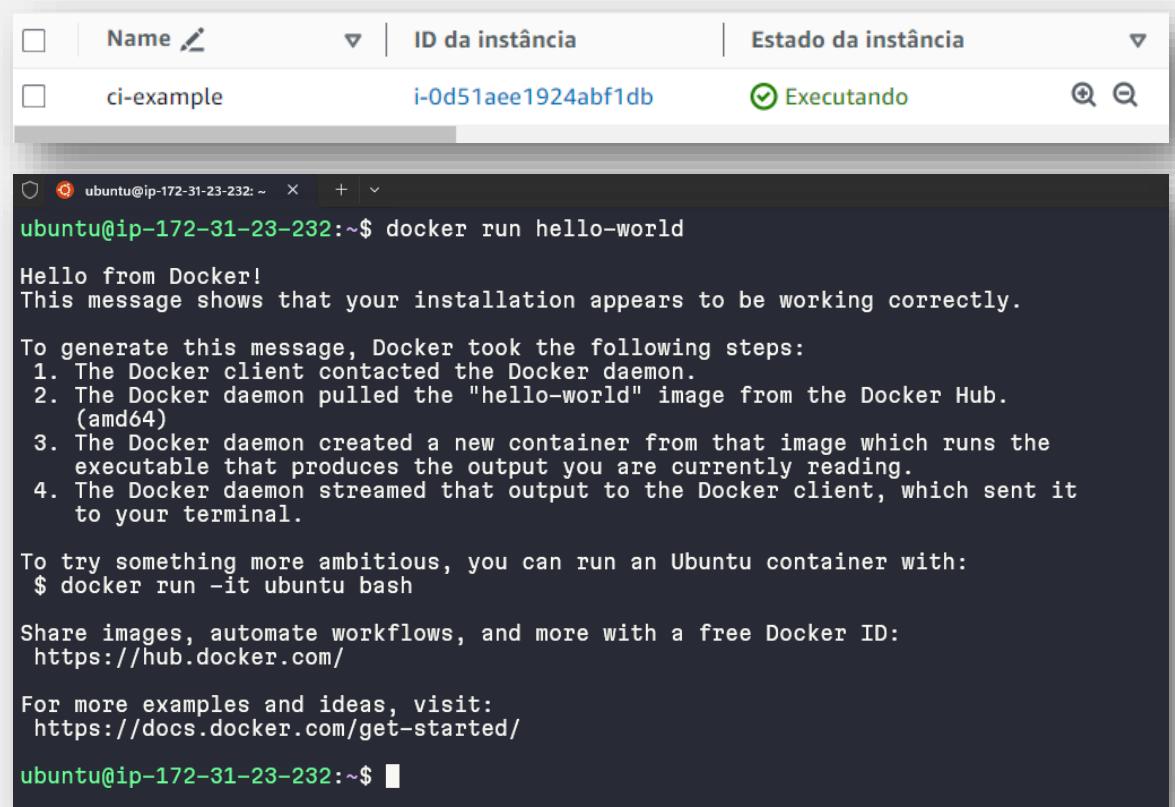
Nossos objetivos nesta seção incluem:

1. Configurar um Runner em uma EC2;
2. Estabelecer um pipeline de CD que contém:
 - Uma etapa que busca a imagem e realiza download;
 - Uma etapa que para o container atual caso ele exista;
 - Uma etapa que remove o container caso ele exista;
 - Uma etapa que roda o container com a imagem recém lançada;
3. Validação do código na nuvem;

Setup EC2

Será necessário configurar uma instância em uma sub-rede com acesso à internet.
Para isso, utilizarei a VPC padrão da AWS.

1. Inicie uma instância EC2;
2. Atribua um **IP público à instância**;
3. Configure o grupo de segurança para **liberar a porta 8080**;
4. Conecte-se à instância via SSH;
5. Execute os comandos a seguir para instalar o Docker:
 - **sudo apt update**
 - **sudo apt install docker.io**
 - **sudo usermod -a -G docker \$(whoami)**
 - **newgrp docker**
 - **docker run hello-world**



The screenshot shows a terminal window with two tabs. The top tab displays a list of EC2 instances with one named 'ci-example' running. The bottom tab shows a terminal session on an Ubuntu instance with the following output:

```
ubuntu@ip-172-31-23-232:~$ docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

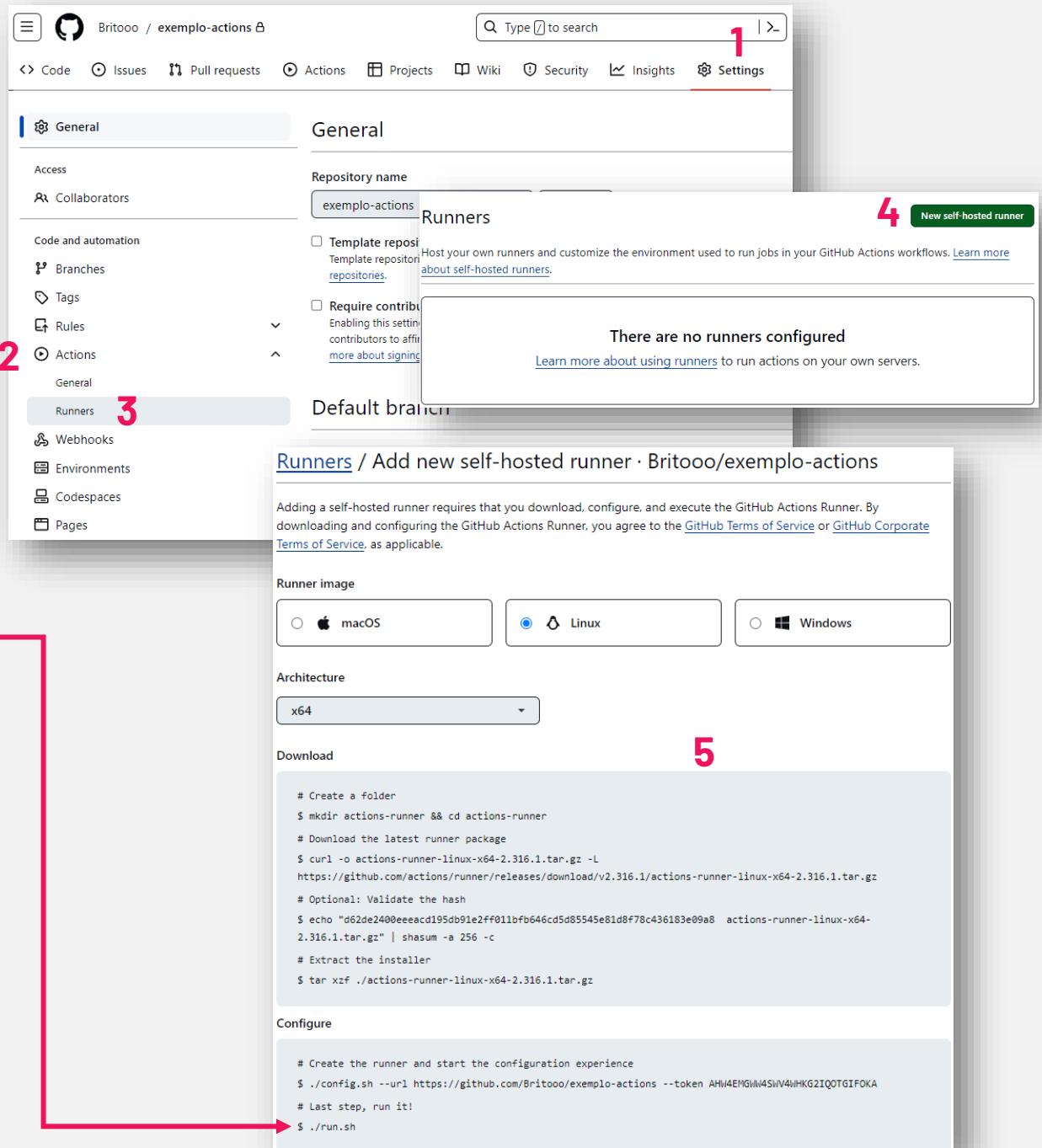
For more examples and ideas, visit:
https://docs.docker.com/get-started/
ubuntu@ip-172-31-23-232:~$ █
```

Setup Runner

Para configurar um runner auto-hospedado em nosso repositório, siga estes passos:

1. Acesse as configurações do repositório;
2. Selecione a aba **Actions**;
3. Clique em **Runners**;
4. Escolha **New self-hosted runner**;
5. Execute o procedimento passo a passo na home da sua instância EC2;
6. No último passo, substitua **./run.sh** por:
 - **nohup ./run.sh > output.log 2>&1 &**

Isso garantirá que o processo seja executado em background e continue rodando mesmo após o fechamento do terminal.



Setup Runner

Durante a configuração, ajuste as definições conforme sua preferência. Após rodar o script do runner e retornar à página de runners do GitHub, você verá sua máquina pronta para receber comandos.

Neste ponto, podemos configurar o processo de CD para executar a imagem Docker nesta instância.

Runners

New self-hosted runner

Host your own runners and customize the environment used to run jobs in your GitHub Actions workflows. [Learn more about self-hosted runners.](#)

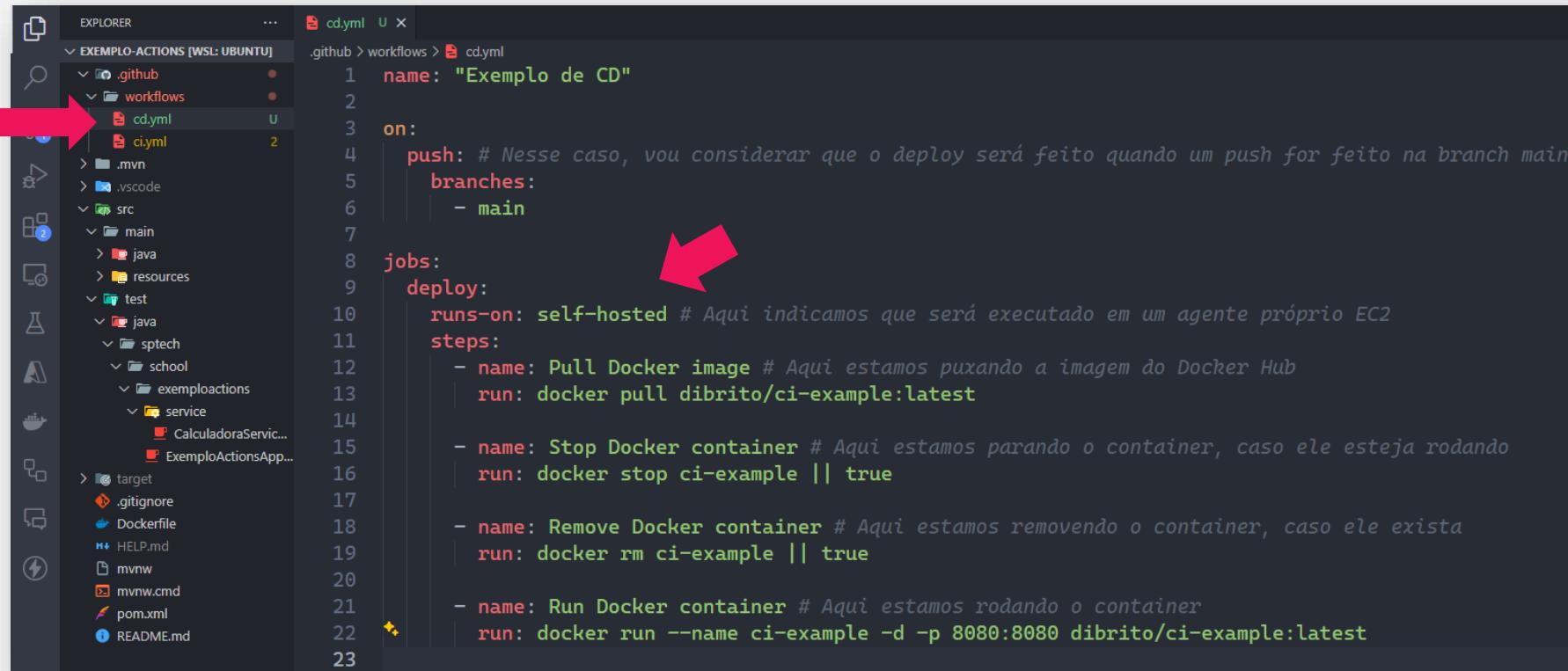
Runners	Status
ci-example <small>self-hosted Linux X64</small>	Idle ...

Self-hosted runner registration

```
# Authentication
✓ Connected to GitHub
# Runner Registration
Enter the name of the runner group to add this runner to: [press Enter for Default]
Enter the name of runner: [press Enter for ip-172-31-23-232] ci-example
This runner will have the following labels: 'self-hosted', 'Linux', 'X64'
Enter any additional labels (ex. label-1,label-2): [press Enter to skip]
✓ Runner successfully added
✓ Runner connection is good
# Runner settings
Enter name of work folder: [press Enter for _work]
✓ Settings Saved.
ubuntu@ip-172-31-23-232:~/actions-runner$ nohup ./run.sh > output.log 2>&1 &
[1] 2484
ubuntu@ip-172-31-23-232:~/actions-runner$ █
```

Finalmente, Continuos Delivery/Deployment

Crie uma branch de trabalho para implementar essas alterações no pipeline; eu denominei a minha de **chore/cd-process**. Em seguida, adicione um novo arquivo ao nosso projeto chamado **cd.yml**, com a configuração a seguir:



```
name: "Exemplo de CD"

on:
  push: # Nesse caso, vou considerar que o deploy será feito quando um push for feito na branch main
  branches:
    - main

jobs:
  deploy:
    runs-on: self-hosted # Aqui indicamos que será executado em um agente próprio EC2
    steps:
      - name: Pull Docker image # Aqui estamos puxando a imagem do Docker Hub
        run: docker pull dibrito/ci-example:latest

      - name: Stop Docker container # Aqui estamos parando o container, caso ele esteja rodando
        run: docker stop ci-example || true

      - name: Remove Docker container # Aqui estamos removendo o container, caso ele exista
        run: docker rm ci-example || true

      - name: Run Docker container # Aqui estamos rodando o container
        run: docker run --name ci-example -d -p 8080:8080 dibrito/ci-example:latest
```

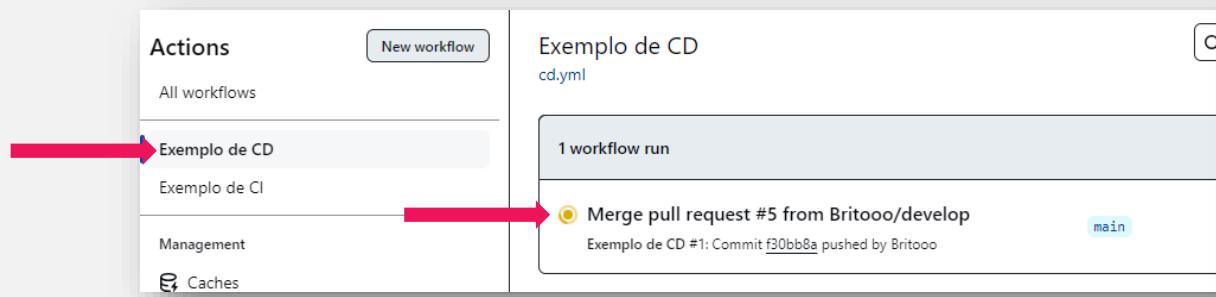
Já que o Docker está instalado em nossa instância, o objetivo agora é automatizar os comandos que normalmente executaríamos manualmente na EC2.

Entregando o código em produção

Nesta fase, normalmente criariamos uma branch de release, mas vou omitir essa parte, pois esses procedimentos podem variar bastante de acordo com o contexto. Para quem deseja se aprofundar mais nesse tema, recomendo consultar:

Git Flow

Após o merge na branch **develop**, iniciaremos outro pull request para a branch **main**.

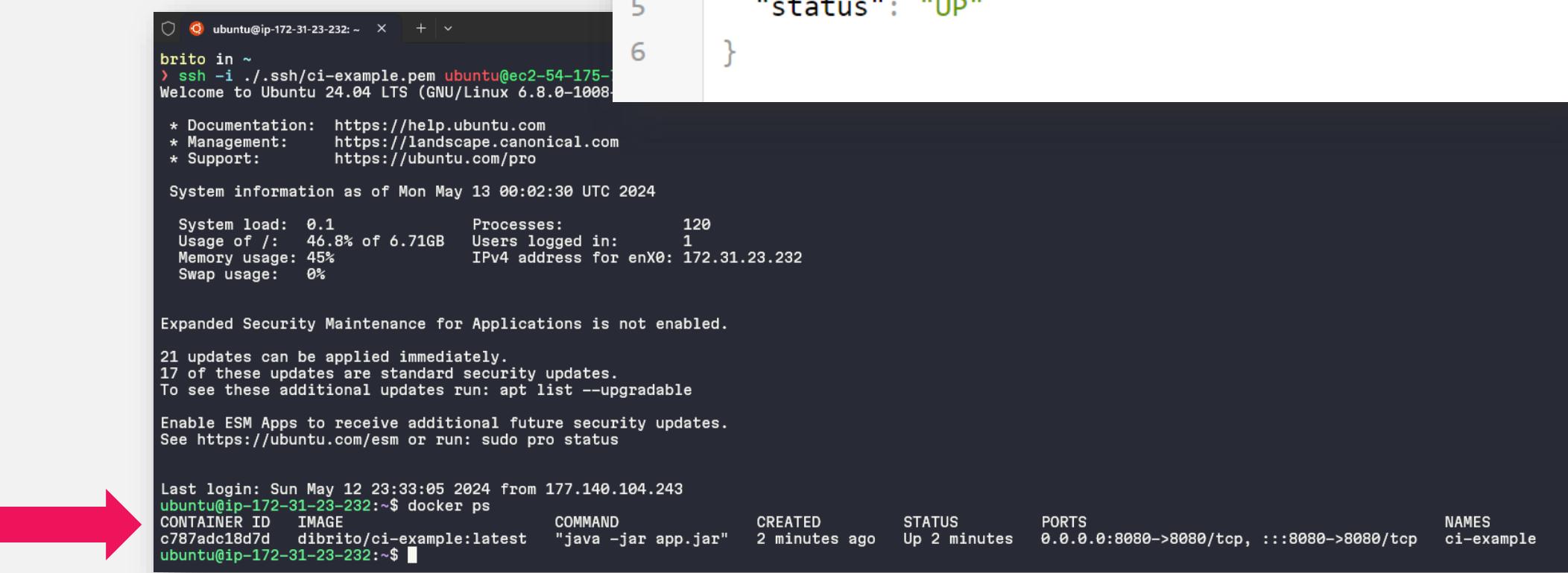


The screenshot shows a GitHub pull request merge screen and a CI pipeline status card. At the top, a pull request for 'CD process #4' is merged into 'Develop #5'. The merge message indicates 7 commits were merged from 'develop' into 'main'. Below this, a CI pipeline status card for 'cd.yml' on 'push' shows a successful run with a duration of 23s. The pipeline steps include 'Triggered via push now', 'Britooo pushed f30bb8a main', 'Status Success', 'Total duration 23s', and 'Artifacts -'. The bottom right of the pipeline card has the text 'Agora é só testar!' (Now it's time to test!).

Teste de container na EC2

Ao acessar a EC2, podemos ver que o container está rodando.

Podemos para fim de teste, acessar o endpoint:
http://IP-MAQUINA:8080/actuator/health



The terminal session shows the following output:

```
brito in ~
> ssh -i ./ssh/ci-example.pem ubuntu@ec2-54-175-
Welcome to Ubuntu 24.04 LTS (GNU/Linux 6.8.0-1008)

 * Documentation: https://help.ubuntu.com
 * Management: https://landscape.canonical.com
 * Support: https://ubuntu.com/pro

System information as of Mon May 13 00:02:30 UTC 2024

 System load: 0.1          Processes:           120
 Usage of /: 46.8% of 6.71GB  Users logged in:      1
 Memory usage: 45%          IPv4 address for enX0: 172.31.23.232
 Swap usage:  0%           

Expanded Security Maintenance for Applications is not enabled.

21 updates can be applied immediately.
17 of these updates are standard security updates.
To see these additional updates run: apt list --upgradable

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

Last login: Sun May 12 23:33:05 2024 from 177.140.104.243
ubuntu@ip-172-31-23-232:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
c787adc18d7d dibrito/ci-example:latest "java -jar app.jar" 2 minutes ago Up 2 minutes 0.0.0.0:8080->8080/tcp, :::8080->8080/tcp ci-example
ubuntu@ip-172-31-23-232:~$
```

The browser screenshot shows the URL `54.175.71.197:8080/actuator/health`. A red arrow points to the status "UP".



Considerações finais:

O processo de CI e CD pode variar significativamente dependendo da organização. Este experimento visava criar um ambiente e uma estrutura mais profissionais, porém, ainda há espaço para aprimoramento. Você pode integrar ferramentas de análise de qualidade como SonarQube, executar linters, e até implementar processos de Infraestrutura como Código (IAC). Sinta-se livre para explorar e expandir a partir daqui, e sempre consulte a [loja de actions](#) mencionada ao longo deste material para mais opções.

Se enfrentar algum problema, como um conflito de versão no Docker Hub, [clique aqui](#).

Considere experimentar o processo usando [Watchtower](#), especialmente útil para quem utiliza **Docker Compose**.

Por favor, anexe à sua atividade a URL do seu repositório de testes e uma captura de tela da execução na EC2, incluindo tanto o terminal quanto o navegador. **ENTREGA INDIVIDUAL**.

Um grande abraço!

Agradeço a sua atenção!

Eduardo Verri

eduardo.verri@sptech.school

SÃO
PAULO
TECH
SCHOOL