

EBOOK

GITHUB ACTIONS PARA DEVOPS

Aprendendo a realizar deploys
automatizados seguindo as melhores
práticas de DevOps

FÁBIO BARTOLI



PREFÁCIO

Em um cenário tecnológico em constante transformação, práticas modernas de DevOps e automação confiável são fundamentais para entregar software de forma ágil e com qualidade. O GitHub Actions surge como uma solução robusta de CI/CD integrada ao GitHub, permitindo orquestrar pipelines do build ao deploy com simplicidade e flexibilidade. Este eBook foi criado para orientar profissionais nesse processo, oferecendo um guia técnico e prático que mostra como dominar o uso do GitHub Actions para automatizar entregas com confiança.

O autor, Fábio Miguel Bartoli, é Senior DevOps Engineer e Cloud Engineer com oito anos de experiência prática em projetos que envolvem automação, containers, Kubernetes, multicloud e pipelines CI/CD. Com domínio de ferramentas como Jenkins, GitLab CI e, especialmente, GitHub Actions, ele compartilha aqui sua experiência aplicada para ajudar você a evoluir sua automação e alcançar maturidade nos processos de entrega contínua.



LINKEDIN

linkedin.com/in/fabiobartoli



GITHUB

github.com/FabioBartoli



SUMÁRIO

INTRODUÇÃO

- 6 Automatize Tudo!
- 6 CI/CD na prática
- 6 Segurança é sempre importante
- 7 Onde executar?
- 7 E quanto vai custar essa brincadeira?

CONCEITOS FUNDAMENTAIS

- 9 Workflow
- 10 Triggers
- 11 Jobs
- 12 Steps
- 12 Actions
- 13 Runners

APROFUNDAMENTO NOS WORKFLOWS

- 15 Localização e Organização dos Workflows
- 17 Contextos do GitHub Actions
- 19 Variáveis de Ambiente Padrão
- 21 Expressões no Workflow
- 23 Compartilhamento de Artefatos entre Jobs

INTEGRAÇÃO CONTÍNUA

- 33 Pipeline CI para Node.js
- 35 Pipeline CI para Python
- 37 Integração Contínua para Infraestrutura
- 38 Pipeline de CI com Terraform



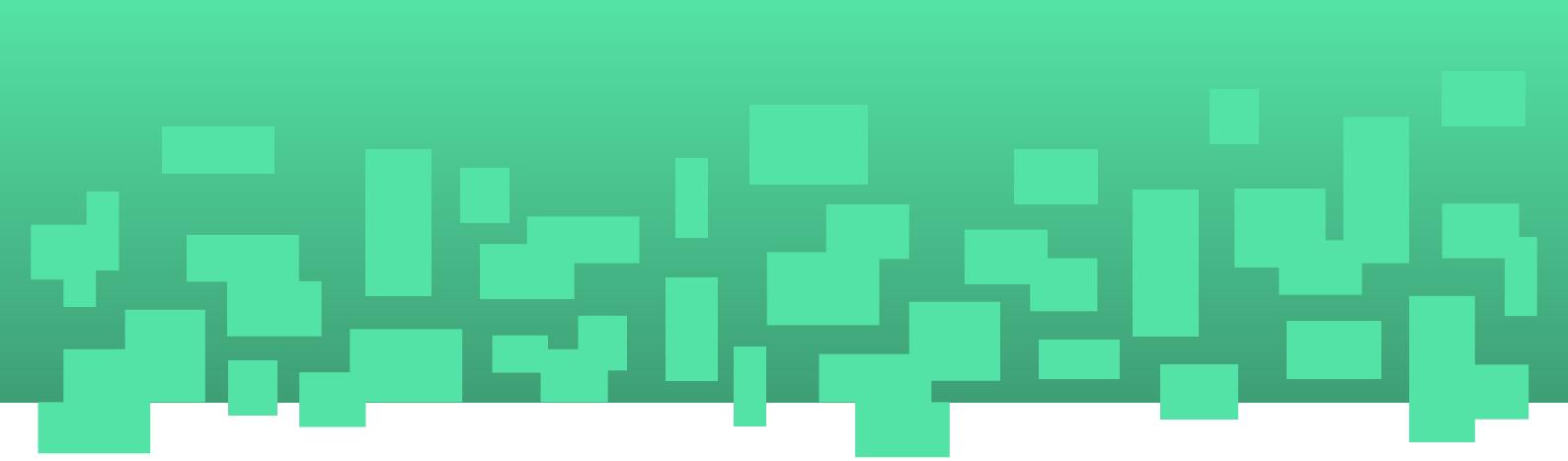
ENTREGA CONTÍNUA

- 42 Controle de Concorrência no Deploy
- 44 Fazendo Deploy no AWS ECS:
- 46 Exemplo de Deploy no Azure App Service
- 47 Exemplo de Deploy no GKE
- 48 Deploy Gates
- 49 Implantação Contínua (Continuous Deployment)

CONCLUSÃO

O QUE VEM A SEGUIR?

AGRADECIMENTOS!



INTRODUÇÃO

Se você chegou até este ebook, então muito provavelmente já está familiarizado com pelo menos alguns cenários de entrega de software, que por vezes acabam sendo bem caóticos. Quem nunca recebeu uma notificação no celular de algum aplicativo de marketplace ou de delivery mandando um “Olá, teste” em produção, não é mesmo? Ou pior ainda, quando aquele site que estamos precisando muito simplesmente está fora do ar com uma mensagem de erro bizarra. No mundo atual, entregar software de maneira ágil já não basta, pois qualquer “release” lançado erroneamente pode causar enormes prejuízos para a empresa. Precisamos garantir a qualidade. Isso significa encurtar cada ciclo de feedback, detectar problemas antes de virarem crises e manter o time focado em criar valor, não em repetir tarefas manuais.

E como transformamos tarefas manuais em processos automatizados? Bom, se você está lendo este ebook, eu não preciso te responder. Usar pipelines automatizados vai garantir que possamos cada vez mais assegurar a qualidade e a segurança do que estamos publicando em nossos ambientes. Cada vez mais a automatização e construção de “esteiras CI/CD” se tornam um pilar de sustentação para o mundo DevOps, e é o que

vai manter a eficiência nos processos de entrega. Nesse contexto, nós conhecemos o nosso querido GitHub Actions, que emerge como uma solução intrínseca e poderosa, redefinindo a maneira como as equipes concebem, constroem, testam e implantam suas aplicações.

Com a vantagem de ser um “add-on” já presente no GitHub, plataforma mais do que consolidada no mercado como uma ferramenta de controle de versão (VCS), ele oferece uma plataforma completa de Integração e Entrega Contínua. Isso significa a capacidade de automatizar uma vasta gama de tarefas ao longo de todo o ciclo de vida do desenvolvimento de software, eliminando a dependência de múltiplas ferramentas externas e simplificando a infraestrutura de automação - e aqui eu estou falando diretamente com você que ainda sofre gerenciando os plugins do Jenkins e que por vez ou outra vê uma ferramenta essencial para seus processos sendo descontinuada. E antes de começarmos a entender como utilizar o Actions, vamos levantar algumas das vantagens de estar fazendo isso.

Automatize Tudo!

USANDO ARQUIVOS YAML no próprio repositório da aplicação nós podemos ter um cenário completo, passando pelo build da aplicação, testes de qualidade de código, empacotamento do container, scan de vulnerabilidades, processos de lint em todos seus arquivos, e até criação de infraestrutura como código – tudo em sequência e sem intervenção manual. Se algo falhar, podemos ser avisados através de automações também.

CI/CD na prática

SE BEM configurados, nossos workflows podem nos prevenir de muitos problemas. Podemos adicionar testes a cada commit, montar um fluxo de promoção de branchs para garantir que só vá para a branch principal do projeto só aquilo que já está 100% funcional e garantir noites de sono mais tranquilas sabendo que ninguém vai conseguir tocar o caos naquela aplicação crítica de madrugada.

Segurança é sempre importante

CONSEGUIMOS, de forma prática, vincular nossos pipelines em diversos serviços de gerenciamento de secrets, como o AWS Secrets Manager e o HashiCorp Vault, além de também contar com uma série de possibilidades nativas da própria ferramenta para manter nossos ambientes mais seguros. Não menos importante, também conseguimos realizar scans e validação de SBOMs de forma facilitada para nossas aplicações, containers e etc.



Onde executar?

AINDA CONTAMOS com a facilidade de ter runners, máquinas virtuais que executam os workflows, hospedados diretamente no GitHub, e que cobrem os principais sistemas que podemos precisar: Linux, Windows e MacOS. Se não quisermos, podemos ainda trigger os actions para serem executados em nossas próprias máquinas, sejam on-premise, máquinas na Cloud e até utilizando o Kubernetes.

E quanto vai custar essa brincadeira?

PARA DESENVOLVEDORES individuais, equipes pequenas e a condução de projetos de prova de conceito, o GitHub Actions se destaca pela sua acessibilidade econômica. Basicamente, o que acontece é o seguinte: Se você estiver usando repositórios públicos e executores auto-hospedados, como em projetos opensource por exemplo, você poderá utilizar o actions a vontade, é grátis! Mas como nem sempre queremos criar um projeto público, também teremos garantido, de maneira gratuita e que se renova todos os meses, 2000 minutos grátis para executar os nossos Actions, o que já é mais que o suficiente na maioria das situações. Só tome cuidado com o seguinte, veja a tabela abaixo:

Sistema Operacional	Fator De Multiplicação	Preço (USD)
Linux	1x	0.008
Windows	2x	0.016
macOS	10x	0.080

Na prática, isso significa que, caso você esteja desenvolvendo algo que necessariamente você precisa testar em máquinas macOS por exemplo, você não terá 2000 minutos/mês, e sim 200 minutos. No caso de usar Windows, seriam 1000 minutos disponíveis. Logicamente você pode usar Linux, Windows e macOS em cenários diferentes e tomando o cuidado para não ultrapassar os 2000 minutos no cálculo final. Por experiência própria, com o Linux nós conseguimos cobrir a grande maioria dos cenários que acabamos desenvolvendo os nossos processos de DevOps, sempre que possível, vá de Linux.

Agora que já entendemos quando e como utilizar o GitHub Actions, está na hora de realmente começarmos a colocar a mão na massa. Através deste ebook, vamos **Descomplicar o GitHub Actions**, fornecendo um guia prático para que você possa aproveitar ao máximo suas capacidades e transformar seus processos de DevOps. Nos próximos capítulos, aprofundaremos os conceitos fundamentais, a criação de workflows, as melhores práticas e exemplos práticos que o ajudarão a implementar automações eficientes e bem seguras.



CONCEITOS FUNDAMENTAIS

VOCÊ VAI perceber que existe uma certa gama de palavras que iremos ouvir e falar o tempo todo quando falamos de GitHub Actions. Neste capítulo, vamos desbrinchar os blocos fundamentais que compõem qualquer automação dentro do GitHub Actions. Ao final, você terá clareza sobre como um evento dispara um workflow, como os jobs se organizam, o que diferencia um job de um step e muito mais. Então, vamos começar do começo:

Workflow

UM **WORKFLOW** é o arquivo YAML que descreve tudo o que sua esteira vai fazer, do primeiro git push até o deploy. Ele fica na pasta `.github/workflows/` do repositório e pode conter um ou vários jobs. Podemos inclusive, ter vários workflows para trabalhos diferentes: um exclusivo para testes em pull requests, outro para criar imagens Docker e um terceiro só para publicar releases, por exemplo. Eles podem ser disparados por eventos, em horários programados ou manualmente.

Por exemplo, esse arquivo “`.github/workflows/hello.yml`” será disparado em nosso Actions depois de qualquer push:



```
name: Hello

on: push

jobs:
  print-ola:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Olá mundo!"
```

Triggers

ELES SERÃO os gatilhos que podem ser utilizados para dar o início a uma execução do actions. Pode ser um evento no seu próprio repositório (tipo alguém fazer um “push” de código, criar uma “release”, ou abrir uma “issue”). Ou pode ser algo que acontece fora do GitHub e que você configura pra avisar ele (usando um **webhook repository_dispatch**). O campo on: define quando o workflow entra em ação. Entre os mais usados, podemos citar:

No exemplo que já criamos, se quiséssemos ter o poder de executar manualmente esse workflow, poderíamos passar, por exemplo:

```
name: Hello

on: workflow_dispatch

jobs:
  print-ola:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Olá mundo!"
```

push

sempre que alguém envia código.

pull_request

a cada atualização de uma PR.

workflow_dispatch

execução manual via botão no GitHub.

schedule

cron jobs (“todo dia às 3 h”).

repository_dispatch

webhooks externos que chamam o pipeline.

Jobs

DENTRO DO seu workflow, você tem um ou mais “jobs” (tarefas maiores). Podemos pensar neles como as grandes etapas a serem executadas dentro de um script. Por exemplo, um job pode ser “Compilar o Código” e outro “Fazer o Deploy”. Eles podem rodar um depois do outro (sequencialmente) ou vários ao mesmo tempo (em paralelo). Cada job roda numa “máquina” separada (o runner) ou dentro de um container. Você define os jobs usando o termo jobs no seu arquivo YAML. Um exemplo de uso de múltiplos jobs seria:

```
name: Compila e Testa

on: push

jobs:
  build-commit:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Compilando o novo commit"

  test-commit:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Testando o novo commit"
```

Nesse exemplo, temos um problema que será explorado mais para frente neste material: o **job de teste**, do jeito que está, não tem acesso ao artefato gerado durante o build da aplicação. Logo veremos como contornar isso.

Steps

SE OS JOBS são as tarefas maiores, os **steps** são tarefas menores que vão executar dentro desses jobs. Um step pode ser rodar um comando (npm install run) ou usar uma “action” pronta. Cada step roda no seu próprio “processo” e compartilham o mesmo runner enquanto fazem parte de um mesmo job, mas rodam em processos separados. Aqui, poderíamos ter algo como:

```
name: Valida qualidade do Container

on: push
jobs:
  test-container:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Scan de Vulnerabilidade no Container"
      - run: echo "Scan de Hadolint no Container"
```

Actions

AS ACTIONS são todos os passos que vão compor o seu workflow. Podemos pensar nelas como **estruturas prontas para rodar uma tarefa específica**, como configurar o ambiente para a sua linguagem de programação ([actions/setup-node](#), [hashicorp/setup-terraform](#)), ou até mesmo autenticar na AWS ([aws-actions/configure-aws-credentials](#)). Em vez de escrever um script gigante pra cada coisa, você usa uma action que já faz isso pra você. **Você pode criar suas próprias actions ou pegar umas prontas no GitHub Marketplace.** Em nosso código, teremos algo assim:

```

jobs:
  list-buckets:
    runs-on: ubuntu-latest
    permissions:
      id-token: write
      contents: read
    steps:
      - uses: aws-actions/configure-aws-credentials@v4
        with:
          role-to-assume:
            arn:aws:iam::123456789012:role/GitHubOIDCRole
          aws-region: us-east-1
      - run: aws s3 ls

```

Runners

NA MINHA opinião, esse aqui é o mais fácil de entender: **Depois que você construiu o seu código do workflow tudo certinho, precisamos colocar isso para ser executado em algum lugar.** Como já comentei no início do ebook, podemos utilizar runners hospedados pela própria GitHub, que são chamados de **GitHub-hosted runners**, ou podemos “plugar” nossas próprias máquinas, o que fica conhecido como **“self-hosted runners”**. Um ponto bem interessante aqui é que as máquinas GitHub-hosted runners são efêmeras, não armazenam dados, e são entregues “zeradas” pra gente no começo de cada execução do Actions.

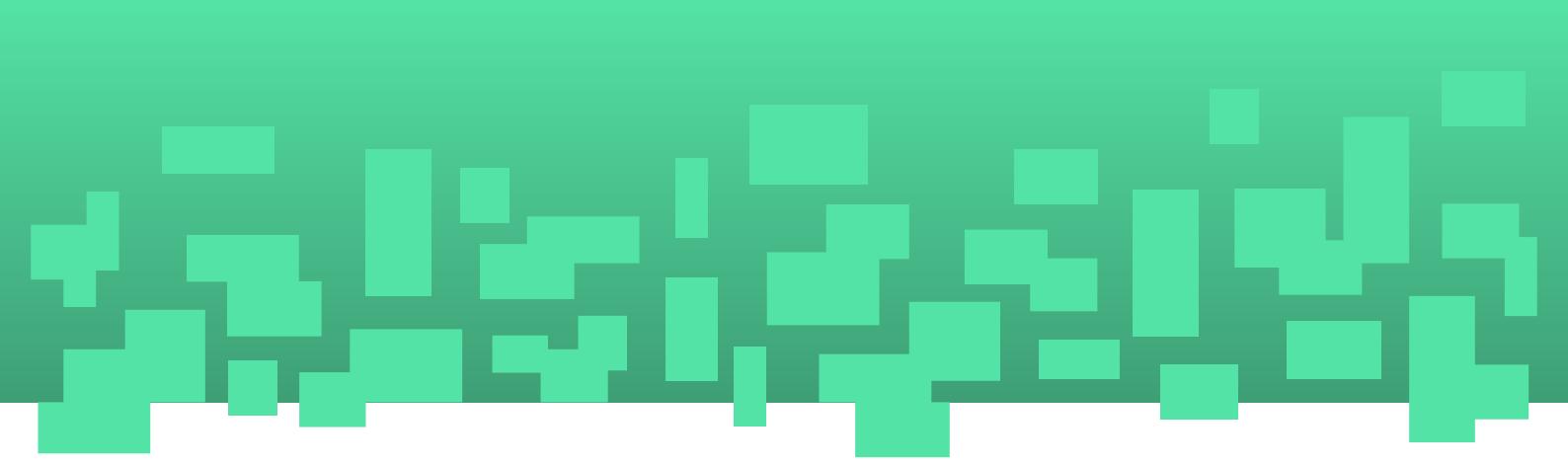
Nos exemplos acima nós já estávamos declarando os runners, pois sem eles nosso código não executa, mas em resumo poderíamos utilizar dentro dos jobs instruções como:



```
jobs:  
  hosted:  
    runs-on: ubuntu-latest      # VM hospedada pelo GitHub  
  
  self:  
    runs-on: self-hosted      # seu runner onpremise
```

A partir da compreensão desses **seis conceitos** já vamos conseguir **entender muito melhor as coisas que acontecem dentro de um pipeline**, e logo você já conseguirá refatorar aqueles actions que estão bem pesados e demorando muito para executar.

Então, vamos continuar!



APROFUNDAMENTO NOS WORKFLOWS

AGORA QUE já entendemos alguns dos componentes básicos essenciais para nosso trabalho, vamos começar a explorar detalhadamente os detalhes do Workflow. Neste capítulo veremos o que acontece dentro dos arquivos **YAML** que criamos dentro de **.github/workflows/**: **como eles são organizados, onde guardar variáveis, como escrever expressões, e de que forma um job passa dados para o outro.**

Localização e Organização dos Workflows

COMO JÁ conversamos, para que o GitHub saiba reconhecer o pipeline que estamos criando, precisamos que ele exista dentro do diretório **.github/workflows/**. Também é importante dizer que seus workflows só ficarão disponíveis para execução desde que eles existam na branch padrão do seu repositório.

Não temos limites de quantidade de arquivos nesse diretório, então separar lógicas distintas em workflows diferentes tende a tornar a manutenção mais simples. O único cuidado que precisamos tomar é não tornar o repositório um monstro

de arquivos e automações. Para isso, veremos como utilizar workflows compartilhados daqui a pouco.

Por definição, o workflow no GitHub Actions é um processo automatizado configurável que executa um ou mais jobs. Dentre os componentes mais básicos de sua composição, destacam-se os eventos, jobs e steps, cuja definições nós já conhecemos.

O nome de um workflow é definido pela chave name no arquivo YAML. O GitHub exibe este nome na aba “Actions” do seu repositório. Se a chave name for omitida, o GitHub utilizará o caminho do arquivo do workflow, relativo à raiz do repositório, como o nome padrão. Por exemplo, se “name” não for especificado, “`.github/workflows/my-workflow.yml`” será o nome padrão.

```
name: Deploy API
on: workflow_dispatch
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Iniciando deploy..."
```

Contextos do GitHub Actions

AGORA, SE TEM algo que eleva a utilização dos workflows a outro nível é a utilização de contextos. Basicamente, contextos são objetos que contêm informações de execução de um workflow. Nós já vimos alguns deles por aqui, como os “**jobs**”, “**steps**” e o **runner**. Você pode acessar esses contextos utilizando a sintaxe de expressão `${{ <context> }}`. Durante nossos exemplos neste ebook, você irá se deparar com diversos contextos sendo utilizados, principalmente os do tipo:

github Contém detalhes do repositório, do evento e do usuário que disparou o fluxo. Propriedades como `github.event_name`, `github.ref` e `github.sha` são utilizadas em diversos projetos que usamos no nosso dia a dia.

env Armazena variáveis de ambiente declaradas no topo do workflow, em um job específico ou em um step. O escopo mais interno sempre sobrepõe o externo.

vars Guarda variáveis de configuração definidas no repositório, organização ou ambiente. É a melhor maneira de trabalharmos com valores estáticos em nossos pipelines, como por exemplo, um Account ID da nossa conta da AWS, Endpoint do nosso cluster HashiCorp, URL de uma API. É bom lembrar que a ideia é guardar valores não sensíveis e que podem ser lidos por quem tiver acesso em nosso projeto. Para os sensíveis, temos o contexto abaixo.

secrets Reúne os valores criptografados em **base64** como tokens e chaves de API. Então, o conteúdo é mascarado nos logs e não pode ser impresso em texto puro.

job, **steps** e **runner** Esses nós já vimos algumas vezes por aqui. Utilizando eles em nossas pipelines, conseguimos por exemplo capturar informações sobre o estado do job atual (**job.status**), sobre outputs de steps anteriores (**steps.<id>.outputs**) e sobre características do host (**runner.os**, **runner.temp**).

Tendo esse conhecimento, podemos criar algumas coisas bem úteis, como um job específico que só executa quando o Actions for "triggado" por um Pull Request:

```
name: Run CI
on: [push, pull_request] # Executa sobre duas condições: Em todos commits ou se um PR for criado
jobs:
  step_default_ci:
    runs-on: ubuntu-latest
    steps:
      - name: Run normal CI
        run: echo "Independente do tipo de trigger, esse step irá ser executado"

  pull_request_ci:
    runs-on: ubuntu-latest
    if: ${{ github.event_name == 'pull_request' }} # Executa apenas quando on = pull_request
    steps:
      - name: Run PR only CI
        run: echo "Para passar nesse step, o trigger do Workflow precisa ser um Pull Request"
```

Variáveis de Ambiente Padrão

O GitHub Actions, além de contextos, utiliza variáveis de ambiente padrão que são automaticamente definidas no ambiente de execução (runner). Algumas dessas variáveis incluem:

- ``GITHUB_TOKEN`` Um token de acesso temporário para autenticação no GitHub em nome do GitHub App instalado no repositório. A sua validade é limitada ao término do job ou a um máximo de 24 horas.
- ``GITHUB_SHA`` O SHA do commit que iniciou o workflow, com seu valor dependendo do evento que o acionou.
- ``GITHUB_REF`` A referência Git completa da branch ou tag que disparou o workflow. Para branches, o formato é ``refs/heads/<nome_do_branch>``.
- ``GITHUB_ACTOR`` O nome de usuário da pessoa ou aplicativo que deu início ao workflow.
- ``GITHUB_WORKFLOW`` O nome do workflow. Caso não seja especificado no arquivo, o valor será o caminho completo do arquivo do workflow no repositório.
- ``GITHUB_WORKSPACE`` O diretório de trabalho padrão no runner para as steps e a localização padrão do repositório ao utilizar a action de checkout.

Se você resolver estudar ainda mais detalhes dos contextos e variáveis, vai acabar percebendo que muitos dos valores que usamos dentro do contexto "github" já existem como variáveis de ambiente padrão. Porém, antes de você desistir do contexto e começar usar só as variáveis em todos os cantos, é importante dizer que as variáveis de ambiente do runner não podem ser usadas em partes do workflow que são processadas pelo GitHub Actions antes de serem enviadas para o runner, como as condições "if". Nesses casos, você deve usar os contextos diretamente.

```
name: Acessando variáveis
on: [push, pull_request]
env:
  GLOBAL_VAR: production
jobs:
  show-info:
    runs-on: ubuntu-latest
    env:
      JOB_VAR: giropops-strigus
    steps:
      - run: |
          echo "Evento: ${{ github.event_name }}"
          echo "SHA: ${{ github.sha }}"
          echo "SO do Runner: ${{ runner.os }}"
          echo "Global: $GLOBAL_VAR / Job: $JOB_VAR"
```

Expressões no Workflow

OUTRA FUNÇÃO que utilizaremos muito nos nossos Actions são as expressões. Elas permitem que você acesse contextos, variáveis e funções para avaliar informações dentro de um workflow (assim como o “if” que eu acabei de citar). **Elas são usadas para criar condições que determinam se um job ou step deve ser executado, ou para definir valores dinâmicos.** A sintaxe para expressões é `${{ <expression> }}`.

```
name: Valida Pull Requests
on: pull_request

jobs:
  only-tux:
    # Executa apenas se o autor do PR for o nosso querido
    pinguim-toskao
    if: ${{ github.event.pull_request.user.login ==

'pinguim-toskao' }}
    runs-on: ubuntu-latest
    steps:
      - run: echo "Executando porque o PR foi aberto por
        pinguim-toskao.. É melhor revisar!"
```

Dentro de steps, a palavra-chave `run:` executa qualquer comando disponível no shell do runner. Você pode mudar o diretório padrão com `working-directory`, escolher outro interpretador (`shell: bash`, `shell: pwsh`) e até construir blocos multilinha usando “|” (pipe). Quando formos criar algum script muito longo ou que dependam de condicionais, laços e etc, é bem mais interessante usarmos arquivos ao invés de colocar os comandos diretos no Workflow, pois isso evita deixar nosso código muito poluído.

```
name: Valida Pull Requests
on: pull_request

jobs:
  job-executor-scripts:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: ./scripts # Define o diretório de trabalho padrão para os steps desse job
    steps:
      - name: Checkout do Repo
        uses: actions/checkout@v4
      - name: Transformando os nossos scripts em executáveis
        run: chmod +x giropops.sh strigus.sh
      - name: Executando
        run: | # Os scripts serão executados a partir de
'./scripts'
          ./giropops.sh
          ./strigus.sh
```

Compartilhamento de Artefatos entre Jobs

PARA FECHARMOS essa parte de estrutura do Workflow, vamos entender o problema que eu citei ao dar este exemplo:

```
name: Compila e Testa

on: push

jobs:
  build-commit:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Compilando o novo commit"

  test-commit:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Testando o novo commit"
```

Quando executamos um código com múltiplos **jobs**, estamos executando tarefas em máquinas diferentes. Portanto, um build da aplicação em uma máquina e os testes em outra é algo que não faz muito sentido, não é mesmo? Nesse caso, seria mais interessante que trabalhássemos com steps diferentes, que rodam um após o outro. Como steps rodam na mesma máquina, aí sim não teríamos nenhum problema de execução.

```
name: Compila e Testa

on: push

jobs:
  build-and-test-commit:
    runs-on: ubuntu-latest
    steps:
      - name: Build da aplicação
        run: echo "Compilando o novo commit"

      - name: Testes na aplicação buildada
        run: echo "Testando o novo commit"
```

Mas e se eu necessariamente preciso que o meu build aconteça em um ambiente e meu teste em outro? Quando a compilação e a bateria de testes exigem ambientes distintos, como por exemplo, compilar em Linux, mas testar em Windows ou ter um job responsável somente pelo build e outro para o deploy, basta colocar **múltiplos jobs** no mesmo workflow e usar duas actions oficiais para levar os artefatos de um job ao outro:

actions/upload-artifact roda no job de build, empacota tudo que você precisa (binário, pacote, relatório, etc.) e envia para o armazenamento temporário do GitHub Actions.

actions/download-artifact roda nos jobs seguintes, recupera o pacote e coloca no sistema de arquivos do runner onde os testes serão executados.

```

name: Build and Test em 2 SO distintos

on:
  push

jobs:
  build:
    runs-on: ubuntu-latest
    outputs:
      artifact-name: build-output
    steps:
      - uses: actions/checkout@v4

      - name: Compile project
        run: |
          make build

      - name: Upload compiled artifact
        uses: actions/upload-artifact@v4
        with:
          name: build-output           # identifica o pacote
          path: dist/                  # conteúdo a armazenar

  test-windows:
    needs: build           # só começa depois que build terminar
    runs-on: windows-latest
    steps:
      - uses: actions/checkout@v4

      - name: Download artifact
        uses: actions/download-artifact@v4
        with:
          name: build-output       # mesmo nome usado no upload
          path: .\artifact         # pasta destino no runner

      - name: Run tests on Windows
        run: .\artifact\app.exe --test

```

Nesse exemplo:

1. O job “build” compila no runner Linux, gera os arquivos em dist/ e faz o upload com upload-artifact.
2. O GitHub armazena o pacote internamente em seu storage, sem necessidade de um bucket ou nada nesse sentido.
3. O job “test-windows” faz o download do mesmo artefato com download-artifact no runner Windows, garantindo que os testes usem exatamente o binário produzido no passo anterior.
4. Se o build falhar, os jobs dependentes nem começam, preservando tempo de runner.

Workflows Reutilizáveis

JÁ QUE começamos a falar das actions oficiais, vamos entender mais no detalhe sobre o que eles são. Workflows reutilizáveis permitem empacotar um pipeline inteiro (vários jobs, steps, segredos, expressões) em um único arquivo YAML e chamá-lo de outros repositórios ou do mesmo repositório. A chamada é feita pela palavra-chave **uses**, seguida do caminho do arquivo e de uma referência git (branch, tag ou SHA) com o operador @:

```
uses: owner/repo/.github/workflows/<workflow>.yml@<ref>
```

Se o workflow estiver no mesmo repositório, basta usar o caminho relativo:

```
uses: ./github/workflows/reusable.yml
```

Os workflows reutilizáveis podem ser do seu próprio projeto, existindo como arquivos **YAML** na pasta `.github/workflows`, ou ainda podem ser de outros repositórios internos da sua organização (no caso de organizações Enterprise no Github), ou de projetos públicos da comunidade.

Workflows reutilizáveis podem receber inputs e secrets de um workflow “chamador”, e podem expor outputs para que o workflow que o chamou os utilize. No workflow de origem, você passa os inputs usando a chave “**with**” e os secrets usando a chave “**secrets**” dentro do job que chama o workflow reutilizável. O tipo de dado do input deve corresponder ao tipo especificado no workflow chamado

No workflow de destino os parâmetros são declarados em `on.workflow_call.inputs` e `workflow_call.secrets` (olha ai os contextos novamente) com três atributos principais:

```
type [string|number|boolean]  
required [true|false]  
default (valor opcional para  
inputs não obrigatórios)
```

```
with:           # para inputs  
<input>: valor  
  
secrets:       # para segredos  
<secret>: ${{ secrets.MEU_TOKEN }}
```

A seguir, vou deixar um exemplo do seguinte:

O primeiro workflow é o que será chamado, percebam que seu trigger é um “**workflow_call**”;

Esse workflow receberá diversos inputs que o permitirão fazer o build, teste e push de uma aplicação para o DockerHub;

Esse workflow reutilizável irá realizar um output que nos informa qual é a URL onde a imagem foi publicada.

```

name: Build-Scan-Push

on:
  workflow_call:
    inputs:
      image-name:
        description: "Nome da imagem (sem usuário)"
        type: string
        required: true
      context-path:
        description: "Diretório do Dockerfile / build context"
        type: string
        default: .
      dockerhub-username:
        description: "Namespace no Docker Hub"
        type: string
        required: true
      tag:
        description: "Tag (padrão = SHA curto do commit)"
        type: string
        default: ${{ github.sha }}
      secrets:
        DOCKERHUB_TOKEN:
          description: "Token do Docker Hub"
          required: true
    outputs:
      image-url:
        description: "URL da imagem publicada"
        value: ${{ jobs.release.outputs.pushed }}

jobs:
  release:
    runs-on: ubuntu-latest
    env:
      IMAGE: ${{ inputs.dockerhub-username }}/${{ inputs.image-name }}:${{ inputs.tag }}
    outputs:
      pushed: ${{ steps.export.outputs.ref }}

```

```

steps:
  - uses: actions/checkout@v4

  - name: Login no Docker Hub
    uses: docker/login-action@v3
    with:
      username: ${{ inputs.dockerhub-username }}
      password: ${{ secrets.DOCKERHUB_TOKEN }}

  - name: Build
    id: build
    uses: docker/build-push-action@v5
    with:
      context: ${{ inputs.context-path }}
      tags: ${{ env.IMAGE }}

  - name: Scan com Docker Scout
    uses: docker/scout-action@v1
    with:
      command: cves
      image: ${{ env.IMAGE }}
      only-severity: critical
      exit-code: true # cancela pipeline se achar CVE crítico

  - name: Push no Docker Hub
    run: docker push $IMAGE # só chega aqui se o scan não
    falhar

  - name: Exporta imagem publicada
    id: export
    run: echo "ref=${IMAGE}" >> "$GITHUB_OUTPUT"

```

E o workflow que vai chamar esse cara fica:

```
name: Release App
on:
  workflow_dispatch:
    inputs:
      image-tag:
        description: "Tag personalizada (opcional)"
        required: false
        type: string

jobs:
  publish:
    uses:
      fabiobartoli/docker-scan/.github/workflows/docker-build-scan-push.yml@v1.0.0 # simulando um repo público e acessando uma versão específica do workflow

    with:
      image-name: api-gateway # obrigatório
      context-path: ./docker
      dockerhub-username: fabiobartoli
      tag: ${{ inputs.image-tag || github.sha }}

    secrets:
      DOCKERHUB_TOKEN: ${{ secrets.DOCKERHUB_TOKEN }}

  announce: # Para chamar o output
    needs: publish
    runs-on: ubuntu-latest
    steps:
      - run: echo "Imagen publicada en: ${{ needs.publish.outputs.image-url }}"
```

Perceba que, desta maneira, o nosso código do workflow chamador pode ser muito mais curto e genérico, e podemos utilizar quantos workflows chamadores diferentes quisermos apontando para o mesmo workflow reutilizável. Com esse arranjo, seus repositórios só precisam saber o nome da imagem e o token do Docker Hub. Todo o resto (build, scan de segurança e push) permanece centralizado, versionado e fácil de dar manutenção.

Então vamos lá, agora já entendemos sobre a lógica de contextos, a troca de dados entre jobs e sobre a maravilha que é transformar pipelines inteiros em workflows reutilizáveis. A essa altura, você começará a tratar qualquer fluxo de CI/CD como um conjunto de peças modulares: gatilhos bem escolhidos, variáveis declaradas no lugar certo, expressões que evitam copy-paste e artefatos que podemos enviar de um job ao outro.

Nos próximos capítulos, vamos começar a ver mais exemplos do mundo real, de como os processos DevOps podem ficar muito melhores e fáceis de gerenciar com a utilização de todo o poder e ferramental que a **construção de pipelines** nos fornece. Vamos lá!



INTEGRAÇÃO CONTÍNUA

A PARTIR DE AGORA, nós começamos a estruturar o nosso pipeline para produção. Já passamos pelos principais conceitos dentro do Actions e temos base para criar as automações que serão necessárias. Eu tenho certeza que você já leu dezenas de documentações que explicam o que é a tal da integração contínua, certo? Mas relaxa que aqui vai ser de forma prática. Quando falamos de integração contínua, ou o famoso CI - Continuous Integration, estamos falando sobre uma prática onde os desenvolvedores integram as mudanças de código em um repositório central várias vezes ao dia, de forma frequente. Cada vez que uma mudança é feita (um push com alterações, por exemplo), um processo automático de "construção" e "teste" é acionado.

A integração contínua nos ajudará a garantir a qualidade e confiança do nosso código, uma vez que ninguém conseguirá compilar códigos com erros ou inseguros em nossos ambientes, a depender das validações realizadas. E como os erros são identificados rapidamente, sua tratativa também é facilitada. O comportamento principal que esperamos de qualquer rotina de Integração Contínua é a capacidade de compilar o projeto e validar seu comportamento, automaticamente, a cada envio de código. Por



padrão, o GitHub já nos oferece um catálogo de Actions para as linguagens mais populares, que podemos simplesmente consumir. Basta você clicar na opção de “actions” do seu repositório enquanto ainda não tiver criado nenhum, e serão exibidas sugestões baseadas na linguagem identificada.

The screenshot shows the GitHub Actions interface. At the top, there's a navigation bar with links for Code, Issues, Pull requests, Actions (which is highlighted with a red arrow), Projects, Security, Insights, and Settings. Below the navigation is a section titled "Get started with GitHub Actions" with the sub-instruction "Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started." A "Skip this and set up a workflow yourself →" link is also present. A search bar labeled "Search workflows" is available. The main area is titled "Suggested for this repository" and displays six workflow cards:

- Publish Node.js Package to GitHub Packages** By GitHub Actions: Publishes a Node.js package to GitHub Packages. Includes "Configure" and "JavaScript" buttons.
- Publish Node.js Package** By GitHub Actions: Publishes a Node.js package to npm. Includes "Configure" and "JavaScript" buttons.
- Grunt** By GitHub Actions: Build a NodeJS project with npm and grunt. Includes "Configure" and "JavaScript" buttons.
- Gulp** By GitHub Actions: Build a NodeJS project with npm and gulp. Includes "Configure" and "JavaScript" buttons.
- Webpack** By GitHub Actions: Build a NodeJS project with npm and webpack. Includes "Configure" and "JavaScript" buttons.
- Deno** By GitHub Actions: Test your Deno project. Includes "Configure" and "JavaScript" buttons.

Pipeline CI para Node.js

Se procurarmos por algum Action que tenha testes, poderemos encontrar esse do próprio GitHub Actions. Vamos clicar em “Configure”:

The screenshot shows the GitHub Actions search results for the term "test". The search bar at the top has "test" typed into it. On the left, there's a sidebar with "Categories" including Deployment, Continuous integration, Automation, and Pages. The main area shows a search result for "Found 34 workflows". The first result is highlighted with a red box and is titled "Node.js" by GitHub Actions. It describes building and testing a Node.js project with npm. It includes "Configure" and "JavaScript" buttons. The entire "Node.js" card is also highlighted with a red box.

Com isso, o GitHub abrirá a sugestão de criação de um workflow dentro do nosso “.github/workflows”, mesmo que a gente ainda não tenha criado ele. Nesse caso do Node, a sugestão de código é:

```
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
```

A screenshot of the GitHub Actions workflow editor. The URL in the address bar is `github-actions-linuxtis / .github / workflows / node.js.yml`. The page shows two tabs: "Edit" and "Preview". The code editor contains the following YAML:

```

1  # This workflow will do a clean installation of node dependencies, cache/restore them, build the source
2  # For more information see: https://docs.github.com/en/actions/automating-builds-and-tests/building-and-
3
4  name: Node.js CI
5
6  on:
7    push:

```

Normalmente, vamos usar esse contexto de “strategy: matrix” para utilizar dois ou mais recursos simultaneamente. No caso desse exemplo, fazemos isso para garantir a compatibilidade com as LTS atuais sem duplicar YAML.

Pipeline CI para Python

Para projetos Python, você também pode criar um workflow de CI para construir e testar. Assim como no Node.js, os runners do GitHub já vêm com Python pré-instalado.

A screenshot of the GitHub Actions “Get started with GitHub Actions” page. The top navigation bar includes “quests”, “Actions” (which is underlined), “Projects”, “Security”, “Insights”, and “Settings”. The main heading is “Get started with GitHub Actions”. Below it, a sub-headline says “Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.” A link “Skip this and set up a workflow yourself →” is present. On the left, there is a sidebar with “Categories” including “Deployment”, “Continuous integration”, “Automation”, and “Pages”. A search bar shows “python test”. The main area displays “Found 4 workflows” with the following cards:

- Django** By GitHub Actions: Build and Test a Django Project. Includes “Configure” and “Python” buttons.
- Python application** By GitHub Actions: Create and test a Python application. Includes “Configure” and “Python” buttons.
- Python Package using Anaconda** By GitHub Actions: Create and test a Python package on multiple Python versions using Anaconda for package management. Includes “Configure” and “Python” buttons.
- Python package** By GitHub Actions: Create and test a Python package on multiple Python versions. This card is highlighted with a red border.

```
name: Python package

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

jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        python-version: ["3.9", "3.10", "3.11"]

    steps:
      - uses: actions/checkout@v4
      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v3
        with:
          python-version: ${{ matrix.python-version }}
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          python -m pip install flake8 pytest
          if [ -f requirements.txt ]; then pip install -r requirements.txt; fi
      - name: Lint with flake8
        run: |
          # stop the build if there are Python syntax errors or undefined names
          flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
          # exit-zero treats all errors as warnings. The GitHub editor is 127 chars wide
          flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
      - name: Test with pytest
        run: |
          pytest
```

The screenshot shows a GitHub Actions workflow configuration in a YAML file named `python-package.yml`. The file contains the following code:

```
9   pull_request:
10     branches: [ "main" ]
11
12   jobs:
```

Como você pode perceber, conseguimos usar os workflows para testes e assim criar um verdadeiro “pente fino” do que os desenvolvedores fizeram push e se realmente funciona do jeito que eles esperam. Os testes nos ajudam a evitar o famoso “na minha máquina funciona”, pois eles precisarão adaptar o código para o funcionamento mais próximo possível do ambiente real. Se não, os testes não passam.

Uma boa prática também é sempre separar as etapas de build e testes em jobs separados, assim como nos exemplos acima, ou até em workflows diferentes. Basta levarmos nossos artefatos de um lado para o outro e realizar os testes necessários.

Integração Contínua para Infraestrutura

NO MUNDO DevOps, é muito comum utilizarmos esses processos de **Integração Contínua** para garantir a qualidade da nossa Infraestrutura como Código. Quando utilizamos IaC, estamos utilizando uma linguagem declarativa de todos nossos recursos (VPCs, grupos de segurança, máquinas virtuais, bancos de dados) através de arquivos declarativos versionados no GitHub.

Ao utilizarmos o Terraform, os recursos são escritos em HCL e se convertem em um “state”, uma forma de manter um “inventário” de seus recursos criados e todas as configurações. Quando unimos toda a potência de uma ferramenta como o Terraform aos conceitos de versionamento e pipelines, teremos a nível de

infraestrutura todas as vantagens que estávamos comentando que acontecem para as aplicações: **revisões rigorosas via Pull Requests, possibilidades de manutenção ágeis, rastreabilidade de problemas e etc.**

Vou deixar o link para o treinamento **Terraform Essentials**, disponível gratuitamente na plataforma de LINUXTips e ministrado pelo grande mestre **Rafael Gomex**. Caso você não tenha muito conhecimento do assunto, vale muito a pena ver o curso para acompanhar o conteúdo a seguir

CLIQUE AQUI para acessar o treinamento **Terraform Essentials** da LINUXtips
<https://linuxtips.io/treinamento/terraform-essentials/>

Vamos utilizar os nossos Workflows para automatizar a gestão da nossa infraestrutura através da combinação do Actions + Terraform.

Podemos utilizar os seguintes steps para validar a qualidade, segurança e validade dos nossos códigos:

terraform fmt

Padronizar estilo, detectar erros triviais.

terraform validate

Verificar coerência e sintaxe HCL.

terraform plan -out tfplan

Simular mudanças contra state remoto, gerar comparações de diferenciais.

Nós também podemos aumentar ainda mais a nossa cobertura de testes de QA e Security utilizando ferramentas externas como o **tfsec** (ferramenta de static analysis para detectar falhas de segurança), **terratest** (framework em Go para testes automatizados de infraestrutura) e etc.

Pipeline de CI com Terraform

VEJA O exemplo de uma implementação de qualidade e segurança de código que podemos fazer para nossas pipelines Terraform. Esse exemplo é criado para executar sempre que pull requests forem abertos com modificações em arquivos ".tf" que fiquem dentro da pasta.



```
on:
  pull_request:
    paths: ["**/*.tf"]
  push:
    branches: [main]

jobs:
  terraform:
    runs-on: ubuntu-latest
    permissions:
      contents: read
    steps:
      - uses: actions/checkout@v4

      # Instala a versão 1.8.2 do Terraform
      - uses: hashicorp/setup-terraform@v3
        with:
          terraform_version: "1.8.2"

      # Faz o format dos nossos arquivos
      - name: Terraform fmt
        run: terraform fmt -check -recursive

      # Inicialização de plugins/back-end Local
      - name: Terraform init
        run: terraform -chdir=iac init -backend=false

      # Executa o validate da config
      - name: Terraform validate
        run: terraform -chdir=iac validate -no-color

      # Gera o plan
      - name: Terraform plan
        run: terraform -chdir=iac plan -no-color -out=tfplan

      # Faz o upload do plan para podemos baixar e consultar
      - uses: actions/upload-artifact@v4
        with:
          name: tfplan
          path: tfplan
```

```

security: # Aqui começamos outro job
  needs: terraform # Só roda se fmt/validate/plan passaram
  runs-on: ubuntu-latest
  steps:
    - name: Clone repo
      uses: actions/checkout@v4

# Análise de segurança com a action da AquaSecurity
- name: tfsec static analysis
  uses: aquasecurity/tfsec-action@v1.0.0
  with:
    additional_args: --minimum-severity CRITICAL,HIGH
    format: sarif
  # Nesse exemplo, vou passar para falhar com vulnerabilidades
  # do tipo Critical ou High.

terratest: # E aqui, um terceiro job
  needs: terraform
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4

# Executa Terratest via action
- name: Run Terratest
  uses: claudposse/github-action-terratest@main
  with:
    sourceDir: test

```

Poderíamos gastar páginas e páginas deste ebook com exemplos de diferentes ferramentas, linguagens e até mesmo tipos de teste diferentes que conseguimos criar via workflows, mas o resumo é que a utilização de pipelines é o que nos dará uma camada a mais de segurança sobre as nossas aplicações. Utilizar a integração contínua nos permitirá realizar rollouts ágeis da nossa aplicação, mas sem abrir mão de componentes essenciais, como a qualidade do código.

Ao longo do próximo capítulo vamos ver no detalhe o que podemos fazer com esta aplicação, agora que já está “buildada” e com todos os testes realizados.

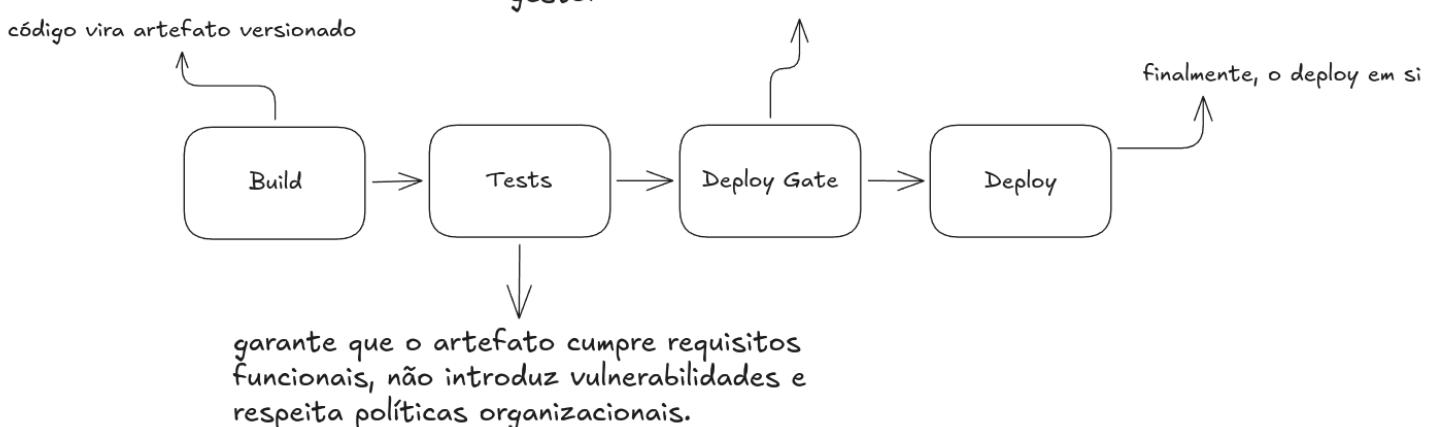
ENTREGA CONTÍNUA

AGORA QUE nosso código já está perfeitamente testado, lint executado, qualidade garantida, segurança garantida e de perfume passado, tá na hora de colocá-lo para executar em nossos ambientes.

A **Entrega Contínua** é justamente o mecanismo que transforma um commit saudável em versão disponível, repetindo o processo quantas vezes forem necessárias e do mesmo jeito. É a garantia de que, a qualquer momento, podemos liberar novas funcionalidades ou correções para nossos clientes com confiança e agilidade. Entrega Contínua (Continuous Delivery) é a prática de manter o artefato de software em estado “pronto para produção” depois de cada passagem pelo pipeline. O ciclo sempre será algo como:

podemos implementar regras como:
só acontece deploy em prd se passar antes em
stg

só pode deploy em prd com aprovação do
gestor



Com esse modelo, o passo de “fazer deploy” não fica mais tão moroso e passa a ser uma decisão de negócio: o time escolhe quando e onde acionar a próxima versão, sabendo que o build mais recente já foi qualificado tecnicamente.

Os workflows de CD vão ditar como nossas aplicações vão ser implementadas nos ambientes. Podemos implementar a nossa entrega como jobs a mais dentro do mesmo workflow onde a integração acontece, ou criar novos workflows para serem chamados assim que necessários.

Controle de Concorrência no Deploy

QUANDO FALAMOS especialmente de ambientes de produção, é crucial evitar que múltiplas implantações ocorram ao mesmo tempo para o mesmo ambiente. Imagine a maravilha que vai virar se dois desenvolvedores fizerem deploy de versões diferentes simultaneamente... O GitHub Actions oferece o controle de concorrência usando a palavra-chave **“concurrency”**.

Você pode definir um grupo de concorrência para um job ou workflow inteiro. Se uma nova execução for acionada enquanto outra do mesmo grupo está em andamento, a nova pode ser cancelada ou colocada em fila, dependendo da sua configuração. Por exemplo:

```
on:
  push:
    branches:
      - main

jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production
    concurrency: production_deployment
    steps:
      - name: Checkout do código
        uses: actions/checkout@v4
      - name: Simular deploy
        run:
          echo "Iniciando deploy na produção..."
          sleep 60 # Simulando um deploy que demore 60 segundos
          echo "Deploy na produção concluído."
```

Caso um novo push para main ocorrer enquanto um deploy de produção já estiver em andamento, a nova execução ficará esperando ou será cancelada, dependendo da configuração padrão ou da configuração que fizemos dentro do concurrency. Isso vai ajudar muito a evitar conflitos e garantir a estabilidade do ambiente.

E além de garantir essas questões de concorrência e toda a conversa sobre testes que tivemos no último capítulo, também é recomendável manter a máxima paridade entre ambientes: a infraestrutura e as configurações do ambiente de teste devem espelhar as de produção na medida do possível. Assim, o que foi validado em dev/staging terá o mesmo comportamento quando implantado para os usuários finais, reduzindo surpresas em tempo de execução.

Outro ponto que é de suma importância quando falamos sobre publicar as nossas aplicações em ambientes é o cuidado que devemos ter com secrets, como tokens, chave de API, credenciais IAM, dentre outras. No GitHub, nós conseguimos utilizar Actions de providers externos para cuidar destas credenciais, como integrações com o HashiCorp Vault ou algum Secrets Manager, como também podemos utilizar esses valores como secrets de repositório/ organização. A recomendação principal é, sempre que possível, utilizar gerenciadores externos. Não que os secrets do GitHub não sejam seguros, mas dentro de um cofre externo conseguimos ter muito mais controle de acesso, protocolos avançados de criptografia, auditoria, autenticação multifator, Single Sign-On e etc, enquanto no GitHub estamos apenas encriptando em Base64.

Por fim, como estamos tratando de ambientes produtivos, ainda é prudente ter mecanismos de mitigação de risco. Técnicas como deploy blue/green ou canary deploy podem ser incorporadas ao fluxo de CD para liberar novas versões gradualmente, monitorando métricas e permitindo rollback rápido se algo inesperado ocorrer. Não irei me aprofundar nesse tema das estratégias de deploy por aqui, mas tenha em mente que a entrega contínua eficaz não se trata apenas de "mandar código para produção", mas de fazê-lo de modo confiável e recuperável. Agora vamos ver tudo isso aqui na prática. Para exemplificar, eu montei 3 cenários de entrega contínua na AWS, Azure e Google Cloud. Vamos lá!

Fazendo Deploy no AWS ECS:

Vamos considerar uma aplicação containerizada sendo implantada no Elastic Container Service (ECS). Suponha que já tenhamos configurado um pipeline de CI que realiza o build e os testes da aplicação e outro responsável por realizar o provisionamento da infraestrutura. Agora, no estágio de CD, adicionaremos passos para empacotar a aplicação em uma imagem Docker e fazer o deploy no ECS automaticamente.

```
name: Deploy Prod - ECS

on:
  workflow_run:
    workflows: ["QA Tests", "Security Tests"]
    types: [completed]

concurrency:
  group: deploy-ecs-${{ github.event.workflow_run.head_branch }}
  cancel-in-progress: true # Se forem feitos 2 deploys
  concurrentes, cancela a execução que ainda estiver em progresso

jobs:
  deploy_to_ecs:
    deploy_to_ecs:
      # Só continua se o gatilho foi "completed" com sucesso
      if: ${{ github.event.workflow_run.conclusion == 'success' }}
      runs-on: ubuntu-latest
      environment: production
      steps:
        - uses: actions/checkout@v4

        - name: Autenticar na AWS
          uses: aws-actions/configure-aws-credentials@v2
          with:
            aws-region: us-east-1
            role-to-assume: ${{ secrets.AWS_OIDC_ROLE_ARN }}
```

```

- name: Login no ECR
  uses: aws-actions/amazon-ecr-login@v1

- name: Build da imagem Docker
  run: docker build -t ${{ env.ECR_URI
}}/giropops-strigus:${{ github.sha }} .

- name: Push da imagem para ECR
  run: docker push ${{ env.ECR_URI }}/giropops-strigus:${{ github.sha }}

- name: Deploy no ECS
  uses: aws-actions/amazon-ecs-deploy-task-definition@v1
  with:
    task-definition: path/to/task-def.json
    cluster: giropops-cluster
    service: gurus-service
    image: ${{ env.ECR_URI }}/giropops-strigus:${{ github.sha }}

```

O deploy só dispara quando os dois workflows externos terminam com sucesso:

“QA Tests” - Validando o funcionamento, integrações.

“Security Tests” - varredura de vulnerabilidades.

O bloco **on.workflow_run** escuta a conclusão desses workflows e, no job, a cláusula **if:** garante que apenas execuções com **conclusion == ‘success’** prosseguem.

Exemplo de Deploy no Azure App Service

Sabemos que no dia a dia nem todas as aplicações que trabalhamos são containerizadas, muitas vezes lidamos com aplicativos web tradicionais entregues como um pacote (ZIP, TAR, etc.).

Nesse próximo exemplo, vamos imaginar que temos um pipeline de entrega contínua para o **Azure App Service**, assumindo que o artefato (build) já foi gerado na fase de CI e armazenado para uso com o nome de "**azure-app**". Nesse caso, nosso job de deploy seria algo como:

```
jobs:  
  deploy_to_azure:  
    runs-on: ubuntu-latest  
    environment: production  
    steps:  
      - uses: actions/checkout@v4  
  
      - name: Baixar artefato de build  
        uses: actions/download-artifact@v3  
        with:  
          name: azure-app  
  
      - name: Deploy no Azure App Service  
        uses: azure/webapps-deploy@v2  
        with:  
          app-name: GiropopsLegado  
          publish-profile: ${{  
            secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}  
          package: ./azure-app.zip
```

Assim que um outro workflow rodar e criar o artefato, podemos então executar este job. Como conversamos, podemos fazer isso de forma manual ou automática, depende muito das regras de negócio da organização. A aplicação será enviada ao Azure App Service com zero intervenção humana, atualizando a versão em produção.

Exemplo de Deploy no GKE

PARA COBRIR mais um cenário, vamos para o Google Cloud – suponha que temos um cluster Kubernetes no GKE onde nossa aplicação deve rodar. Assim como no caso anterior, vamos imaginar também que a imagem já está disponível em algum Container Registry que temos acesso. Nosso foco na etapa de CD será pegar essa imagem e atualizar a aplicação no Kubernetes. O nosso Actions será algo como:

```
jobs:
  deploy_to_gke:
    runs-on: ubuntu-latest
    environment: production
    steps:
      - uses: actions/checkout@v4

      - name: Autenticar no GCP
        uses: google-github-actions/setup-gcloud@v1
        with:
          service_account_key: ${{ secrets.GCP_SA_KEY }}
          project_id: strigus-girus-id
          export_default_credentials: true

      - name: Obter credenciais do GKE
        run: gcloud container clusters get-credentials meu-cluster
--zone us-central1-a

      - name: Atualizar imagem no Deployment
        run: kubectl set image deployment/minha-app
        container-principal=gcr.io/gioproject/strigus-girus:${
          github.sha }
```

Aqui o workflow de CD faz a autenticação no GCP, configura acesso ao cluster e realiza o rollout da nova imagem. Após isso, o Kubernetes inicia novos pods com a versão mais recente da aplicação.

Nesse caso do deploy do Kubernetes e em muitos outros, nós temos ferramentas mais interessantes para realizar a parte de entrega contínua, como o ArgoCD, por exemplo. Nele nós já teríamos uma entrega onde conseguimos criar estratégias de rollout específicas para cada cenário, além de acompanhando e validação da saúde do ambiente de forma mais dinâmica. O objetivo aqui é mais sabermos que, caso necessário, também conseguimos fazer diretamente pelo Actions, sem nenhum componente externo.

Deploy Gates

PARA DEIXAR os nossos deploys, principalmente os produtivos, ainda mais seguros, podemos utilizar validações para a realização dos deploys, através do que chamamos de deploy gate. Em essência, um deploy gate é qualquer mecanismo que condiciona ou atrasa a implantação em produção até que certos critérios sejam atendidos. Isso significa que podemos negar uma publicação de nova versão até que ela passe por etapas obrigatórias, como por exemplo, primeira ser publicada em desenvolvimento, depois staging e só depois poder ir para produção.

Também podemos incluir requisitos como aprovação manual de um líder técnico, de forma que qualquer job direcionado a ele entre em modo de espera até que alguém aprove manualmente no GitHub, ou conclusão bem-sucedida de todos os testes de QA e segurança.

Do ponto de vista técnico, há várias maneiras de implementar gates. Algumas organizações adotam uma estratégia baseada em branches: por exemplo, a branch develop dispara deploy em dev, a branch staging dispara deploy em homologação, e só merges aprovados na branch main vão para produção. Nesse caso, o próprio fluxo de Git Flow com revisão humana nos merges atua como um gate para produção.

Como vimos no exemplo do Docker + ECS, uma forma de implementar gates automáticos entre ambientes é encadear workflows usando o evento `workflow_run`. Com esse recurso, podemos dividir nosso pipeline de CI/CD em múltiplos arquivos YAML e fazer com que o disparo de um dependa da conclusão bem-sucedida do anterior.

Implantação Contínua (Continuous Deployment)

TAMBÉM É importante comentar que, indo no **caminho totalmente oposto aos deploys gates**, nós temos uma segunda abordagem quando falamos do “Continuous Delivery”, que é a **Implantação Contínua**. Nesta abordagem, qualquer mudança que passe por todas as etapas do pipeline já é implantada automaticamente em produção, sem necessidade de aprovação humana a cada release. Em outras palavras, eliminamos o passo de “decisão de negócio” sobre quando fazer o deploy. Isso leva o conceito de entrega contínua ao nível máximo de agilidade, permitindo que features e correções cheguem aos usuários em tempo real, logo após serem validadas tecnicamente.

Mas como nem tudo são flores, adotar implantação contínua exige um grau de maturidade técnica e organizacional ainda maior.

É fundamental que os testes automatizados tenham altíssima qualidade e cobertura, pois sem aprovação manual o sistema precisa ter confiança total nos resultados dos testes para barrar qualquer problema antes de atingir os clientes. Também precisamos implementar um monitoramento muito detalhado, principalmente em produção. Alertas, métricas, traces, tudo que tivermos direito, para detectar rapidamente qualquer comportamento inesperado que eventualmente passe pelo pipeline.

Não só isso, mas estratégias de mitigação (como feature flags) tornam-se indispensáveis, garantindo que, se algo der errado em produção, seja possível reverter ou corrigir de forma imediata e segura. Em resumo, a implementação contínua só irá funcionar em locais onde a cultura DevOps esteja bem difundida, pois a organização toda deve estar alinhada com a ideia de lançamentos frequentes e automáticos, confiando nos processos de CI/CD e reagindo rapidamente a incidentes.

Para fechar este capítulo, vamos relembrar o estado em que estamos: nós refinamos o pipeline além da Integração Contínua, transformamos artefatos testados em versões produtivas. Se aprofunde e realize testes sobre as ferramentas e padrões vistos aqui, e você estará preparado para tornar o deploy tão trivial quanto um merge bem-sucedido. Ou, como diz o Jeferson, “*simples como voar :)*”

CONCLUSÃO

SE VOCÊ chegou até aqui, entendeu os conceitos apresentados e replicou os exemplos em seu ambiente, você está preparado para começar a utilizar o GitHub Actions no seu dia a dia! É claro que muitos outros tópicos relevantes (como controle de versão avançado e observabilidade) poderiam ser explorados, mas isso fica para outra oportunidade. O essencial é que agora desvendamos o caminho das pedras: aprendemos a implementar pipelines de CI/CD de forma consistente utilizando o GitHub Actions.

Durante o ebook, nós aprendemos sobre os principais tópicos mais importantes, dos fundamentos da ferramenta até o uso mais avançado. Exploramos como configurar nossos runners e como utilizar contextos, variáveis de ambiente, secrets e expressões para tornar as automações mais dinâmicas e inteligentes. Espero que sua jornada de leitura tenha sido satisfatória e que tenha entendido bem sobre os tópicos abordados por aqui.

O **Marketplace de Actions** que já estão prontas do GitHub conta com **mais de 5000 actions** e muito provavelmente terá um exemplo para a ferramenta/linguagem que você quer implementar. Ao entender no detalhe sobre o funcionamento de alguma automação específica, você terá a capacidade de adaptá-la para seu cenário e gerar seus próprios actions personalizados!

O QUE VEM A SEGUIR?

Vou repetir a mesma frase do começo deste ebook: “Se você chegou até este ebook, então muito provavelmente já está familiarizado com pelo menos alguns cenários de entrega de software, que por vezes acabam sendo bem caóticos”. Espero que este material consiga te dar uma base de entendimento, mas é claro que não podemos parar por aqui: Faça muitos exemplos, revise documentações (tire bastante proveito da documentação oficial de Actions no GitHub) e te convido também para conhecer o meu treinamento **Criando Pipelines e Automações com GitHub Actions** na **LINUXTips!**

Neste treinamento, teremos mais de **10 dias de conteúdo prático**, onde você aprenderá a **criar e gerenciar pipelines completos utilizando o GitHub Actions**. Iremos explorar **desde gatilhos simples até configurações mais elaboradas** envolvendo secrets, variables, cache e integrações com serviços em nuvem, além de trazer muitos exemplos e cenários do mundo real para resolvemos juntos. Este treinamento visa ajudar tanto quem é iniciante nesse mundo de pipelines quanto quem já tem alguma experiência. Também vamos aprender a teoria necessária, caso você queira realizar a Certificação Github Actions da GitHub.



[CLIQUE AQUI](#) para acessar o treinamento **Criando Pipelines e Automações com GitHub Actions** da LINUXtips

<https://linuxtips.io/github-actions/>

[CLIQUE AQUI](#) para acessar a documentação oficial do GitHub Actions

<https://docs.github.com/pt/actions>

AGRADECIMENTOS!

Primeiramente, agradeço você, leitor, pelo tempo dedicado a este material. Te convido a me adicionar como conexão do LinkedIn e também seguir meu perfil do GitHub. Se puder deixar um feedback no LinkedIn sobre este material no seu feed ou mesmo na minha DM, ficarei muito feliz! :)

Também quero agradecer muito a todo o time da LinuxTips pela oportunidade de estar compartilhando conhecimento de valor com uma grande comunidade e de forma gratuita! É muito gratificante participar de um ambiente onde o foco é realmente ensinar e apoiar seus alunos, sem enrolação.

Por fim, agradeço à minha companheira querida, Mayara, também revisora deste material.



LINKEDIN

linkedin.com/in/fabiobartoli



GITHUB

github.com/FabioBartoli

