Aula 10 – Construindo um Pipeline de CI com GitHub Actions - Parte 1

Imagine que você e sua equipe acabaram de passar semanas desenvolvendo uma funcionalidade incrível. O código parece perfeito na sua máquina. Chega o dia da entrega, você envia tudo para o repositório principal e... algo quebra. Um colega esqueceu de instalar uma dependência, outro usou uma versão diferente da linguagem, e o projeto simplesmente não funciona mais. Esse cenário, infelizmente comum, é o equivalente a uma orquestra onde cada músico tem uma partitura ligeiramente diferente. O resultado é o caos, a perda de tempo e a frustração.

É exatamente para evitar essa sinfonia do caos que a automação se torna nossa maior aliada. Pense em um pipeline de CI/CD (Integração Contínua e Entrega Contínua) como um maestro experiente e incansável. Ele não apenas garante que todos os "músicos" (desenvolvedores) estejam na mesma página, mas também verifica cada "nota" (linha de código) assim que ela é escrita, garantindo que o conjunto soe harmonioso. Ao final desta aula, você não será apenas um desenvolvedor; você será o arquiteto desse maestro digital, capaz de construir um fluxo de trabalho básico com GitHub Actions que valida seu código automaticamente, liberando sua mente para o que realmente importa: criar.

Nesta aula, vamos desvendar os segredos do GitHub Actions, começando do zero. Exploraremos os conceitos fundamentais de *workflows*, *events*, *jobs* e *steps*, que são os blocos de construção de qualquer automação. Aprenderemos a escrever nossa primeira "partitura" de automação usando a linguagem YAML e faremos nosso robô entrar em ação a cada push no repositório. Por fim, daremos a ele uma tarefa real: compilar nosso código e gerenciar suas dependências, dando o primeiro passo para um desenvolvimento verdadeiramente profissional e à prova de falhas.

O Mapa da Orquestra: O Que é o GitHub Actions?

Toda vez que entregamos um projeto, seja um trabalho acadêmico ou um software complexo, existe um ritual. Primeiro, verificamos a ortografia. Depois, garantimos que todas as partes estão presentes. Formatamos de acordo com as regras. Só então, com tudo conferido, nós o consideramos "pronto". No desenvolvimento de software, esse ritual é ainda mais rigoroso e repetitivo. Fazer isso manualmente é como revisar um livro de 500 páginas palavra por palavra, toda vez que uma única frase é alterada. É ineficiente e um convite ao erro humano.

> **GitHub Actions** é uma plataforma de automação integrada diretamente ao GitHub, projetada para executar esse "ritual" para nós.

Aqui entra o GitHub Actions. Ele é uma plataforma de automação integrada diretamente ao GitHub, projetada para executar esse "ritual" para nós. Pense nele como uma equipe de robôs assistentes que vive dentro do seu repositório. Você não precisa comprar um servidor, instalar um programa complexo ou fazer configurações mirabolantes. Ele já está lá, esperando por suas instruções. A principal tarefa desses robôs é realizar a **Integração Contínua (CI)**, que é a prática de mesclar as alterações de código de vários desenvolvedores em um repositório central de forma automática e frequente, validando cada nova adição.

Essa automação é definida em arquivos de texto simples, que funcionam como roteiros ou "playbooks" para os robôs. A beleza disso se conecta diretamente a uma tendência chamada **GitOps**: o próprio repositório Git, que já guarda nosso código, passa a guardar também as instruções para testar, construir e até mesmo implantar esse código. Tudo fica centralizado, versionado e rastreável. Em vez de apertar botões em painéis complexos, nós descrevemos o que queremos que aconteça, e o GitHub Actions cuida da execução, transformando nosso repositório em uma verdadeira fábrica de software inteligente.

A Anatomia da Automação: Workflows, Events, Jobs e Steps

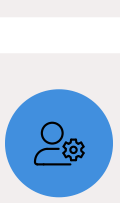
Para ensinar nossos robôs assistentes a trabalhar, precisamos de uma linguagem e uma estrutura que eles entendam. Não podemos simplesmente dizer "verifique meu código". Precisamos detalhar o processo, passo a passo, de forma lógica e organizada. O GitHub Actions nos oferece uma estrutura hierárquica clara para fazer exatamente isso, e entendê-la é como aprender a gramática de um novo idioma. A fluência virá da compreensão de suas quatro peças fundamentais: *workflows*, *events*, *jobs* e *steps*.

Workflow

O processo completo que queremos automatizar. Como pedir uma pizza do início ao fim.

Event

O gatilho que inicia tudo. Como clicar no botão "Fazer Pedido" ou fazer um push no repositório.



Job

Grandes etapas que podem ocorrer em paralelo. Como a cozinha preparar a pizza e o entregador ser acionado.

Step

Ações específicas dentro de um job. Como "abrir a massa", "adicionar o molho", "colocar no forno".

Vamos usar uma analogia do cotidiano: pedir uma pizza por um aplicativo. O ato completo, desde a escolha do sabor até a entrega na sua porta, é o **workflow** (fluxo de trabalho). É o processo completo que queremos automatizar. O gatilho que inicia tudo, como você clicar no botão "Fazer Pedido", é o **event** (evento). No nosso caso, um evento poderia ser alguém enviando código novo para o repositório (push). Dentro do seu pedido, há duas grandes etapas que podem ocorrer em paralelo: a cozinha prepara a pizza e um entregador é acionado. Cada uma dessas grandes etapas é um **job** (trabalho). Finalmente, as ações específicas dentro da cozinha, como "abrir a massa", "adicionar o molho", "colocar no forno", são os **steps** (passos).

Essa estrutura nos permite criar automações incrivelmente flexíveis. Podemos ter um *workflow* simples com apenas um *job* e alguns *steps*, ou *workflows* complexos com múltiplos *jobs* que dependem uns dos outros (por exemplo, o *job* de "empacotar a pizza" só pode começar depois que o *job* de "assar a pizza" terminar). Essa organização é a base de tudo o que construiremos. Dominar essa hierarquia é o primeiro e mais importante passo para se tornar um arquiteto de pipelines eficientes.

A Partitura da Automação: A Estrutura de um Arquivo YAML

Já entendemos a estrutura lógica do nosso processo de automação, mas como comunicamos isso ao GitHub Actions de forma prática? A resposta está em um formato de arquivo chamado YAML (lê-se "iâmel"). Se o HTML estrutura páginas da web e o Python executa lógicas complexas, o YAML serve para estruturar dados de configuração. Ele foi projetado para ser extremamente fácil de ler por humanos, o que é perfeito para definir nossos *workflows*.

> ⚠️ **Atenção:** A indentação no YAML é tudo! Um espaço a mais ou a menos pode mudar completamente o significado da instrução.

Pense no YAML como uma receita de bolo escrita de forma muito organizada. Em vez de um texto corrido, você usa títulos e subtítulos com um recuo (indentação) específico para mostrar a hierarquia. A receita principal tem um nome (name). Os ingredientes e as instruções são pares de chave: valor. Por exemplo: nome-da-receita: "Bolo de Chocolate". Para listar os ingredientes, você usaria hifens. A coisa mais importante sobre o YAML é que **a indentação é tudo**. Um espaço a mais ou a menos pode mudar completamente o significado da instrução, assim como errar a quantidade de um ingrediente pode arruinar a receita.

No contexto do GitHub Actions, nosso arquivo .yml ou .yaml sempre terá uma estrutura básica. Começamos dando um name ao nosso *workflow*. Em seguida, definimos o *evento* que o acionará na seção on. Depois, descrevemos os jobs. Dentro de cada *job*, especificamos em qual tipo de máquina ele deve rodar com runs-on (como ubuntu-latest, a versão mais recente do Ubuntu) e, finalmente, listamos os steps que ele deve executar. Cada step pode ter um name para descrever o que faz e uma instrução run para executar um comando no terminal, como se estivéssemos digitando diretamente nele.

```
# Nome do nosso fluxo de trabalho
name: Meu Primeiro Workflow

# Evento que aciona o workflow
on:
  push:
    branches: [ main ]

# Trabalhos a serem executados
jobs:
  meu-primeiro-job:
    # Máquina virtual onde o job vai rodar
    runs-on: ubuntu-latest

    # Passos que compõem o job
    steps:
      - name: Imprime uma mensagem de boas-vindas
        run: echo "Olá, mundo! A automação começou!"

      - name: Mostra a estrutura de arquivos
        run: ls -la
```

Este pequeno bloco de código é uma receita completa. Ele diz ao GitHub: "Seu nome é 'Meu Primeiro Workflow'. Quero que você execute sempre que houver um push no branch main. Para isso, inicie um job chamado meu-primeiro-job em uma máquina Ubuntu. Nesse *job*, execute dois *steps*: primeiro, imprima uma mensagem de boas-vindas e, segundo, liste todos os arquivos no diretório." Simples, legível e poderoso.

O Santuário da Automação: Criando o Primeiro Arquivo de Workflow

Agora que já conhecemos a teoria e a linguagem, é hora de colocar a mão na massa. Onde exatamente esses arquivos YAML devem morar dentro do nosso projeto? O GitHub Actions é muito específico sobre isso. Ele não vai procurar por arquivos de *workflow* em qualquer pasta; ele espera encontrá-los em um diretório secreto e padronizado, uma espécie de santuário dedicado exclusivamente à automação.

Localização Obrigatória

Os arquivos de workflow devem estar em `.github/workflows` na raiz do seu repositório.

Formato do Arquivo

Use a extensão `.yml` ou `.yaml` para seus arquivos de workflow.

Detecção Automática

Qualquer arquivo válido nesta pasta será automaticamente reconhecido pelo GitHub Actions.

Esse lugar especial é uma pasta chamada .github/workflows na raiz do seu repositório. O ponto no início do nome .github a torna um diretório oculto na maioria dos sistemas operacionais, sinalizando que contém arquivos de configuração e metadados, e não o código-fonte da sua aplicação. Qualquer arquivo .yml ou .yaml que você colocar dentro da pasta workflows será automaticamente detectado pelo GitHub e tratado como uma definição de *workflow* a ser executada.

A criação é simples. No seu projeto local, você pode simplesmente criar essa estrutura de diretórios e o arquivo. Por exemplo, `mkdir -p .github/workflows` e depois `touch .github/workflows/ci-basico.yml`. A partir desse momento, você pode abrir o arquivo ci-basico.yml no seu editor de código preferido e começar a escrever a "receita" de automação que vimos na página anterior. Assim que você commitar e enviar (push) essa nova estrutura de pastas e o arquivo para o seu repositório no GitHub, o sistema o reconhecerá instantaneamente. Na verdade, o próprio ato de enviar esse arquivo pela primeira vez já vai acionar o *workflow* que ele define! É um ciclo que se autoinicia, demonstrando o poder imediato da automação baseada em eventos.

Executando o Roteiro: Acionando a Execução com um push

Temos o roteiro (nosso arquivo YAML) e o palco (a infraestrutura do GitHub Actions). Agora, precisamos do ator principal para entrar em cena e dar início à peça. Esse "início" é determinado pelo evento que especificamos na cláusula on: do nosso arquivo de *workflow*. Embora existam dezenas de eventos possíveis, como abrir uma *pull request*, criar uma *issue* ou até mesmo agendar uma execução para um horário específico, o evento mais fundamental e comum é, sem dúvida, o push.

O push é o comando que todo desenvolvedor usa dezenas de vezes ao dia. É o ato de enviar as alterações do seu computador local para o repositório remoto no GitHub. Ao configurar nosso *workflow* para ser acionado on: push, estamos criando um elo direto e poderoso entre o nosso trabalho diário e o processo de validação automática. Cada vez que você ou um colega de equipe decidir que um pedaço de código está pronto para ser compartilhado, o nosso maestro robótico entrará em ação imediatamente para reger a orquestra da verificação.

> 💡 **Dica de Otimização:** Especificar branches específicos evita execuções desnecessárias, economizando tempo, dinheiro e energia (FinOps e GreenOps).

Vamos refinar um pouco mais. Muitas vezes, não queremos que a automação rode para *qualquer* push em *qualquer* branch. Em geral, as verificações mais importantes são reservadas para os branches principais, como o main ou develop. Podemos especificar isso facilmente no nosso YAML, como no exemplo abaixo. Essa configuração diz ao GitHub Actions: "Fique atento a qualquer push, mas só inicie o *workflow* se o push ocorrer especificamente no branch main." Esse nível de controle é crucial para otimizar recursos, alinhando-se a práticas de **FinOps** e **GreenOps**, pois evita a execução de automações desnecessárias, economizando tempo de processamento, dinheiro e energia.

```
on:
 push:
 branches:
 - main # Você poderia adicionar outros branches aqui, como 'develop'
```

O Palco da Execução: A Aba "Actions" e os Logs

Depois de enviar o seu novo arquivo de *workflow* e fazer um push no branch main, como sabemos se algo realmente aconteceu? Onde podemos assistir à performance do nosso robô? O GitHub oferece uma interface completa e transparente para que possamos acompanhar cada passo da execução, um verdadeiro "backstage" da nossa automação. Essa área é a aba **"Actions"**, localizada na barra de navegação principal do seu repositório, ao lado de "Code", "Issues" e "Pull Requests".

01

Lista de Execuções

Veja todos os workflows executados com ícones de status: amarelo (em andamento), verde (sucesso), vermelho (falha).

02

Visão Detalhada

Clique em uma execução para ver os jobs que a compõem e seu status individual.

03

Logs Granulares

Clique em um job para ver os logs de cada step, exatamente como no terminal.

Ao clicar nessa aba, você verá uma lista de todos os *workflows* que já foram executados ou estão em execução. Cada execução é representada por uma linha, com um ícone que indica seu status: um círculo amarelo girando para execuções em andamento, um "check" verde para sucesso, ou um "X" vermelho para falha. Clicar em uma execução específica abre uma visão detalhada, mostrando os *jobs* que a compõem. Clicando em um *job*, você finalmente chega ao nível mais granular: os logs de cada *step*.

Essa visualização detalhada é a essência da **Observabilidade** em CI/CD. Não basta saber *se* o processo falhou; precisamos saber *onde* e *por quê*. Os logs mostram a saída de cada comando que você definiu no seu arquivo YAML, exatamente como se estivessem sendo executados no seu próprio terminal. Se um comando falhar, a mensagem de erro estará lá, em vermelho, indicando o ponto exato do problema. Essa capacidade de depuração é o que transforma o GitHub Actions de uma "caixa preta" mágica em uma ferramenta de engenharia poderosa e controlável, permitindo que você identifique e corrija problemas no seu processo de automação de forma rápida e eficiente.

Do Ensaio à Peça Principal: Compilando o Código-Fonte

Até agora, nossos exemplos foram como ensaios. Usamos o comando echo para imprimir mensagens simples, o que é ótimo para confirmar que a automação está funcionando, mas não agrega valor real ao nosso projeto. Chegou a hora de dar ao nosso robô uma tarefa de verdade: interagir com o nosso código-fonte. Afinal, o objetivo da Integração Contínua é validar o *código*, não apenas imprimir "Olá, Mundo".

O Problema

Quando o GitHub Actions inicia um job, a máquina virtual está completamente vazia. Ela não contém o código do seu projeto por padrão.

Seria como contratar um chef para sua cozinha, mas não lhe dar acesso à despensa.

A Solução

Usamos a ação `actions/checkout` para "baixar" o código do repositório para a máquina virtual.

É o equivalente a executar `git clone` no início do processo.

O primeiro desafio é: quando o GitHub Actions inicia um *job*, a máquina virtual que ele cria está completamente vazia. Ela não contém o código do seu projeto por padrão. Seria como contratar um chef para sua cozinha, mas não lhe dar acesso à despensa. Para resolver isso, usamos um dos "passos" mais importantes e onipresentes do ecossistema do GitHub Actions: a ação actions/checkout. Uma "ação" é um *step* pré-fabricado e reutilizável, criado pela comunidade ou pelo próprio GitHub para realizar tarefas comuns.

A ação checkout tem uma única e crucial responsabilidade: ela "baixa" o código do seu repositório para dentro da máquina virtual onde o *job* está rodando. É o equivalente a executar git clone no início do processo. Ao adicionar este *step* — geralmente como o primeiro de qualquer *job* que precise manipular o código —, garantimos que todos os passos subsequentes terão acesso aos arquivos do projeto. Isso nos permite avançar para tarefas mais significativas, como instalar dependências e compilar o código.

```
steps:
 # 1. Baixa o código do repositório para a máquina virtual
 - name: Checkout do código
 uses: actions/checkout@v4 # Sempre use uma versão específica

 # 2. Agora podemos interagir com os arquivos
 - name: Listar arquivos para confirmar
 run: ls -la
```

Reunindo as Ferramentas: Gerenciando Dependências

Com o nosso código finalmente disponível na máquina virtual graças ao actions/checkout, enfrentamos o próximo passo lógico em qualquer projeto de software moderno: as dependências. Nenhum projeto vive isolado. Seja usando pacotes do npm em um projeto JavaScript, bibliotecas do pip em Python ou gems em Ruby, nosso código depende de ferramentas e bibliotecas externas para funcionar. A máquina virtual do GitHub Actions, por padrão, não tem nenhuma dessas dependências instaladas.

Código Disponível

Checkout realizado com sucesso

Configurar Ambiente

Instalar linguagem (Node, Python, etc.)

Instalar Dependências

npm install, pip install, etc.

Pronto para Build

Ambiente reproduzível configurado

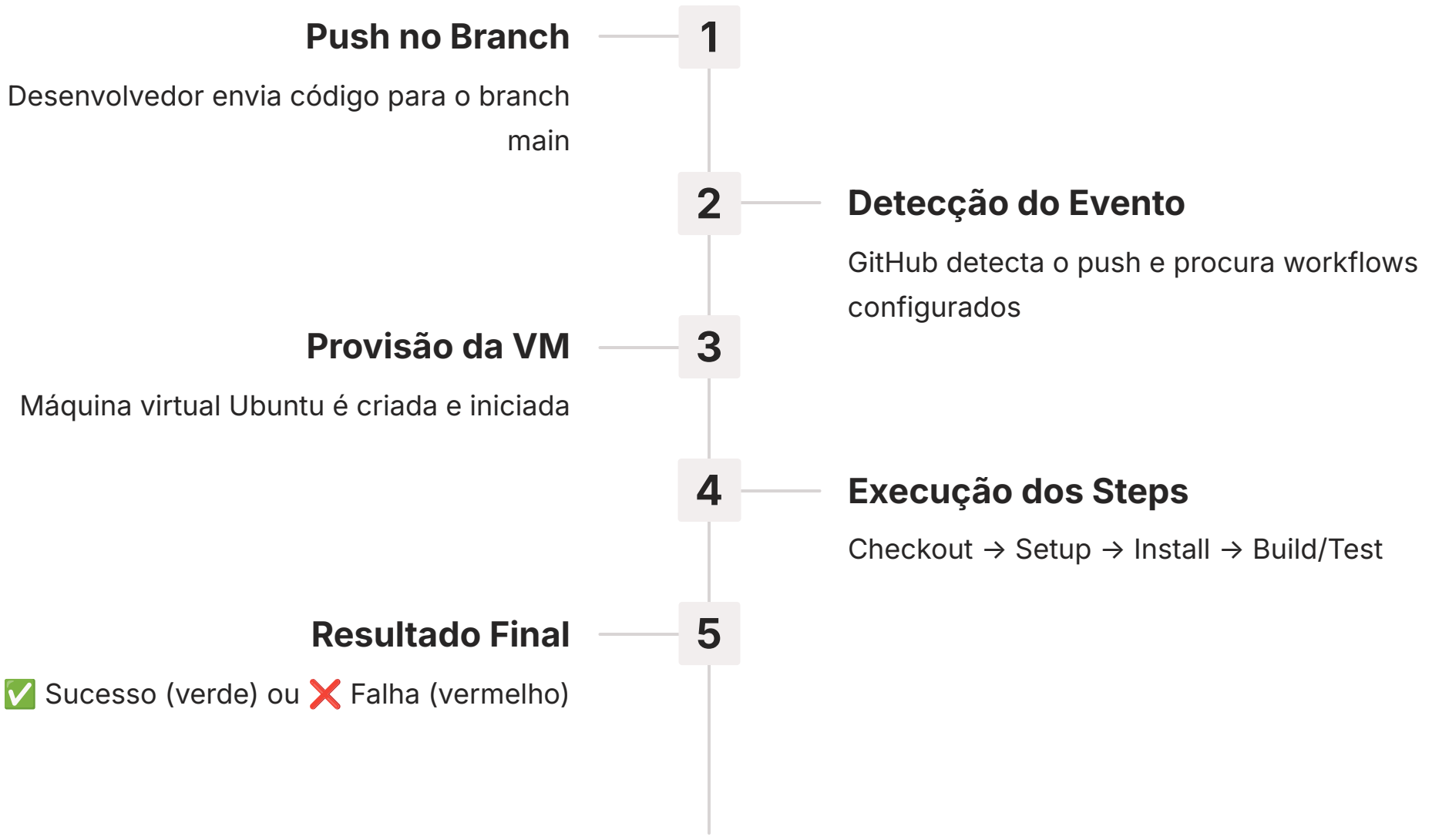
A tarefa do nosso *workflow* de CI é simular o ambiente de um novo desenvolvedor (ou de um servidor de produção) e garantir que o projeto possa ser configurado do zero sem problemas. Portanto, um passo crucial é a instalação dessas dependências. Isso é geralmente feito executando o comando padrão do gerenciador de pacotes da linguagem que estamos utilizando. Por exemplo, em um projeto Node.js, adicionaríamos um *step* para rodar npm install. Em um projeto Python, seria pip install -r requirements.txt.

Para que esses comandos funcionem, a máquina virtual precisa ter a linguagem de programação correspondente instalada (Node.js, Python, etc.). Felizmente, existem outras ações prontas para isso, como a actions/setup-node ou actions/setup-python. Essas ações preparam o ambiente de forma rápida e confiável. Este processo de configurar o ambiente e instalar as dependências é o coração da CI. Ele valida que o "manual de instruções" do nosso projeto (o package.json ou requirements.txt) está correto e que o projeto é autossuficiente e reproduzível em qualquer lugar.

```
steps:
 - uses: actions/checkout@v4

 - name: Configurar Node.js
 uses: actions/setup-node@v4
 with:
 node-version: '20' # Especifica a versão do Node.js

 - name: Instalar dependências
 run: npm install

 # Passo seguinte seria rodar testes ou o build
 - name: Rodar o build (exemplo)
 run: npm run build --if-present
```

O Fluxo Completo: Do push à Validação

Vamos juntar todas as peças que aprendemos até agora. Vimos os conceitos, a sintaxe do YAML, como acionar a automação e como lidar com o código e suas dependências. Agora, podemos visualizar o fluxo completo de um pipeline de CI básico, desde o momento em que um desenvolvedor digita git push até o feedback final do GitHub Actions. É a coreografia completa da nossa automação em ação.



A jornada começa com o **push** para o branch main. Este é o nosso **evento** gatilho. O GitHub detecta esse evento e procura por arquivos de *workflow* em .github/workflows/ que estejam configurados para responder a ele. Ao encontrar nosso arquivo, ele inicia uma nova execução do *workflow*. A plataforma então provisiona uma máquina virtual, geralmente baseada em Ubuntu, conforme especificado em runs-on. Este é o nosso **job** começando.

Dentro do *job*, os **steps** são executados em sequência. O primeiro, actions/checkout, clona nosso repositório para a máquina. O segundo, actions/setup-node (ou equivalente), instala a versão correta da nossa linguagem de programação. O terceiro, npm install (ou equivalente), baixa todas as dependências do projeto. Finalmente, um quarto *step* poderia rodar um comando de compilação (npm run build) ou de teste (npm test). Se todos os *steps* forem concluídos com sucesso (código de saída 0), o *job* é marcado com um "check" verde. Se qualquer um deles falhar, a execução para imediatamente e o *job* é marcado com um "X" vermelho. O desenvolvedor recebe uma notificação e pode clicar para ver os logs e entender exatamente o que deu errado.

A Automação e as Tendências: Conectando a Teoria à Prática de 2025

O que estamos construindo aqui não é apenas um exercício acadêmico. Este simples pipeline de CI é a pedra fundamental sobre a qual as práticas de DevOps mais avançadas e as tendências de mercado para 2025 são construídas. Compreender como conectar nosso trabalho de hoje com o futuro da tecnologia nos dá uma vantagem competitiva enorme, seja para passar em um concurso público ou para se destacar no mercado de trabalho.

DevSecOps

O pipeline é o lugar perfeito para adicionar análise de segurança estática (SAST), detectando vulnerabilidades automaticamente a cada push e movendo a segurança para o início do ciclo.

Platform Engineering

Equipes de plataforma criam modelos de workflows e os oferecem como serviço, abstraindo a complexidade para os desenvolvedores.

Observabilidade & AIOps

Logs e métricas alimentam sistemas de IA que preveem falhas e otimizam processos, garantindo eficiência e resiliência.

Pense no **DevSecOps**, ou "Shift-Left Security". O pipeline que criamos é o lugar perfeito para, no futuro, adicionarmos um *step* que executa uma ferramenta de análise de segurança estática (SAST). Em vez de esperar que a equipe de segurança encontre vulnerabilidades semanas depois, nós as detectamos automaticamente a cada push, movendo a segurança para a "esquerda", para o início do ciclo de desenvolvimento.

Da mesma forma, esta é a base da **Engenharia de Plataforma (Platform Engineering)**. Em grandes empresas, uma equipe de plataforma cria modelos de *workflows* como este e os oferece como um serviço para as equipes de desenvolvimento. Os desenvolvedores não precisam saber os detalhes de como o pipeline funciona; eles simplesmente usam a "plataforma" pronta para garantir a qualidade de seu código, abstraindo a complexidade. Cada log e métrica gerados por essas execuções alimenta sistemas de **Observabilidade** e **AIOps**, que usam inteligência artificial para prever falhas e otimizar o processo, garantindo que tudo funcione de maneira eficiente e resiliente.

Quadro Comparativo: Os Pilares da Automação

Depois de explorarmos narrativamente os componentes do GitHub Actions, um quadro comparativo pode ajudar a solidificar as distinções entre esses conceitos-chave. Pense neles como as diferentes camadas de uma organização: do plano estratégico geral à tarefa individual.

Conceito	Âmbito/Aplicação	Base/Origem	Exemplo Prático
Workflow	O processo de automação completo, do início ao fim.	Definido no arquivo .yml inteiro.	Um pipeline de "Integração Contínua" ou de "Implantação em Produção".
Event	O gatilho que inicia a execução de um workflow.	Seção `on:` no arquivo YAML.	Um push para o branch main, ou a criação de uma pull_request.
Job	Um conjunto de passos que executa em uma máquina virtual.	Seção `jobs:` no arquivo YAML.	Um job para "Construir o Código" (build) e outro para "Rodar os Testes" (test).
Step	Uma tarefa ou comando individual dentro de um job.	Seção `steps:` dentro de um job.	Executar `actions/checkout@v4`, rodar o comando `npm install` ou `echo "Olá"`.

Esta estrutura hierárquica é o que confere poder e flexibilidade ao GitHub Actions, permitindo a criação de automações que vão desde a verificação mais simples até orquestrações de implantação em múltiplos ambientes de nuvem.

Polindo a Nossa Criação: Dicas e Boas Práticas Iniciais

Criar nosso primeiro *workflow* funcional é uma grande vitória, mas como em qualquer ofício, a diferença entre um trabalho amador e um profissional está nos detalhes. Adotar boas práticas desde o início não apenas torna nossos pipelines mais fáceis de manter, mas também mais seguros e eficientes. É como aprender a organizar suas ferramentas depois de usá-las; economiza muito tempo e dor de cabeça no futuro.

1	2	3
Nomes Descritivos	**Fixar Versões**	**Workflows Focados**
Sempre dê nomes claros aos seus jobs e steps. Em vez de "Run command", use "Instalar dependências NPM". A clareza é um presente para o futuro.	Use `actions/checkout@v4` em vez de `@main`. Isso garante que seu workflow não quebre inesperadamente com atualizações incompatíveis.	Mantenha workflows com responsabilidades únicas. Um para CI (testar e construir), outro para CD (implantar). Facilita gerenciamento e depuração.

A primeira dica é sempre dar **nomes descritivos** aos seus *jobs* e *steps*. Em vez de um *step* chamado apenas "Run command", use name: Instalar dependências NPM. Quando você revisitar o arquivo daqui a seis meses, ou quando um colega de equipe precisar entendê-lo, a clareza será um presente. A segunda prática crucial é **fixar as versões das ações** que você usa. Note que usamos actions/checkout@v4 e actions/setup-node@v4. Usar @v4 em vez de @main garante que seu *workflow* não quebre inesperadamente quando o autor da ação lançar uma nova versão com alterações incompatíveis.

Por fim, mantenha seus *workflows* focados. É tentador criar um único arquivo YAML monolítico que faz tudo: testa, constrói, analisa a segurança e implanta em três ambientes diferentes. Uma abordagem melhor é ter *workflows* menores e com responsabilidades únicas. Um para CI (testar e construir), outro para CD (implantar). Isso os torna mais fáceis de gerenciar, depurar e acionar com base em eventos diferentes, seguindo o princípio da responsabilidade única, um pilar da boa engenharia de software.

Conexões para o Futuro: Além da Compilação

O que realizamos hoje é o equivalente a construir o chassi e o motor de um carro. Ele liga, anda e já é imensamente útil. Nosso código agora é verificado a cada push, garantindo um nível básico de sanidade e reprodutibilidade. Já saímos do mundo manual e entramos na era da automação. Vimos como os conceitos de *workflow*, *job* e *step* formam a espinha dorsal de qualquer pipeline e como a sintaxe YAML os organiza de forma legível.

O que construímos hoje

- Pipeline básico de CI
- Checkout automático de código
- Configuração de ambiente
- Instalação de dependências
- Compilação do código

O que vem a seguir

- Testes automatizados
- Gerenciamento de artefatos
- Notificações da equipe
- Análise de qualidade
- Deploy automatizado

Mas a história não termina aqui. O carro que construímos ainda não tem rodas de teste, um sistema de navegação para implantação ou um painel de controle com métricas de segurança. O pipeline de CI é a fundação, mas a verdadeira potência do DevOps vem das etapas que construiremos sobre ele. Como podemos garantir que, além de compilar, o código também passa em uma suíte de testes automatizados? Como salvamos o resultado da compilação – o "artefato" – para usá-lo mais tarde? E como notificamos a equipe sobre o sucesso ou a falha do processo?

Essas são exatamente as perguntas que nos guiarão na próxima aula. Vamos pegar a base sólida que criamos hoje e adicionar camadas essenciais de qualidade e feedback. Exploraremos como integrar testes automatizados, gerenciar artefatos de build e configurar notificações, transformando nosso simples verificador de código em um pipeline de integração contínua robusto e completo, pronto para os desafios do desenvolvimento de software profissional.

Consolidação e Próximos Passos

Nesta aula, demos o primeiro e mais importante passo no mundo da automação com GitHub Actions. Desmistificamos a "mágica" por trás da Integração Contínua, traduzindo-a em conceitos claros e práticos. Começamos entendendo o problema da validação manual e apresentamos o GitHub Actions como nosso maestro robótico. Navegamos por sua estrutura fundamental — *workflows*, *events*, *jobs* e *steps* — usando a analogia de uma receita. Aprendemos a escrever essa receita em YAML, a acioná-la com um push e a dar-lhe tarefas reais, como fazer o checkout do código e instalar dependências. Este é o alicerce sobre o qual toda automação de qualidade é construída.

Em Prática

Automatize o Básico

Crie um workflow simples em um projeto pessoal para rodar a cada push, mesmo que seja apenas para listar arquivos e imprimir uma mensagem.

Valide as Dependências

Adicione os passos de setup da linguagem e instalação de dependências a esse workflow. Isso já garante que seu requirements.txt ou package.json está sempre válido.

Explore a Aba "Actions"

Acione uma falha de propósito no seu workflow (por exemplo, um comando inválido) e navegue pelos logs na aba "Actions" para identificar a causa do erro.

Fixe as Versões

Ao usar uma ação da comunidade (como actions/checkout), sempre especifique uma versão (@v4) para garantir a estabilidade do seu pipeline.

Autoavaliação

1. (Nível Fácil) Em um arquivo de workflow do GitHub Actions, qual é a principal função da chave on:?

- A) Definir o nome do workflow para exibição na UI.
- B) Listar os comandos a serem executados na máquina virtual.
- C) Especificar o evento (ou eventos) que irá acionar a execução do workflow.
- D) Indicar a versão do sistema operacional que o job deve usar.

2. (Nível Médio) Qual das seguintes ações é fundamental para permitir que um job acesse o código-fonte do repositório?

- A) actions/setup-node@v4
- B) actions/upload-artifact@v3
- C) actions/checkout@v4
- D) actions/cache@v3

3. (Nível Concurso) Para um analista de sistemas que precisa garantir que um pipeline de CI seja executado apenas quando o código é mesclado no branch principal de produção, denominado prod, qual seria a configuração YAML mais adequada dentro da seção on:?

- A) on: push
- B) on: push: branches: [main]
- C) on: push: branches: [prod]
- D) on: pull_request: branches: [prod]

4. (Nível Difícil) Um desenvolvedor criou um workflow com dois jobs: job_A e job_B. Por padrão, como o GitHub Actions executa esses dois jobs?

- A) Em sequência, job_A sempre antes de job_B.
- B) Em sequência, job_B sempre antes de job_A.
- C) Em paralelo, ao mesmo tempo.
- D) A execução falha, pois as dependências não foram especificadas.

5. (Discursiva) Utilizando uma analogia, explique a relação hierárquica entre um workflow, um job e um step no GitHub Actions.

Gabarito e Recursos Adicionais

Gabarito

Questão 1

Resposta: C

A chave `on:` especifica o evento que aciona o workflow.

Questão 2

Resposta: C

A ação `actions/checkout@v4` baixa o código para a VM.

Questão 3

Resposta: C

Deve especificar o branch `prod` na configuração de push.

Questão 4

Resposta: C

Por padrão, jobs executam em paralelo simultaneamente.

Resposta Discursiva (Exemplo)

A relação é como planejar uma viagem de carro (workflow). A viagem inteira tem etapas principais, como "Dirigir de São Paulo ao Rio de Janeiro" (job). Cada etapa principal é composta por ações menores e sequenciais, como "Entrar no carro", "Ligar o GPS", "Dirigir pela Dutra" (steps). O workflow é o plano completo, os jobs são as grandes fases, e os steps são as ações individuais dentro de cada fase.

Conexão com a Próxima Aula

Na **Aula 11 – Construindo um Pipeline de CI com GitHub Actions - Parte 2**, vamos evoluir nosso pipeline. Adicionaremos a capacidade de rodar testes automatizados para validar a lógica do nosso código, aprenderemos a gerenciar e salvar os artefatos gerados pelo processo de build e exploraremos como configurar notificações para manter a equipe informada.

Recursos Adicionais

Documentação Oficial

A fonte definitiva para todos os conceitos e sintaxes do GitHub Actions.

docs.github.com/pt/actions

GitHub Skills - CI/CD

Um curso interativo e prático oferecido pelo próprio GitHub para aprender fazendo.

skills.github.com

NOTA IMPORTANTE: As informações técnicas e exemplos de sintaxe desta aula estão atualizadas até 2025. O ecossistema do GitHub Actions evolui rapidamente; consulte sempre a documentação oficial para verificar as versões mais recentes das ações e as melhores práticas.