Implementação de Pipeline CI/CD para Aplicações com GitHub Actions e Docker: Um Estudo de Caso do Beekeeper Studio

Matheus Rodrigues Fernandes Arcelino
matheus.arcelino@sempreceub.com

15 de setembro de 2024

Resumo

Este documento apresenta a implementação de um pipeline CI/CD para o Beekeeper Studio usando o GitHub Actions, focando na automação das etapas de construção, teste, análise de segurança e publicação da aplicação. Além disso, ele explica como configurar um Dockerfile, publicar imagens no Docker Hub e rodar a aplicação localmente a partir de uma imagem Docker. O estudo enfatiza que a automação é vital para projetos de software para garantir qualidade e eficiência durante o processo de desenvolvimento e entrega contínuos.

Conteúdo

1 Introdução

O presente estudo de caso tem como objetivo o desenvolvimento de uma pipeline de Integração Contínua e Entrega Contínua (CI/CD) para um projeto open source utilizando o GitHub Actions. As etapas de construção, testes, análise de qualidade de código e publicação de artefatos foram automatizadas pelo pipeline desenvolvida, permitindo um fluxo contínuo de integração e entrega do software. Esse desafio permitiu contribuir para projetos de software reais e aplicar na prática os conceitos discutidos em aula.

CI/CD em projetos de software é essencial, quando se deseja ter qualidade e agilidade. Através da integração contínua (CI), é possível garantir que o código enviado ao repositório seja sempre verificado por meio de testes automatizados e análises de qualidade, reduzindo a probabilidade de falhas e erros no ambiente de produção. Já a entrega contínua (CD), garante que novas funcionalidades e correções de bugs e erros sejam entregues aos usuários finais de forma confiável e ágil. O ciclo de desenvolvimento pode ser escalável e mais eficiente com essas práticas. Isso resulta em produtos de maior qualidade e menor tempo de entrega ao mercado.

A seguir, é apresentado o software escolhido para o desenvolvimento deste estudo de caso.

1.1 Beekeeper Studio

Beekeeper Studio é um gerenciador de banco de dados e editor SQL que fornece uma interface gráfica intuitiva para se conectar a diversos bancos de dados, como MySQL, PostgreSQL e SQLite. Como ele podemos gerenciar schema, executar consultas SQL, visualizar e editar dados. O histórico de consultas, o suporte a múltiplas abas e os temas personalizáveis são alguns dos recursos presentes no software.

Disponível para Linux, Mac e Windows, o Beekeeper Studio é um software multiplataforma. A versão Community Edition, que é distribuída e sob a GPL, ou seja, é gratuita e livre para uso e modificação, foi a escolhida para o estudo de caso.

2 Objetivo e Justificativa do Pipeline CI/CD

2.1 Objetivo do Pipeline CI/CD

Objetivo da pipeline de Integração Contínua e Entrega Contínua (CI/CD) do projeto Beekeeper Studio é garantir a confiabilidade e eficiência, durante o processo de distribuição. Cada modificação enviada ao repositório passa automaticamente por um processo de construção, testes automatizados, validação de qualidade e publicação. Garantindo uma entrega de código de forma contínua, diminuindo o risco de erros e acelerando o ciclo de desenvolvimento.

2.2 Justificativa do Pipeline CI/CD

A principal justificativa para a implementação do CI/CD é a necessidade de otimizar o fluxo de desenvolvimento, garantindo que o código seja constantemente testado e validado antes de ser integrado ao projeto principal. Tal processo reduz os custos e o tempo de correção de bugs e erros, ao mesmo tempo, aumentando a qualidade do código durante o desenvolvimento. Além disso, permite a entrega de novas versões de forma mais rápida e segura, o que permite que o projeto evolua de forma contínua e sem interrupções.

3 Clone do Repositório

Para o desenvolvimento da pipeline, foi utilizada a versão de código aberto do Beekeeper Studio, um cliente completo de gerenciamento de banco de dados que é totalmente gratuito e open source. O projeto da comunidade conta com 16.100 estrelas e 1.100 forks, demonstrando seu crescente reconhecimento e uso pela comunidade de desenvolvedores.

3.1 Passo a passo do clone do repositório original

Para garantir que as alterações e testes no pipeline CI/CD possam ser feitos de forma independente, sem afetar o repositório principal, foi realizado fork do repositório original do projeto beekeeper-studio/beekeeper-studio. O fork foi clonado para o ambiente de desenvolvimento local, facilitando a implementação e a validação das mudanças necessárias. Abaixo, estão as imagens que demonstram o processo de fork e clonagem do repositório.

- Repositório oficial do Beekeeper Studio no GitHub

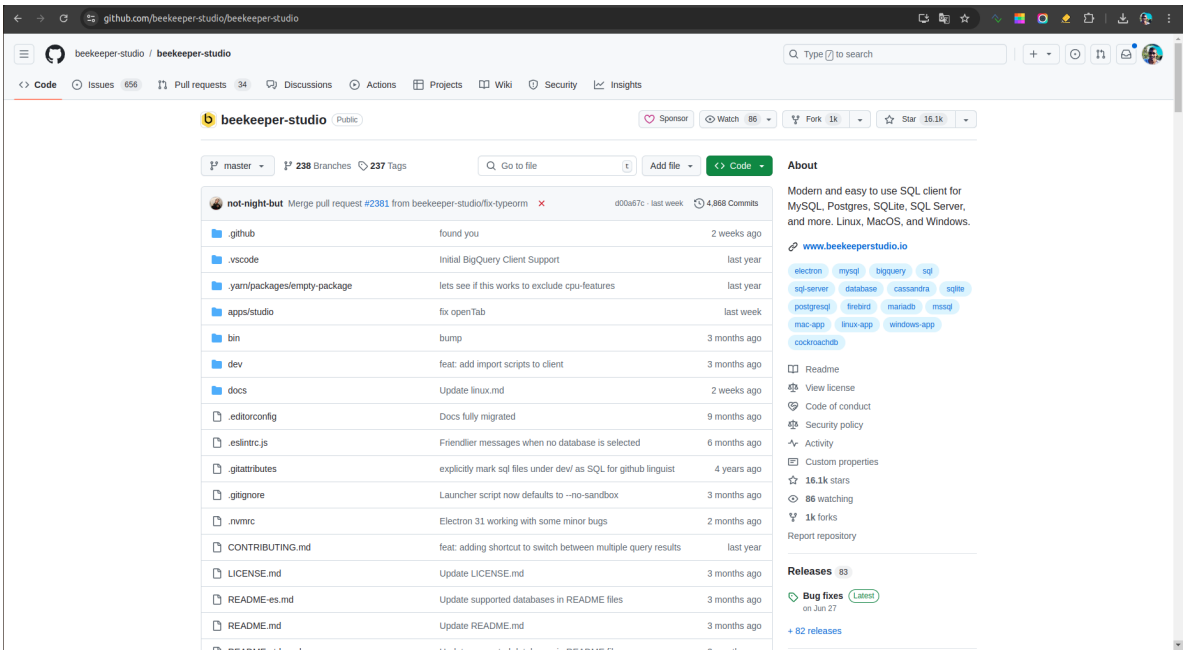

Figura 1: Repositório oficial do Beekeeper Studio, disponível em GitHub.

- Realizando o fork do projeto

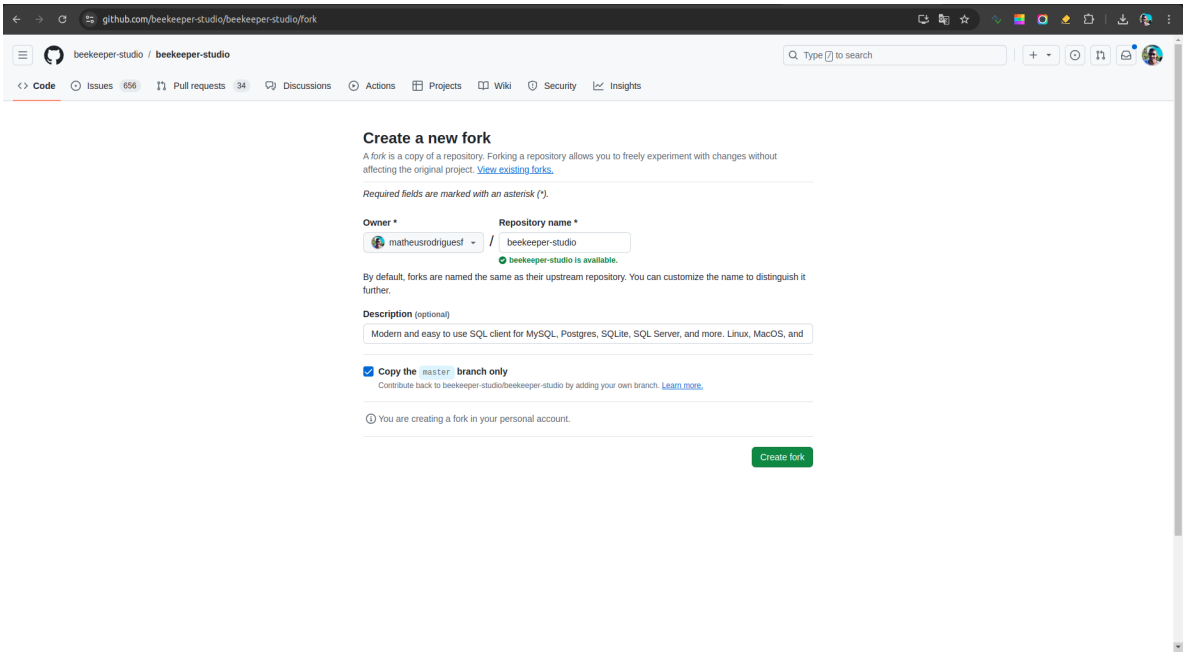

Figura 2: Processo de fork do repositório, criando uma cópia em um repositório pessoal para personalização.

- Repositório após o fork

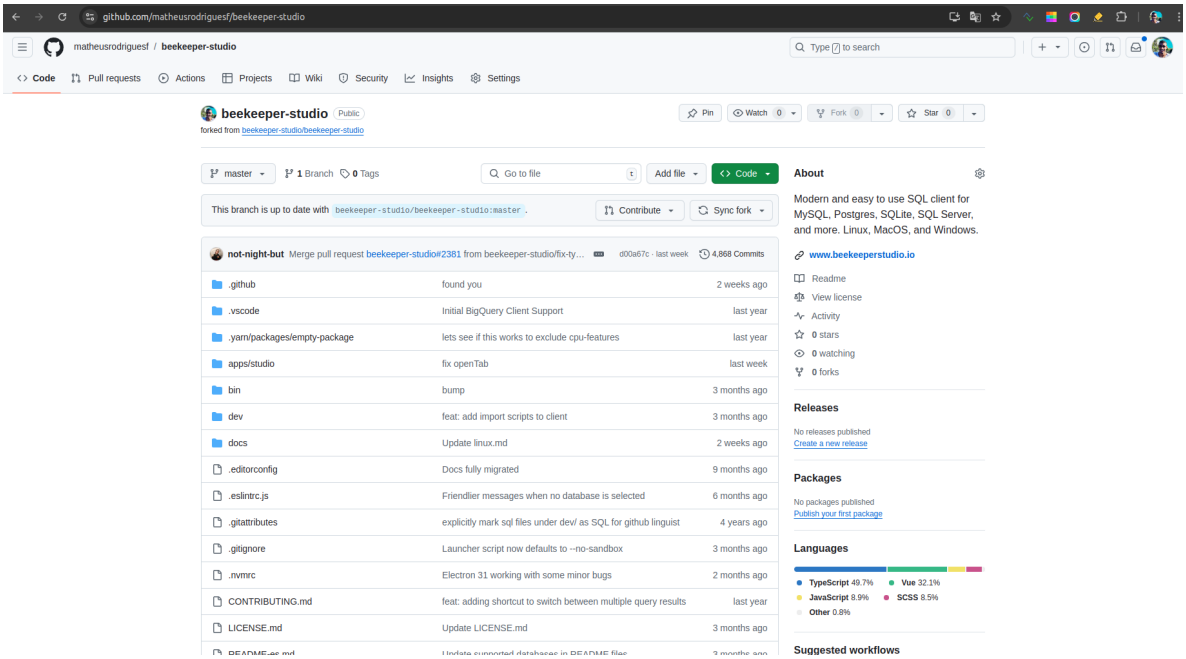

Figura 3: Repositório pessoal após a realização do fork, pronto para ser clonado e modificado conforme necessário.

- Clonagem do repositório

Figura 4: Clonagem do repositório forkado para o ambiente local utilizando o comando `git clone`.

3.2 Estrutura do projeto

O projeto Beekeeper Studio apresenta uma estrutura bem organizada, dividida em vários módulos. O código-fonte principal está localizado na pasta apps/studio/src, que contém os arquivos JavaScript e TypeScript responsáveis pela lógica de backend e pelas APIs. Além disso, o projeto conta com diversas bibliotecas auxiliares que são gerenciadas como dependências do Node.js, e o diretório tests para a execução dos testes automatizados, permitindo verificar o funcionamento correto de diferentes partes do sistema. Os seguintes frameworks e bibliotecas são usados no projeto:

- Electron: Utilizado para construir aplicação desktop. Para realizar essas tarefas, os scripts `electron:build` e `electron:serve` são usados.
- SQLTools: Utilizado para funcionalidades relacionadas ao SQL, com scripts específicos como `sqltools:build` e `sqltools:dev` para build e desenvolvimento.
- ESLint: Utilizado para linting do código, garantindo a qualidade e consistência do código. O script `all:lint` executa o linting em todos os workspaces.
- Yarn: Utilizado como gerenciador de pacotes e para scripts de automação. O projeto utiliza workspaces do Yarn para gerenciar múltiplos pacotes dentro da pasta **apps/***.
- Jest: Utilizado para testes automatizados, com scripts como `test:e2e`, `test:integration`, `test:unit` e `test:ci`.
- Markdown: Utilizado para documentação, com o script `docs:serve` para servir a documentação do projeto.

4 Criação do Dockerfile

O Dockerfile é um componente essencial do processo de integração e entrega contínua (CI/CD), pois define como o projeto será empacotado em uma imagem Docker para que o mesmo possa ser executado de forma consistente em vários ambientes. No contexto do projeto, o Dockerfile foi implementado para criar uma imagem funcional do sistema, encapsulando todas as dependências e configurações necessárias.

A estrutura básica do Dockerfile utilizada neste projeto é a seguinte:

```
FROM node:20-bullseye-slim

RUN apt-get update && apt-get install -y \
    libx11-xcb1 \
    libxcb-dri3-0 \
    libxtst6 \
    libnss3 \
    libatk-bridge2.0-0 \
    libgtk-3-0 \
    libxss1 \
    libasound2 \
    libdrm2 \
    libgbm-dev \
    python-is-python3 \
    --no-install-recommends && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

```

```
WORKDIR /app

COPY . .

RUN chown -R node:node /app

USER node

RUN yarn install && npx electron-rebuild

USER root
RUN chmod 4755 /app/node_modules/electron/dist/chrome-sandbox

USER node

CMD ["yarn", "run", "electron:serve", "--no-sandbox"]
```

Explicação das Etapas do Dockerfile:

Este Dockerfile usa como base a imagem `node:20-bullseye-slim`, que é uma versão compacta do Node.js, para criar um ambiente de execução para uma aplicação Electron. A seguir, é detalhado cada passo:

1. Imagem Base A imagem `node:20-bullseye-slim` é uma versão estável e leve do Node.js 20, baseada na distribuição Debian Bullseye. É ideal para aplicações de produção que requerem um ambiente otimizado.

2. Instalação de Dependências do Sistema: São instaladas bibliotecas para o funcionamento das aplicações Electron que dependem de componentes gráficos.

3. Definição do Diretório de Trabalho: Define o diretório de trabalho como `/app`, onde o código da aplicação será copiado e executado.

4. Cópia dos Arquivos do Projeto: Copia todos os arquivos do diretório atual no host para o diretório `/app` dentro do contêiner. Incluindo código-fonte e arquivos de configuração.

5. Ajuste de Permissões: Altera o dono dos arquivos no diretório `/app` para o usuário `node`. Garante que o usuário com menos privilégios possa executar os comandos com segurança.

6. Mudança para o Usuário `node`: Todos os comandos serão executados como o usuário `node` em vez do `root`.

7. Instalação de Dependências e Rebuild do Electron: Instala as dependências da aplicação utilizando o `yarn` e, em seguida, executa o `electron-rebuild`, que é necessário para garantir que os pacotes nativos funcionem corretamente no ambiente Electron.

8. Reajuste Temporário para o Usuário Root: Ajusta as permissões do `chrome-sandbox`, componente necessário para a segurança da execução do Electron, permitindo que ele seja executado corretamente no modo sandbox.

9. Execução Aplicação O container inicia a aplicação Electron utilizando o comando `yarn run electron:serve`, com a `flag --no-sandbox` para evitar problemas de permissão que podem ocorrer no ambiente do contêiner.

4.1 Importância do Dockerfile no Pipeline CI/CD

Na pipeline CI/CD, a etapa de construção da imagem Docker é fundamental, visto que o mesmo garante que a mesma versão da aplicação que foi testada e validada será a mesma utilizada em produção. Isso elimina problemas de inconsistências entre ambientes, como incompatibilidade de dependências ou configurações divergentes.

Através do uso de Docker no pipeline, também é possível implementar o deploy automatizado da aplicação em plataformas de nuvem ou servidores, tornando o processo de entrega contínua mais confiável e ágil.

Para o presente estudo de caso, não será abordado o deploy da aplicação em um servidor ou ambiente de nuvem, visto que o Beekeeper Studio é uma aplicação com ênfase em ser executada em um ambiente de desktop.

5 Criação do Workflow com GitHub Actions

A automação da pipeline CI/CD do projeto Beekeeper Studio foi por meio do GitHub Actions, plataforma nativa do GitHub que permite definir fluxos de trabalho (workflows) baseados em eventos, como o envio de código (push) ou abertura de pull requests. O processo para a criação do workflow é realizado dentro da pasta **.github/workflows/** do repositório. Um arquivo YAML chamado **beekeeper-studio-sistematizacao-ci.yml** foi criado dentro da pasta. Nesse arquivo, descrevem-se todas as etapas do pipeline, desde o checkout do código até a publicação dos artefatos.

GitHub Actions foi escolhido, devido à sua integração com o GitHub, simplicidade de uso e suporte a múltiplos ambientes. Ele oferece diversas ações pré-construídas, que facilitam a configuração de ambientes, execução de testes e outras tarefas comuns. Além disso, a plataforma suporta uma gama de linguagens de programação, incluindo JavaScript e TypeScript, utilizado no beekeeper studio, aumentando a eficiência do processo de automação e permitindo que todas as etapas do ciclo de desenvolvimento sejam executadas diretamente no repositório.

O workflow foi configurado para ser disparado automaticamente ao detectar eventos de `commits` na branch **master** ou na criação de `pull requests`, garantindo que todas as alterações sejam testadas e validadas antes de serem integradas ao projeto principal.

6 Etapas do Pipeline

O arquivo YAML contendo o pipeline CI/CD do projeto Beekeeper Studio inclui várias etapas automatizadas. Disposto em quatro tarefas principais: construção, teste, análise e publicação. Cada etapa desempenha um papel crucial para manter a qualidade do código e garantir a entrega contínua.

Nas subseções a seguir, são apresentadas as descrições e os passos de cada etapa da pipeline.

6.1 Job: Build

A primeira etapa do pipeline é o job de **build**, que é responsável pela compilação do projeto. Esse job roda em um ambiente Ubuntu e segue os passos abaixo:

- Checkout do código: Utilizando a ação `actions/checkout@v4`, o pipeline clona o repositório para que os arquivos estejam disponíveis para as próximas etapas.
- Configuração do Node.js: A ação `actions/setup-node@v4` configura a versão 20 do Node.js, necessária para rodar as dependências do projeto.
- Cache de módulos Yarn: Para otimizar a execução do pipeline, a ação `actions/cache@v3` armazena os módulos instalados pelo Yarn, evitando reinstalações desnecessárias.
- Instalação de dependências: O comando `yarn install` instala as dependências listadas no arquivo `package.json`.
- Build do projeto: O comando `yarn electron:build` compila o projeto Electron e gera os pacotes necessários para a aplicação.

Abaixo é apresentado o trecho do job:

```yaml
build:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: 20
    - name: Cache Yarn modules
      uses: actions/cache@v3
      with:
        path: ~/.cache/yarn
        key: ${{ runner.os }}-yarn-${{ hashFiles('**/yarn.lock') }}
        restore-keys: |
          ${{ runner.os }}-yarn-
    - run: yarn install
    - run: yarn electron:build
      env:
        GH_TOKEN: ${{ secrets.GH_TOKEN }}
```

6.2 Job: Test

O job de **test** depende da execução bem-sucedida da etapa **build** e é responsável por rodar os testes automatizados do projeto. Ele segue uma estrutura semelhante, com as seguintes etapas:

- Checkout do código e configuração do Node.js: As mesmas etapas de checkout e configuração do Node.js são realizadas.
- Cache de módulos Yarn: O cache dos módulos Yarn é restaurado para otimizar o tempo de execução.
- Instalação de dependências: O comando `yarn install` é executado novamente para garantir que as dependências estejam disponíveis.
- Execução dos testes: O comando `yarn test:ci` executa os testes de integração e unitários em um ambiente contínuo, validando que o código está funcionando corretamente.

Abaixo é apresentado o trecho do job:

```yaml
test:
  needs: build
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-node@v4
      with:
        node-version: 20
    - name: Cache Yarn modules
      uses: actions/cache@v3
      with:
        path: ~/.cache/yarn
        key: ${{ runner.os }}-yarn-${{ hashFiles('**/yarn.lock') }}
        restore-keys: |
          ${{ runner.os }}-yarn-
    - run: yarn install
    - run: yarn test:ci
```

6.3 Job: Analyze

O job **analyze** realiza uma análise estática do código, utilizando a ferramenta **CodeQL** e apresenta as seguintes etapas:

- Análise por linguagem: O job é configurado para rodar em diferentes ambientes e linguagem. Neste caso, o job é configurado para JavaScript/TypeScript.
- Inicialização do CodeQL: A ação `github/codeql-action/init@v3` é utilizada para configurar a análise de código estático.
- Execução do CodeQL: O job executa a análise de segurança e qualidade de código, buscando possíveis vulnerabilidades e problemas.
- Upload dos resultados: Os resultados da análise são armazenados utilizando a ação `actions/upload-artifact@v3`, permitindo revisão dos relatórios.

Abaixo é apresentado o trecho do job:

```yaml
analyze:
  needs: build
  name: Analyze (${{ matrix.language }})
  runs-on: ${{ (matrix.language == 'swift' && 'macos-latest') || 'ubuntu-latest' }}
  permissions:
    security-events: write
    packages: read
    actions: read
    contents: read
  strategy:
    fail-fast: false
    matrix:
      include:
      - language: javascript-typescript
        build-mode: none
  steps:
    - name: Checkout repository
      uses: actions/checkout@v4
    - name: Initialize CodeQL
      uses: github/codeql-action/init@v3
      with:
        languages: ${{ matrix.language }}
        build-mode: ${{ matrix.build-mode }}
    - if: matrix.build-mode == 'manual'
      shell: bash
      run: |
        echo 'If you are using a "manual" build mode for one or more of the' \
          'languages you are analyzing, replace this with the commands to build' \
          'your code, for example:'
        echo '  make bootstrap'
        echo '  make release'
        exit 1
    - name: Perform CodeQL Analysis
      uses: github/codeql-action/analyze@v3
      with:
        category: "/language:${{matrix.language}}"
    - name: Upload CodeQL Results
      uses: actions/upload-artifact@v3
      with:
        name: codeql-results
        path: codeql-results/
```

6.4 Job: Publish

O job **publish** é executado após a conclusão bem-sucedida das etapas anteriores. Este job automatiza a publicação da imagem Docker do projeto. As etapas incluem:

- **Login no Docker Hub**: Utilizando as credenciais armazenadas como segredos no GitHub `DOCKER_HUB_USERNAME` e `DOCKER_HUB_TOKEN`, o pipeline faz login no Docker Hub para preparar o envio da imagem.
- **Build e publicação da imagem Docker**: A ação `docker/build-push-action@v2` constrói a imagem Docker do Beekeeper Studio e a publica no Docker Hub com duas tags: uma baseada no hash do commit `${{ github.sha }}` e outra com a tag `latest`.

Abaixo é apresentado o trecho do job:

```yaml
  publish:
    needs: [test, analyze]
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Login no Docker Hub
        uses: docker/login-action@v1
        with:
          username: ${{ secrets.DOCKER_HUB_USERNAME }}
          password: ${{ secrets.DOCKER_HUB_TOKEN }}

      - name: Construir e publicar imagem Docker
        uses: docker/build-push-action@v2
        with:
          context: .
          push: true
          tags: |
            ${{ secrets.DOCKER_HUB_USERNAME }}/beekeeper-studio:${{ github.sha
                }}
            ${{ secrets.DOCKER_HUB_USERNAME }}/beekeeper-studio:latest
```

7 Etapa de Deploy: Execução Local do Beekeeper Studio via Docker

Após a publicação da imagem Docker no Docker Hub, é possível executar o Beekeeper Studio localmente. Sem precisar compilar o código ou configurar manualmente o ambiente, o usuário pode baixar a imagem publicada diretamente do Docker Hub e executá-la em seu ambiente local.

7.1 Baixando a Imagem e Rodando o Container

Para baixar a imagem do Beekeeper Studio no repositório docker execulte o seguinte comando:

```
docker pull matheusrfa/beekeeper-studio:latest
```

Após executar o comando, para baixar a image Docker é aguardado o seguinte resultado como saída:

Figura 5: Download Image Docker Beekeeper Studio

Após baixar a imagem, o container pode ser iniciado com o seguinte comando:

```
docker run --name beekeeper-studio \
    --privileged \
    -e DISPLAY=$DISPLAY \
    -v /tmp/.X11-unix:/tmp/.X11-unix \
    -v $HOME/.Xauthority:/root/.Xauthority \
    --net=host \
    matheusrfa/beekeeper-studio:latest
```

Ao executar o comando para rodar o container é esperado o seguinte resultado como saída:

Figura 6: Rodando o container Beekeeper Studio

Caso o container tenha sido iniciado com sucesso é esperado o seguinte saída:

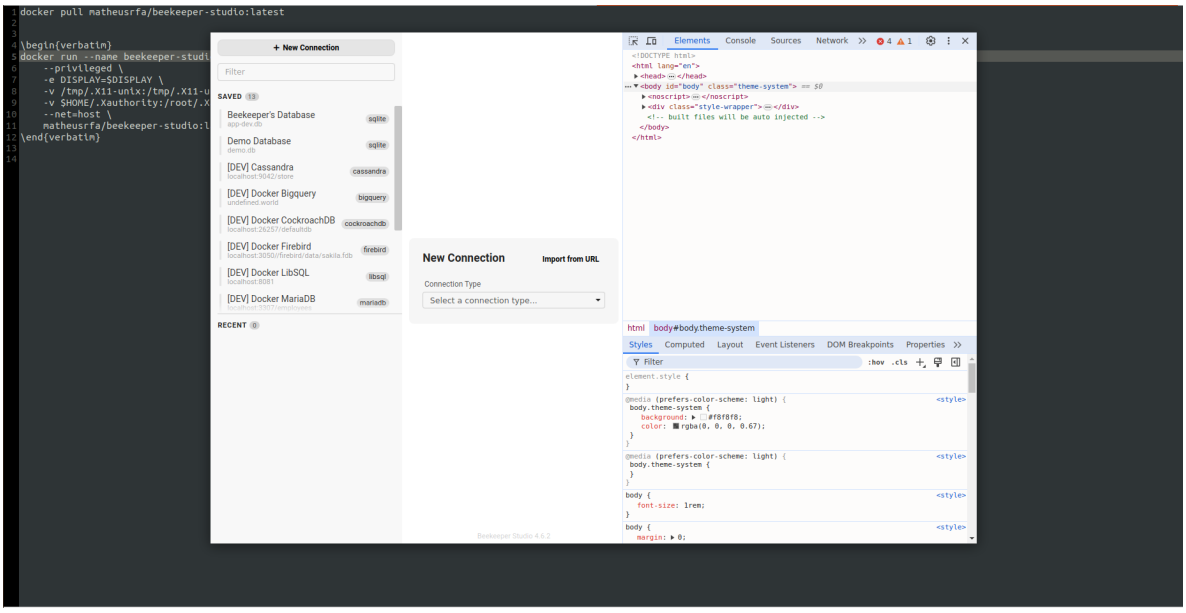

Figura 7: Interface gráfica Beekeeper Studio

7.2 Vantagens do Deploy Local via Docker

A abordagem de deploy via docker permite que os desenvolvedores e usuários, utilizem a aplicação sem a necessidade de configurar o ambiente de desenvolvimento ou instalar dependências. A execução em um container garante um ambiente isolado e preparado para uso.

8 Desafios Enfrentados e Soluções Implementadas

Um dos principais desafios enfrentados foi a configuração inicial do pipeline CI/CD, especialmente quando se tratava de integrar várias tecnologias, como Docker e Node.js. A solução foi usar ações específicas do GitHub, como `actions/setup-node` para configurar o ambiente Node.js e `docker/build-push-action` para construção e publicação de imagens Docker. Permitindo criar um pipeline modular e flexível.

Outro desafio foi garantir que o pipeline fosse executado de maneira eficiente, minimizando o tempo de execução sem comprometer a qualidade. A implementação do cache de dependências usando o `actions/cache` como solução reduziu o tempo de build e testes ao reutilizar pacotes já instalados anteriormente.

9 Melhorias Propostas

Durante o desenvolvimento do pipeline CI/CD, algumas melhorias foram identificadas para otimizar o fluxo de integração e entrega contínua. A primeira melhoria seria a introdução de uma etapa de análise de segurança usando o CodeQL. Essa etapa verifica o código em busca de vulnerabilidades e problemas de segurança potenciais. Essa etapa aumenta a confiança no código, especialmente em projetos open source, onde a segurança é um problema constante.

A implementação de um sistema de cache para dependências do projeto, como pacotes do Yarn, foi outra melhoria abordada. O tempo de execução do pipeline reduziu significativamente, visto que as dependências não precisavam ser baixadas a cada nova execução. Além disso, foi implementado o uso de um sistema de deploy automatizado, que cria e publica imagens Docker, o que facilita a implantação da aplicação em vários ambientes, garantindo que ela seja compatível com os ambientes de desenvolvimento, teste e produção.

10 Reflexão Pessoal

Ao desenvolver este pipeline CI/CD, percebi a importância de automatizar processos repetitivos e críticos para garantir a qualidade do software e agilidade na entrega. A implementação de um pipeline robusto trouxe um grande aprendizado e valor da integração contínua na verificação do código por meio de testes e na detecção de vulnerabilidades por meio da análise estática com CodeQL. Esse processo reforçou minha compreensão sobre boas práticas de DevOps e a necessidade de manter um ciclo de desenvolvimento mais rápido e confiável.

A adoção de ferramentas como o GitHub Actions facilita a integração com diversas tecnologias e serviços. A ideia de que um processo bem organizado tem um impacto direto na produtividade da equipe e na qualidade do produto final foi fortalecida pela facilidade de configuração do pipeline e pela capacidade de adicionar novas etapas.

Por fim, a portabilidade e o versionamento consistente tornaram-se cada vez mais cruciais em ambientes de produção ao trabalhar com tecnologias como Docker e automatizar a publicação de imagens. Essas práticas garantem que o software seja instalado de forma segura e eficiente, com previsibilidade e controle de qualidade, mesmo em projetos e equipes dispersas.

11 Conclusão

Este estudo de caso examinou como configurar um pipeline CI/CD usando o GitHub Actions para construir, testar, analisar e publicar uma aplicação dockerizada chamada Beekeeper Studio. A implementação deste pipeline mostrou a importância e os benefícios da integração contínua e da entrega contínua em projetos de software e demonstrou como a automação pode garantir a qualidade do código e otimizar o fluxo de trabalho.

A capacidade do pipeline de garantir que cada alteração no código fosse automaticamente verificada por meio de testes e análises e garantir que as imagens Docker fossem construídas e publicadas de forma confiável foi demonstrada por meio das etapas descritas. O uso das Actions do GitHub facilitou a integração com o Docker Hub e a automação das tarefas críticas do desenvolvimento de software.

O estudo também mostrou a escalabilidade e a flexibilidade das práticas de CI/CD, que permitem o desenvolvimento de software mais ágil e com menos probabilidade de erros. A configuração do pipeline também enfatiza a necessidade de uma abordagem bem estruturada e documentada para a automação de processos, pois facilita a escalabilidade e a manutenção das soluções adotadas.

Por último, a inclusão de imagens e exemplos detalhados no documento fornece uma visão útil das etapas necessárias para configurar o pipeline CI/CD. Este estudo de caso pode servir como exemplo para outros projetos, fornecendo informações úteis sobre como realizar uma estratégia de integração e entrega contínua eficiente.

Referências

1. **GitHub - CodeQL Action:** Repositório oficial da ação CodeQL para análise de código no GitHub Actions. Disponível em: https://github.com/github/codeql-action. Acesso em: 15 Set. 2024.

2. **GitHub - Electron Builder Action:** Ação do GitHub Marketplace para construir aplicações Electron. Disponível em: https://github.com/marketplace/actions/electron-builder-action. Acesso em: 15 Set. 2024.

3. **GitHub Documentation:** Criação de ações Docker para o GitHub Actions. Disponível em: https://docs.github.com/pt/actions/sharing-automations/creating-actions/creating-a-docker-contai Acesso em: 15 Set. 2024.

4. **R. Groffe,** "Docker e GitHub Actions - Parte 1: Build automatizado de aplicações,"Medium, 2022. Disponível em: https://renatogroffe.medium.com/docker-github-actions-parte-1-build-automatiza C3%A7%C3%B5es-7346f04c7f4e. Acesso em: 15 Set. 2024.

5. **Beekeeper Studio Documentation:** Documentação oficial do Beekeeper Studio. Disponível em: https://docs.beekeeperstudio.io/. Acesso em: 15 Set. 2024.

12 Código Fonte

O código fonte para este projeto está disponível no GitHub. Você pode acessar o repositório através do seguinte link:

https://github.com/matheusrodriguesf/beekeeper-studio