Criando uma pipeline

Antes de mais nada, é preciso entender um pouco melhor a estrutura de um script de pipeline pois é à partir deste script que a nossa pipeline "nasce" no Gitlab CI/CD, então a primeira coisa a notarmos é que o script segue um "padrão de linguagem".

Aqui estamos falando do YAML que segundo a definição do próprio site https://yaml.org/, e transcrito por mim para melhor seu entendimento, indica que o YAML é uma "linguagem" fácil de ser lida por humanos que possibilita a transformação dessas informações contidas no arquivo com a linguagem YAML para qualquer linguagem de programação.

O YAML tornou-se um padrão de mercado simples, menos verboso que um XML por exemplo, de fácil entendimento por pessoas e que não te obriga a conhecer outras linguagens de programação para entregar valor.

Há toda uma especificação completa encontrada em https://yaml.org/spec/1.2.2/, só que aqui nós ficaremos no básico, ou seja, não precisaremos percorrer por todas as possibilidades que o YAML nos traz.

O que precisamos saber é:

- Nós podemos fazer um mapeamento de uma chave e valor tão simples quanto indicar, por exemplo:

```
NomeDaAplicacao: teste
```

- No exemplo acima temos a chave chamada "NomeDaAplicacao" e o valor desta chave que é "teste"

- Também podemos ter uma lista de itens, para indicar uma lista nós incluímos um traço "-" à frente, como no seguinte exemplo:

- Item 1

- Item 2

- Item 3

- Podemos ter também um valor que é uma lista, como por exemplo:

Brinquedos: [Bola, Carrinho, Boneca]

- E também sequências de chave-valor que pertençam à uma chave "pai":

variaveis:

    variavel1: valor1

    variavel2: valor2

    variavel3: valor3

Com essas simples definições e mais algumas que veremos, conseguiremos construir nosso script usando YAML para que fique algo legível para nós (humanos) e suficiente para que uma pipeline seja criada no repositório de teste do Gitlab que criamos.

Agora que entendemos o que é um YAML, vamos entender como criar nossa pipeline do Gitlab CI/CD, há uma estrutura que devemos respeitar para que o Gitlab CI/CD também consiga entender nosso YAML e estruturar a pipeline à partir disso.

Temos uma referência completa em https://docs.gitlab.com/ee/ci/yaml/index.html à qual começaremos pelo

seguinte:

stages

A chave **stages** indica que iremos organizar nossa pipeline em estágios distintos, é uma forma de conseguirmos separar o que será feito caso você precise apartar um fluxo de **test**, de **build** e de **deploy**, esses são os stages de exemplo para melhor entendimento. Se você não apartar o Gitlab CI/CD tentará rodar tudo em paralelo, o que pode não ser uma boa estratégia. Mas é bom lembrar que as **stages** são opcionais!

stages:

- test

- build

- deploy

No exemplo acima, se eu defino 3 estágios distintos, eles rodarão sequencialmente na ordem em que se encontram, neste exemplo serão todos os jobs de test, depois todos de build e depois todos de deploy.

Vamos usar o script do nosso repositório de apoio https://gitlab.com/everton.juniti/descomplica, o script está na pasta cicd_generico\01-Primeiro_pipeline, o nome do arquivo é .gitlab-ci.yml

Vejamos as primeiras 2 linhas do arquivo:

```
stages:
    - deploy
```

Neste exemplo estou indicando apenas um estágio, que será **deploy**.

O que vier abaixo desta chave stages, depois da lista (no nosso caso temos só o item **deploy** na lista), serão o que chamamos de jobs.

Um job é um trabalho a ser feito pelo runner do pipeline, o runner irá executar tantos jobs quanto forem definidos, em nosso exemplo temos 2:

```
Implantacao_Nginx:
    stage: deploy
    script:
        docker run -d --name Nginx_teste -p 3080:80 nginx:latest

Implantacao_Nginx_2:
    stage: deploy
    script:
        docker run -d --name Nginx_teste_2 -p 3180:80 nginx:latest
```

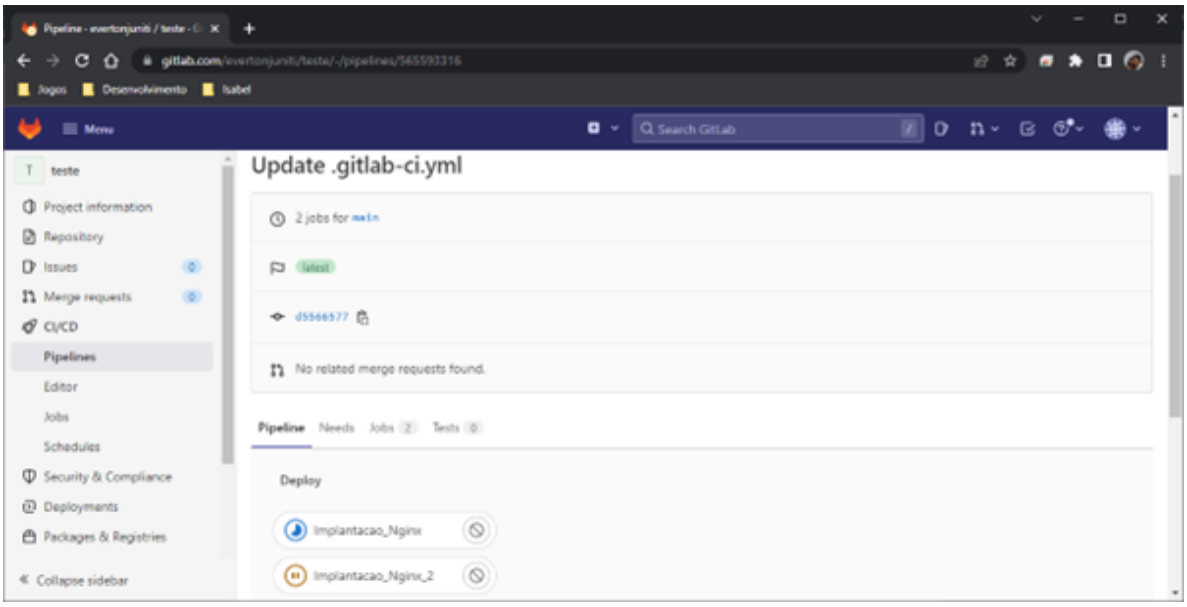
O nome dos nossos jobs são "Implantacao_Nginx" e "Implantacao_Nginx_2", usamos esses nomes de job apenas para organização e melhor visualização do que está acontecendo na nossa pipeline.

Veja que há uma "identação" de tudo que está abaixo de cada job, assim sabemos que tanto a chave **stage** quanto a chave **script** declarados neste exemplo pertencem a estes jobs.

A chave **stage** dentro de cada job indica à qual stage este job pertencerá, no nosso exemplo temos apenas 1, mas quando há mais de 1 (lembrando que cada stage executa sequencialmente), indicamos qual job é de qual stage aqui.

A chave **script** veremos depois, o que você precisa saber é que esta chave compõe cada job.

Ao salvarmos este script no nosso repositório de testes, vemos que os 2 jobs são executados, mas neste momento observamos que é executado 1 job por vez:



Só que não é isso que realmente gostaríamos, queremos ver rodando em paralelo!

Para isso, nós teremos que alterar uma configuração no nosso runner, que está rodando em nosso Docker.

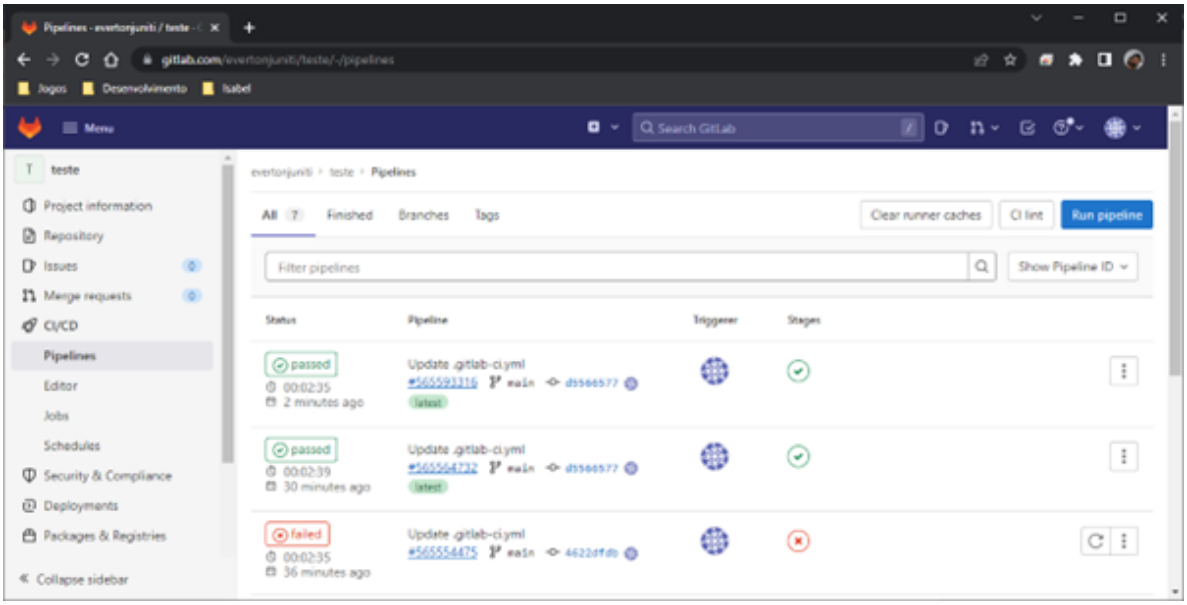
Lembra que ao configurarmos nosso runner foi indicado, no exemplo aqui, que seria utilizado um volume com um arquivo de configuração? No meu exemplo aqui seria: D:\docker\volumes\gitlab-runner\config\config.toml, eu vou abrir este arquivo em um editor (como o bloco de notas) para alterar a seguinte linha:

De: concurrent = 1

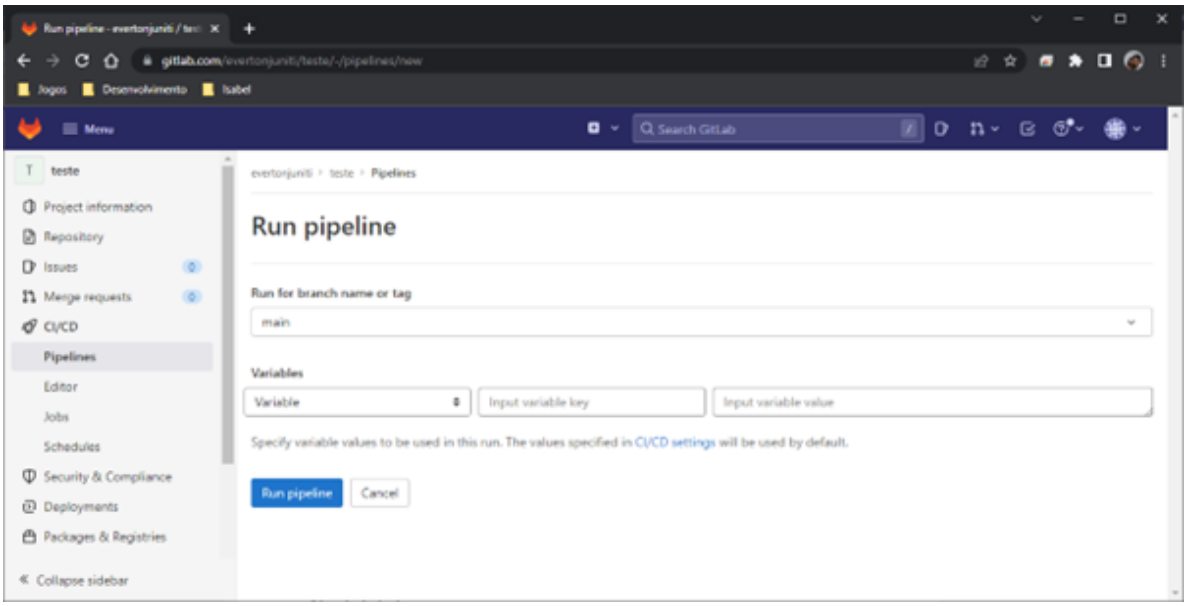
Para: concurrent = 4

Ao fazer isso nós estamos indicando que este runner nosso rodará até 4 jobs em paralelo, bastará salvar o arquivo e pronto! A configuração nova é sensibilizada no runner a cada 3 segundos, então não precisamos nem parar e iniciar novamente o container do runner. Vamos só excluir os containers do Nginx recém criados e iremos rodar novamente o pipeline.

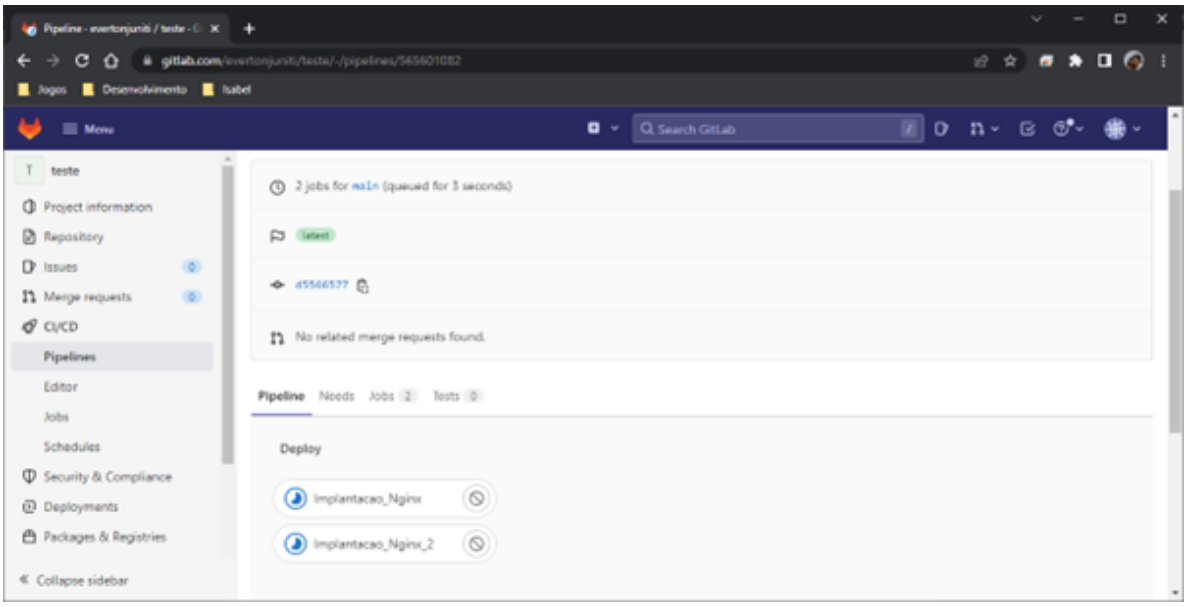
Para executar novamente a pipeline, vá no seu repositório de testes e clique no menu à esquerda em CI/CD -> Pipelines, depois clique no botão “Run pipeline”:



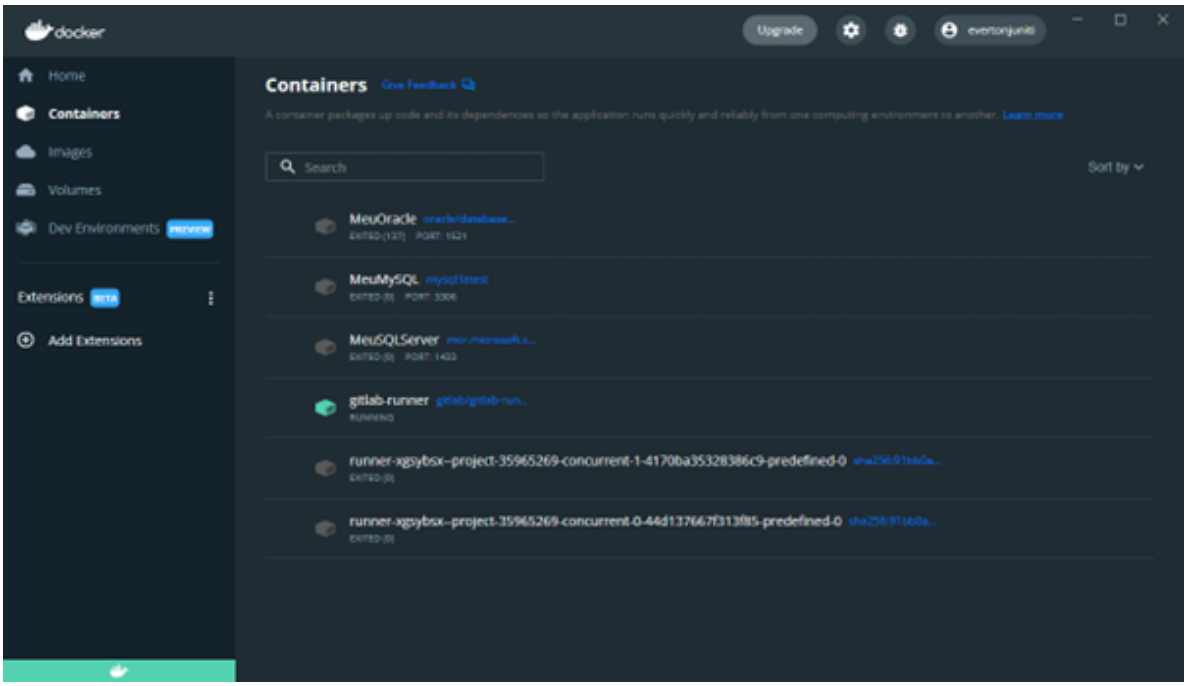
Na tela que surgir, não se preocupe com as opções, apenas clique no botão “Run pipeline”



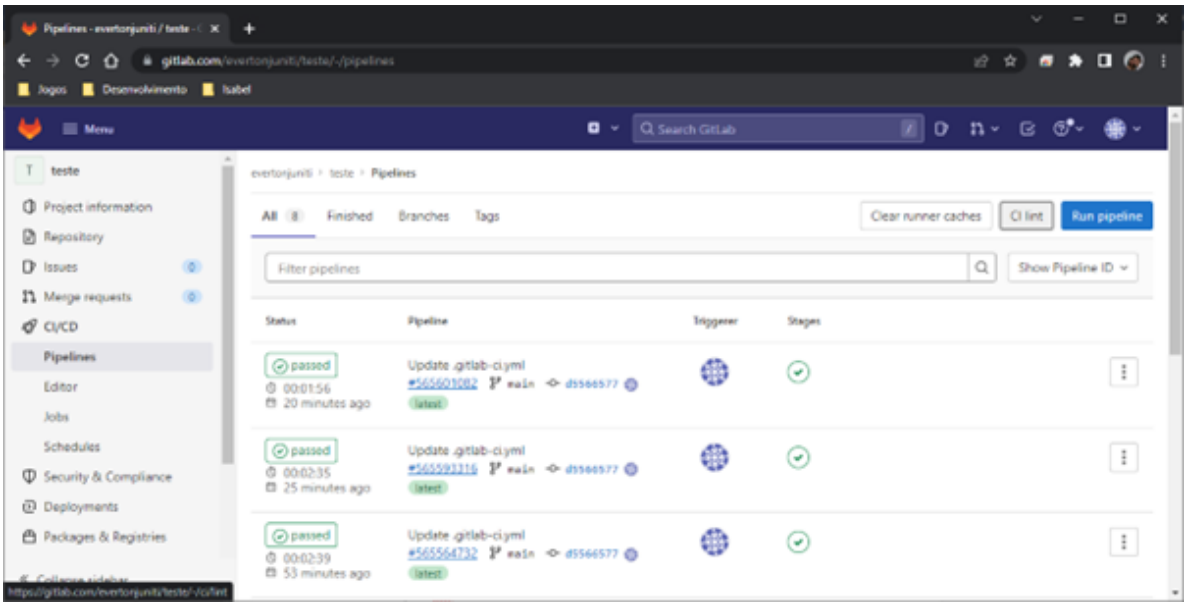
Agora sim veremos os 2 jobs do nosso script (Implantacao_Nginx e Implantacao_Nginx_2) rodando em paralelo:



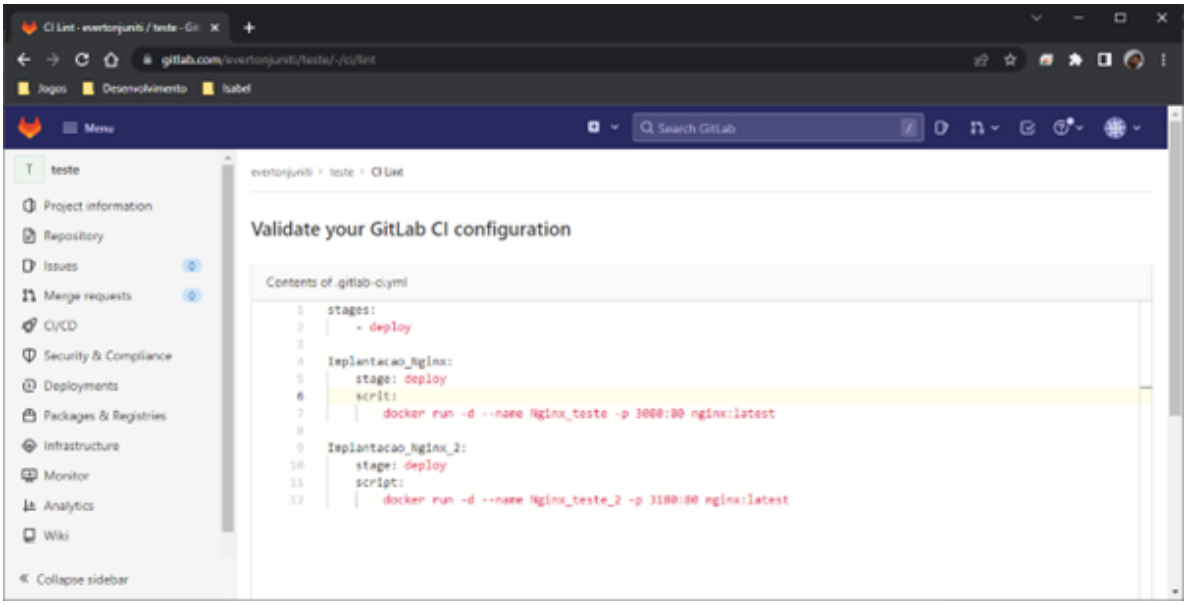
Veja no Docker Desktop que aparecem 2 containers temporários que indicam a concorrência de execução dos 2 jobs (olhe o nome do container, concurrent-0 e concurrent-1):



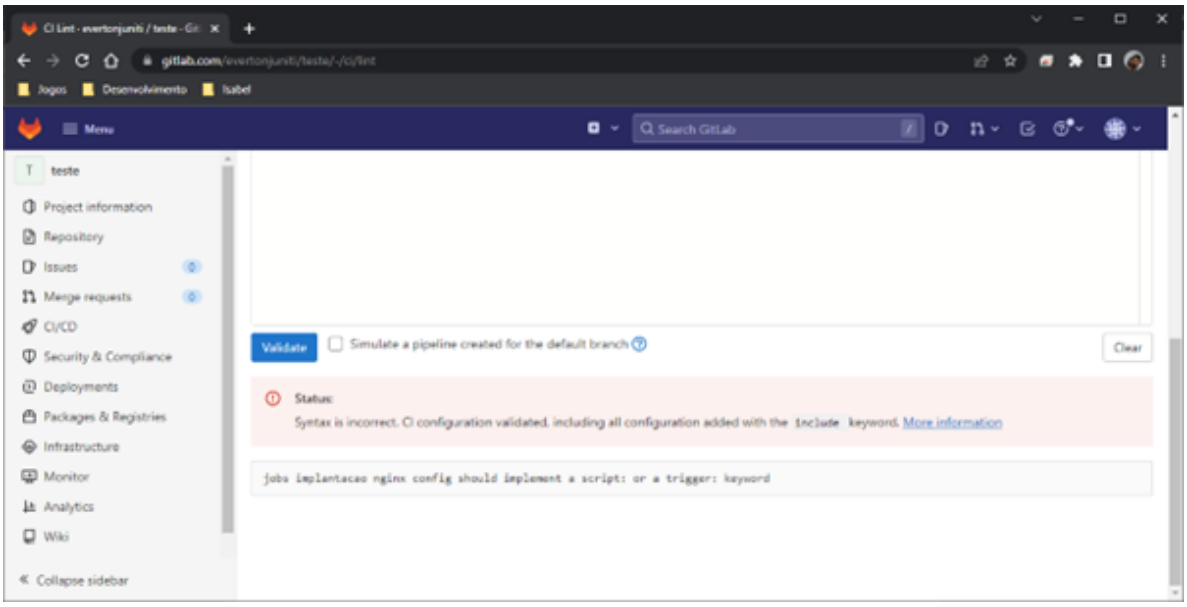
Tá bom, aqui vimos o “caminho feliz”, mas e se errarmos algo no script? É aí que vem o “LINT”. Com o “CI lint” do Gitlab nós podemos validar se o script está escrito corretamente, para isto vá em seu repositório de teste e clique no menu à esquerda CI/CD -> Pipelines e na tela que surgir clique no botão “CI lint”:



Na tela que surgir coloque uma cópia do script de exemplo, só que altere a palavra “script” por “scrit” em qualquer lugar, propositadamente:



Por último clique no botão “Validate” e observe o resultado:



Esta validação sempre será feita ao alterar o script de uma pipeline no repositório, mas para que você não corra o risco de ter que fazer vários "push" de código do script, a validação do "CI lint" é uma boa opção para que você consiga verificar isso antes de mandar a pipeline pra frente! Esta validação é necessária para que a pipeline "não esteja quebrada" antes mesmo do runner tentar executar o conteúdo do script.

Vamos voltar um pouco no nosso script de exemplo. A chave **script** é específico de um job, indicamos aqui o que o job deverá fazer e no nosso caso como o runner que estamos utilizando é baseado no Docker, o script indica o comando do Docker que iremos executar.

No nosso exemplo atual, só há 1 único script em cada job, mas se fosse necessário executar mais de um comando, poderíamos ter uma lista de scripts, desta forma:

```
Job_exemplo:

    stage: deploy

script:

    - comando 1
```

- comando n

Em nosso exemplo temos isso aqui:

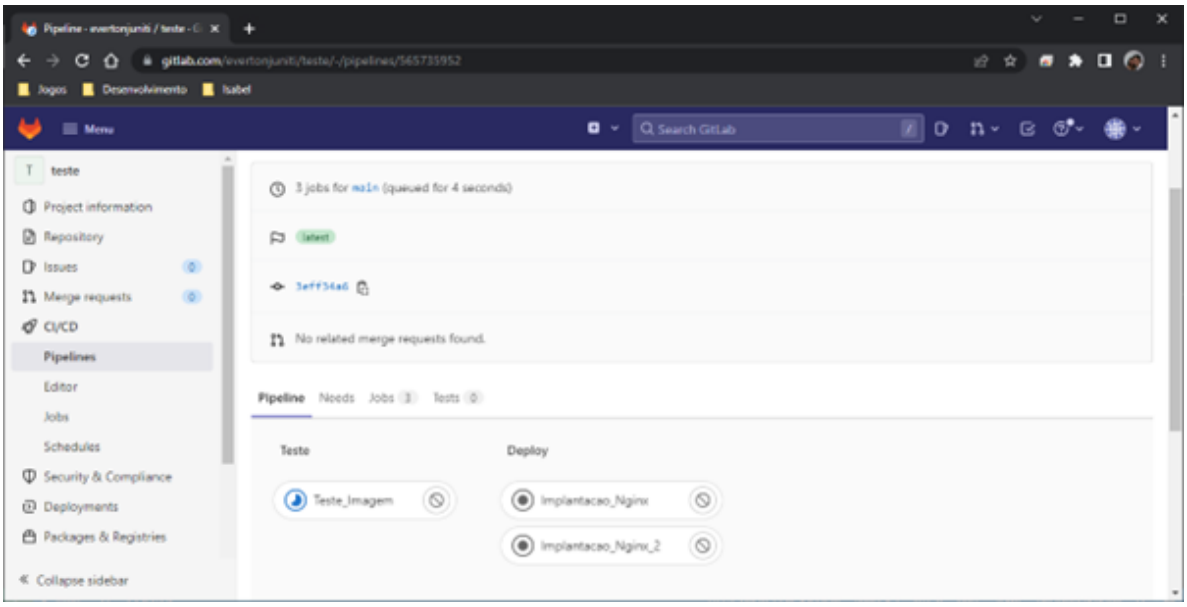
```
    script:
        docker run -d --name Nginx_teste -p 3080:80 nginx:latest
```

Como nosso runner usa uma imagem base do Docker, podemos executar comandos do Docker como um docker run que cria um container. Podemos executar diferentes comandos sempre baseados no que a imagem nos oferece.

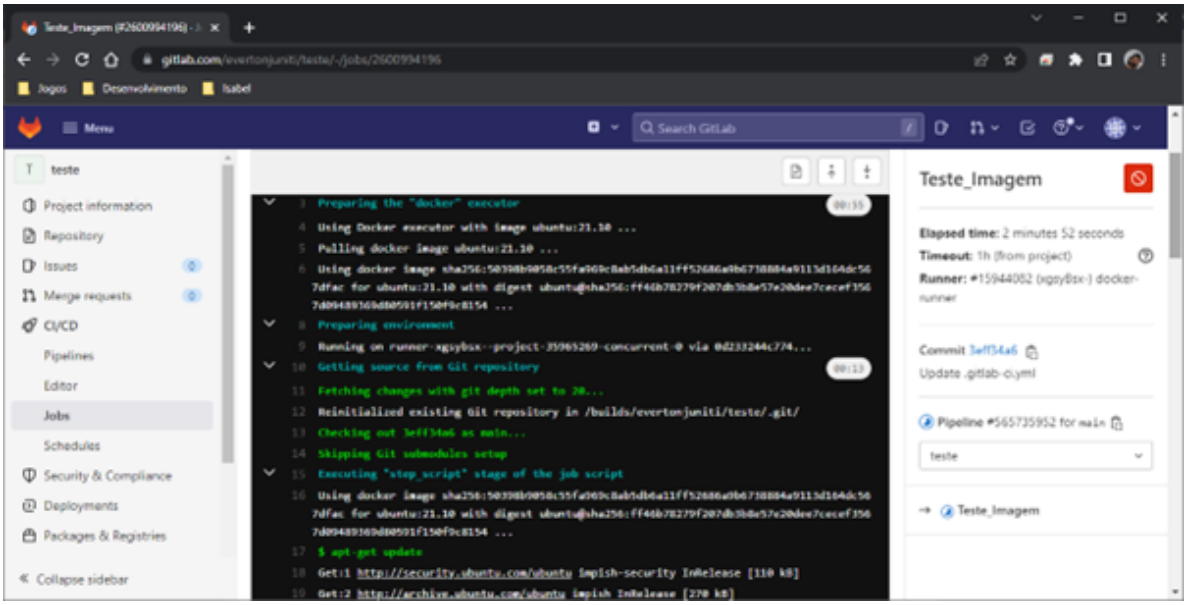
Vamos olhar o exemplo do arquivo .gitlab-ci.yml presente neste repositório, na pasta cicd_generico\02-Imagem:

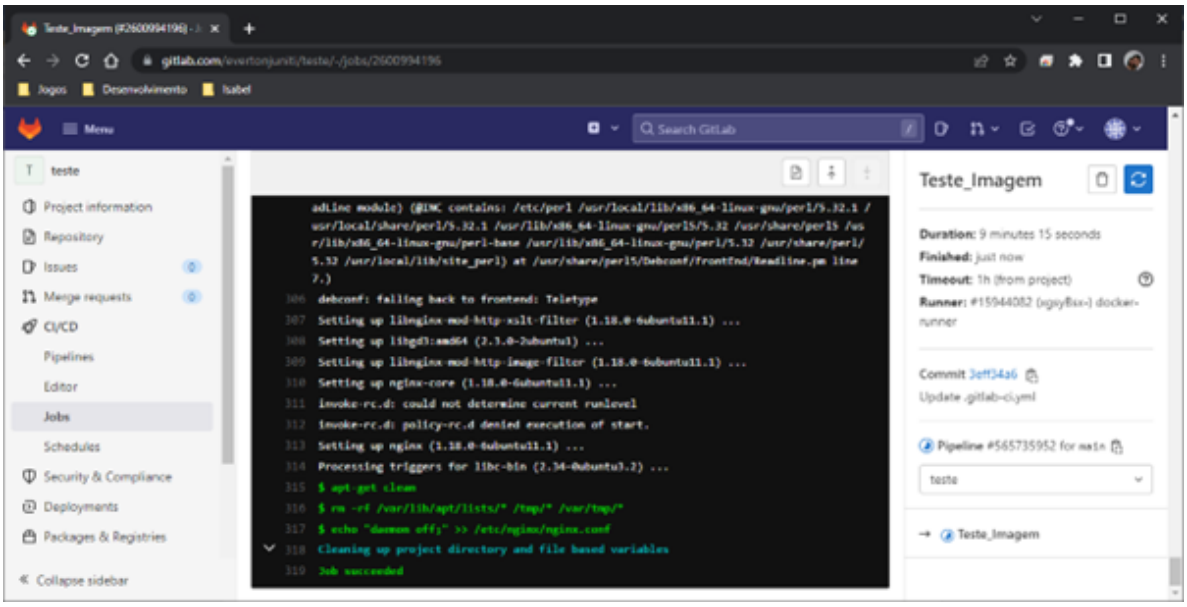
```
stages:
    - teste
    - deploy

Teste_Imagem:
    stage: teste
    image: ubuntu:21.10
    script:
        - apt-get update
        - apt-get install -y nginx
        - apt-get clean
        - rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
        - echo "daemon off;" >> /etc/nginx/nginx.conf
```

Aqui adicionamos um **stage** de teste (para mostrar que os **stages** podem ter qualquer nome) e criamos um job atrelado a este **stage** de teste. Este job puxa uma imagem base (Ubuntu Linux 21.10) e executa alguns comandos em cima desta imagem, olhe como fica o pipeline:

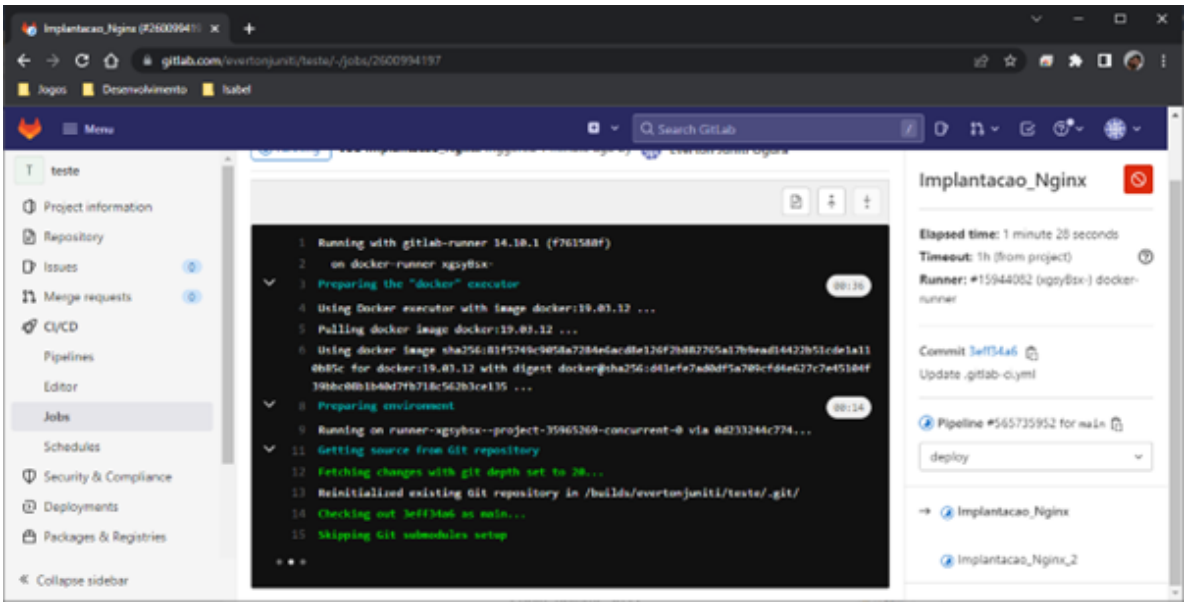


Vejamos no log de execução da pipeline o que é feito: o runner usa o Docker por detrás dos panos para fazer toda a execução, e nesse job de teste nós indicamos que o executor irá usar uma imagem do Ubuntu. Logo após puxar do Docker Hub essa imagem base, são executados os comandos que incluímos no script para executar “em cima” dessa imagem base do Ubuntu. Vamos olhar os logs:





Olhando para os jobs de deploy, nós não alteramos a imagem base, então manteve-se a imagem base padrão que cadastramos ao registrar o runner no nosso repositório, no caso é a imagem base do docker:19.03.12



O **stage** de teste não faz nada no final das contas para este script de pipeline de teste, mas mostramos que podemos usar imagens base distintas para execuções entre os **stages**, podemos indicar especificamente qual imagem base queremos utilizar para executar os comandos do **script** ou podemos simplesmente omitir e deixar usa a imagem base padrão que cadastramos ao registrar nosso runner.

Atividade Extra

Para se aprofundar no assunto desta aula leia o documento de referência: “Jobs”. Neste material há um detalhamento maior do que pode compor um job numa pipeline.

Link do documento: https://docs.gitlab.com/ee/ci/jobs/

Referência Bibliográfica

OGURA, Everton J. Repositório do GitLab, projeto Descomplica. Disponível em https://gitlab.com/everton.juniti/descomplica. Acesso em 16 de junho de 2022.

YAML: YAML Ain’t Markup Language. Disponível em https://yaml.org/. Acesso em 16 de junho de 2022.

YAML Ain’t Markup Language (YAML™) version 1.2. Disponível em https://yaml.org/spec/1.2.2/. Acesso em 16 de junho de 2022.

.gitlab-ci.yml keyword reference. Disponível em https://docs.gitlab.com/ee/ci/yaml/index.html. Acesso em 16 de junho de 2022.

Validate GitLab CI/CD configuration. Disponível em https://docs.gitlab.com/ee/ci/lint.html. Acesso em 16 de junho de 2022.

Advanced configuration. Disponível em https://docs.gitlab.com/runner/configuration/advanced-configuration.html. Acesso em 16 de junho de 2022.

Ir para exercício