Ambientes

Depois que subimos nossa solução de ponta-a-ponta, vamos agora olhar para o todo só que com outra perspectiva.

Imagine que você estivesse trabalhando com uma solução como esta em uma empresa, só que você tem coleguinhas trabalhando em conjunto no mesmo repositório (sim, isso é comum). Como é que você faria para organizar o que você está fazendo sem atrapalhar o coleguinha?

É aí que vem as branches do Gitlab.

As branches (galhos em tradução livre) são como se fossem cópias do que estiver no repositório para que várias pessoas consigam trabalhar de forma colaborativa, em paralelo.

Isso significa que eu teria uma branch para que eu consiga trabalhar e você teria uma outra branch (diferente da minha) para que você trabalhe. Ao final nós faríamos o que chamamos de merge (mesclagem) do que produzimos para termos uma única solução.

Até agora nós brincamos apenas com 2 branches: a main (que o repositório já cria automaticamente) e a Release (que nós criamos para testar os fluxos da pipeline), mas podemos organizar de outras formas.

Pensando no dia-a-dia de trabalho, o mais comum é vermos uma organização de branches como à seguir:

- Main / Master / Prod / Production / Produção: geralmente essas nomenclaturas de branches representam o código que vai para produção, é onde o negócio da empresa disponibiliza seus serviços aos clientes/parceiros.

- Staging / QA / Hom / Homologation / Homologação: geralmente essas nomenclaturas de branches representam o código que vai para homologação, para ser testado pelo time de qualidade ou por um time de negócios, antes de ir para o ambiente de produção.

- Release / Dev / Development / Desenvolvimento: geralmente essas nomenclaturas de branches representam o código que vai para um ambiente de desenvolvimento, para testes do desenvolvedor antes de ir para os testes de QA ou negócios.

- Feature: geralmente essa nomenclatura de branch representa o código que é cópia da branch de desenvolvimento só que específico para algo que alguém esteja trabalhando, é onde o desenvolvedor faz o clone e trabalha.

- Emergency / HotFix: geralmente essas nomenclaturas de branches representam o código que está em ambiente de produção, só que ao identificar um problema geralmente acompanhado de um incidente, é indicado para que se pule outras etapas (como homologação) a fim de restabelecer o ambiente produtivo à sua normalidade.

A organização do repositório em várias branches até aqui não impacta em nada a pipeline, mas podemos usar um atributo especial chamado “Environment” no nosso script.

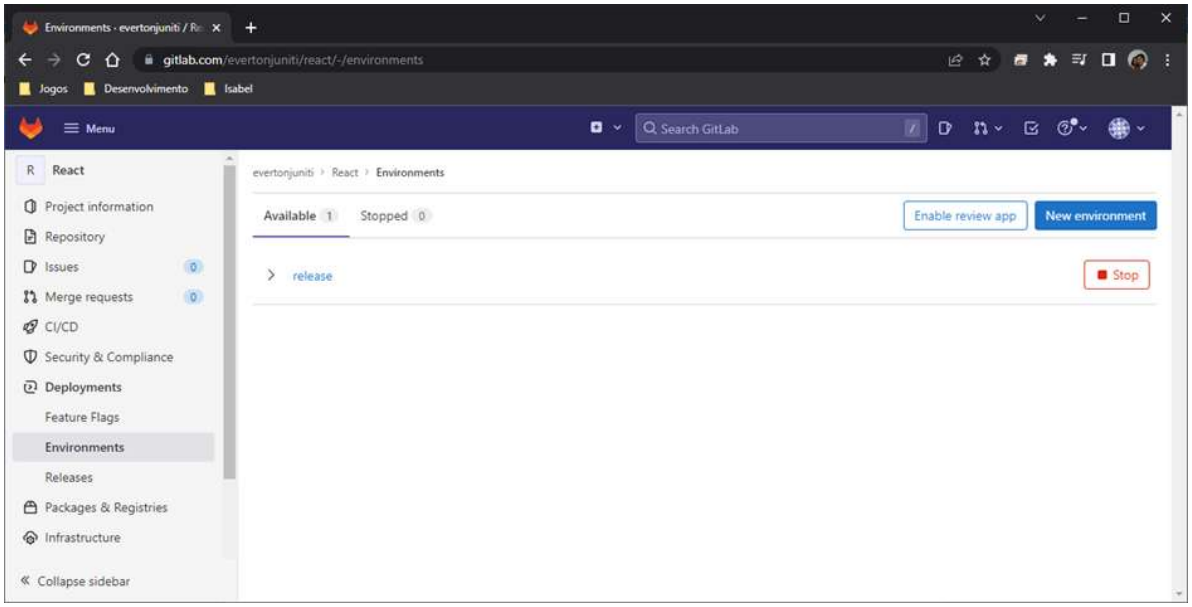
No repositório https://gitlab.com/everton.juniti/descomplica, na pasta cicd_frontend -> 02-Environment tem o script com uma pequena adição, ao final:

```
Criar_Container:
    stage: deploy
    only:
      - /^release_[0-9]+(?:.[0-9]+)+$/
    variables:
        GIT_STRATEGY: none
    script:
        - docker run --name $NOME_DO_CONTAINER --network $NOME_DA_REDE -p
$BIND_DE_PORTA -d $DOCKER_TAG_NAME
    environment:
        name: release
```

Veja que incluímos esse trecho aqui:

```
    environment:
        name: release
```

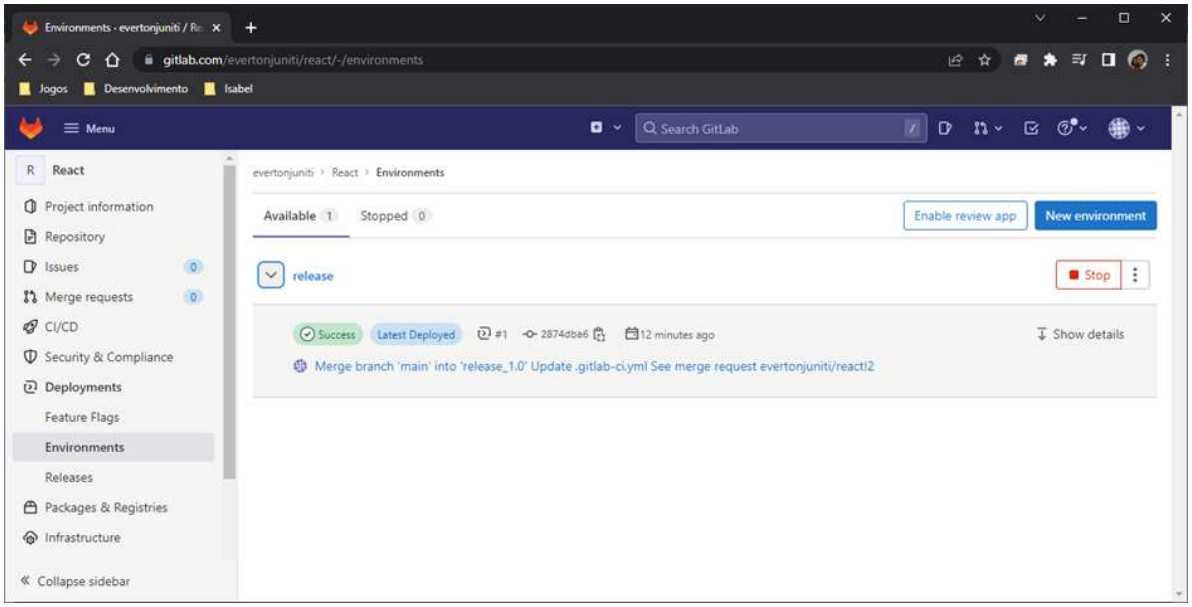
Ao subir esse código e fazer o merge request para a branch releas_1.0, além do pipeline iniciar você pode acompanhar também o deploy (a implantação) no Gitlab, menu à esquerda “Deployments” e depois clique no item “Environments”.



O “Environment” é apenas uma forma de organizar as execuções das pipelines, você terá uma visão das implantações que você fez (no nosso caso, indicamos que o ambiente é o de release e que somente o job Criar_Container da stage de

"deploy" é que aparecerá aqui), é uma maneira de você acompanhar o que aconteceu em cada ambiente que a pipeline entregou.

Veja como fica esta informação assim que o deploy é finalizado:



Então além de podermos acompanhar as execuções na tela das Pipelines, uma outra opção é esta em que podemos acompanhar as implantações por ambiente.

Agora vamos falar especificamente da Release branch. Nós estamos utilizando esta branch até agora mas não adentramos no significado que ela tem.

A Release (que significa Liberar) é comumente usada para indicar que o desenvolvedor terminou parte ou totalmente aquilo que estava desenvolvendo e está "liberando" o produto de seu trabalho para ser implantado em algum ambiente. A isto damos o nome de Release.

Mas não necessariamente a branch Release deverá implicar na implantação em algum ambiente, ainda mais quando estamos com vários desenvolvedores mexendo em suas Features e submetendo o merge request para a Release!

Imagine várias pessoas fazendo isso em paralelo, quantas vezes a pipeline não seria iniciada!

Então vamos fazer mais uma modificação em nosso script, vamos no repositório https://gitlab.com/everton.juniti/descomplica, na pasta cicd_frontend -> 03-Develop e observar as mudanças no script:

```
variables:
    APP_PATH: reactsite
    DOCKER_TAG_NAME: $APP_PATH:latest
    NOME_DO_CONTAINER: MeuReactSite
    NOME_DA_REDE: MinhaRede
    BIND_DE_PORTA: 3000:3000

stages:
    - prebuild
    - build
    - deploy

Limpar_Imagem:
    stage: prebuild
    only:
      - develop
    variables:
        GIT_STRATEGY: none
    script:
        - docker container stop $(docker container ls -a -f
ancestor=$DOCKER_TAG_NAME -q) || FAILED=true
        - echo Tentativa de parada de container. Falhou? - $FAILED
        - FAILED=false
        - docker container rm $(docker container ls -a -f
ancestor=$DOCKER_TAG_NAME -q) || FAILED=true
        - echo Tentativa de remoção de container. Falhou? - $FAILED
        - FAILED=false
        - docker image rm $DOCKER_TAG_NAME || FAILED=true
        - echo Tentativa de remoção da image. Falhou? - $FAILED

Criar_Imagem:
    stage: build
    only:
      - develop
    script:
        - docker build --no-cache -t $DOCKER_TAG_NAME ./$APP_PATH

Criar_Container:
    stage: deploy
    only:
      - develop
    variables:
        GIT_STRATEGY: none
    script:
        - docker run --name $NOME_DO_CONTAINER --network $NOME_DA_REDE -p
$BIND_DE_PORTA -d $DOCKER_TAG_NAME
    environment:
        name: develop
```

Veja que todos os locais onde havia o atributo “only” mudamos para que os jobs só iniciem caso a branch se chame “develop”:

```
  only:
    - develop
```

Também mudamos o environment para “develop”:

```
  environment:
      name: develop
```

Suba o script alterado e depois faça o merge request da “main” para a “release_1.0”. Veja que a pipeline não iniciou.

Crie uma branch nova chamada “develop” à partir da branch “release_1.0”.

Veja que agora sim o pipeline iniciou, pois devido ao atributo “only” no script, somente alterações na branch “develop” iniciam os jobs.

Note também lá em “Deployments” -> “Environments” que surgiu um novo ambiente chamado “develop”, pois isso também alteramos no job “Criar_Container”.

Ok, mas e se eu propositadamente quiser fazer a implantação a cada Release com a alteração que eu fiz só pra ver como ficou?

É possível com o que chamamos de “Environment Dinâmico”, iremos alterar novamente nosso script para fazer um build diferente, para a Release em questão e assim termos um container partindo da nossa Release que é diferente do container da “develop” que é do nosso ambiente de desenvolvimento.

Observação: aqui como estamos praticando tudo com o Docker da sua máquina, tanto a versão de “desenvolvimento” quanto a versão em “release” ficarão no mesmo Docker (na sua máquina), mas num ambiente real essas implantações ficariam em locais diferentes!

Então vamos fazer mais uma última modificação em nosso script, vamos no repositório https://gitlab.com/everton.juniti/descomplica, na pasta cicd_frontend -> 04-Environment_Dinamico e observar as mudanças no script:

```
variables:
    APP_PATH: reactsite
    DOCKER_TAG_NAME: $APP_PATH:latest
    DOCKER_RELEASE_TAG_NAME: $APP_PATH:$CI_COMMIT_REF_NAME
    NOME_DO_CONTAINER: MeuReactSite
    NOME_DO_CONTAINER_RELEASE: MeuReactSite_$CI_COMMIT_REF_NAME
    NOME_DA_REDE: MinhaRede
    BIND_DE_PORTA: 3000:3000
    BIND_DE_PORTA_RELEASE: 3001:3000

stages:
    - prebuild
    - build
    - deploy

Limpar_Imagem_Release:
    stage: prebuild
    only:
      - /^release_[0-9]+(?:.[0-9]+)+$/
    variables:
        GIT_STRATEGY: none
    script:
        - docker container stop $(docker container ls -a -f
ancestor=$DOCKER_RELEASE_TAG_NAME -q) || FAILED=true
        - echo Tentativa de parada de container. Falhou? - $FAILED
        - FAILED=false
        - docker container rm $(docker container ls -a -f
ancestor=$DOCKER_RELEASE_TAG_NAME -q) || FAILED=true
        - echo Tentativa de remoção de container. Falhou? - $FAILED
        - FAILED=false
        - docker image rm $DOCKER_RELEASE_TAG_NAME || FAILED=true
        - echo Tentativa de remoção da image. Falhou? - $FAILED
    environment:
        name: release
        action: stop
    when: manual

Limpar_Imagem:
    stage: prebuild
    only:
      - develop
    variables:
        GIT_STRATEGY: none
    script:
        - docker container stop $(docker container ls -a -f
ancestor=$DOCKER_TAG_NAME -q) || FAILED=true
        - echo Tentativa de parada de container. Falhou? - $FAILED
        - FAILED=false
```

```
        - docker container rm $(docker container ls -a -f
ancestor=$DOCKER_TAG_NAME -q) || FAILED=true
        - echo Tentativa de remoção de container. Falhou? - $FAILED
        - FAILED=false
        - docker image rm $DOCKER_TAG_NAME || FAILED=true
        - echo Tentativa de remoção da image. Falhou? - $FAILED

Criar_Imagem_Release:
    stage: build
    only:
      - /^release_[0-9]+(?:.[0-9]+)+$/
    script:
        - docker build --no-cache -t $DOCKER_RELEASE_TAG_NAME ./$APP_PATH

Criar_Imagem:
    stage: build
    only:
      - develop
    script:
        - docker build --no-cache -t $DOCKER_TAG_NAME ./$APP_PATH

Criar_Container_Release:
    stage: deploy
    only:
      - /^release_[0-9]+(?:.[0-9]+)+$/
    variables:
        GIT_STRATEGY: none
    script:
        - docker run --name $NOME_DO_CONTAINER_RELEASE --network $NOME_DA_REDE
-p $BIND_DE_PORTA_RELEASE -d $DOCKER_RELEASE_TAG_NAME
    environment:
        name: release
        on_stop: Limpar_Imagem_Release

Criar_Container:
    stage: deploy
    only:
      - develop
    variables:
        GIT_STRATEGY: none
    script:
        - docker run --name $NOME_DO_CONTAINER --network $NOME_DA_REDE -p
$BIND_DE_PORTA -d $DOCKER_TAG_NAME
    environment:
        name: develop
```

Note que incluímos alguns jobs, todos específicos para branch que tenha o nome de “Release”:

```
    only:
      - /^release_[0-9]+(?:.[0-9]+)+$/
```

Tem apenas uma coisa nova e muito importante que estamos adicionando especificamente na parte de deploy da release, olhe o "environment":

```
  environment:
      name: release
      on_stop: Limpar_Imagem_Release
```

Esse atributo "on_stop" faz o seguinte: na pipeline teremos a opção de clicar num botão de parada no job Criar_Container_Release e assim que clicarmos nesse botão, será feito o que está no job Limpar_Imagem_Release, que basicamente para o container dessa nossa release se estiver rodando, depois destrói o container e por último destrói essa imagem de release. Por isso que é dinâmico, você cria algo que é para ser temporário mesmo, com a vantagem de que estamos reutilizando um job que já faz parte do processo normal da pipeline para fazer essa limpeza! Assim não fica nenhum "lixo" no meio dos containers!

Tivemos que incluir o "environment" também nesse job de Limpar_Imagem_Release, dá uma olhada no job e veja que ao final incluímos:

```
  environment:
      name: release
      action: stop
  when: manual
```

Incluímos o atributo "action" com o valor "stop" justamente para que o Gitlab CI/CD saiba que deverá executar este job em uma parada, também foi incluído o atributo "when" com o valor "manual", isso é necessário para que o Gitlab CI/CD saiba que esta ação de parada será uma ação manual no console do Gitlab.

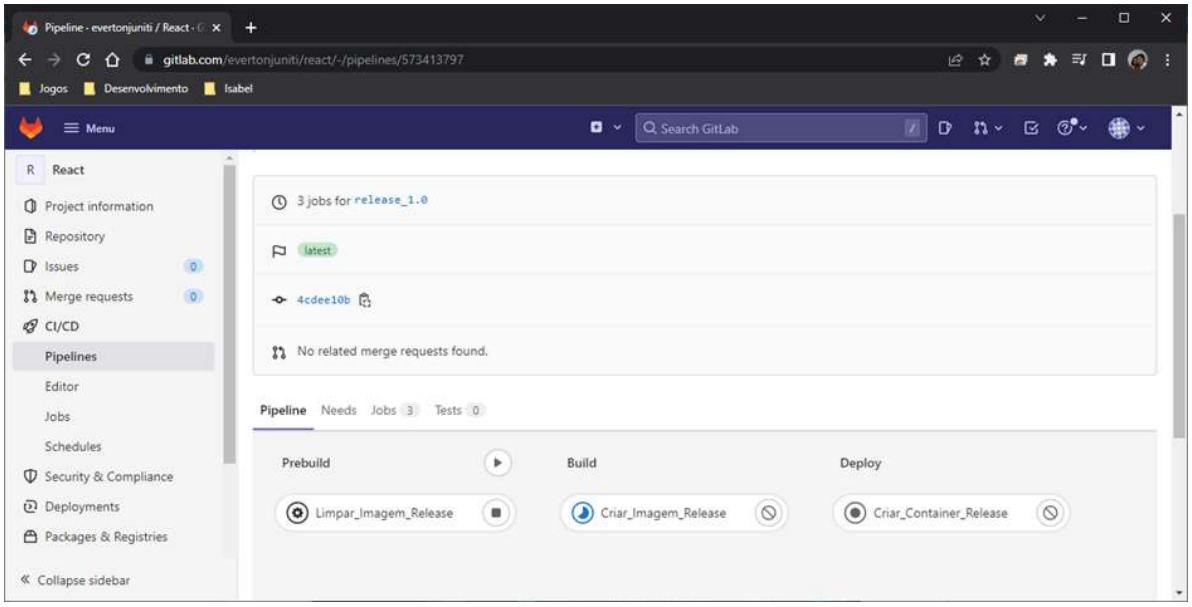
Note que também criamos algumas variáveis novas, utilizadas nestes jobs novos:

```
  DOCKER_RELEASE_TAG_NAME: $APP_PATH:$CI_COMMIT_REF_NAME
  NOME_DO_CONTAINER_RELEASE: MeuReactSite_$CI_COMMIT_REF_NAME
  BIND_DE_PORTA_RELEASE: 3001:3000
```

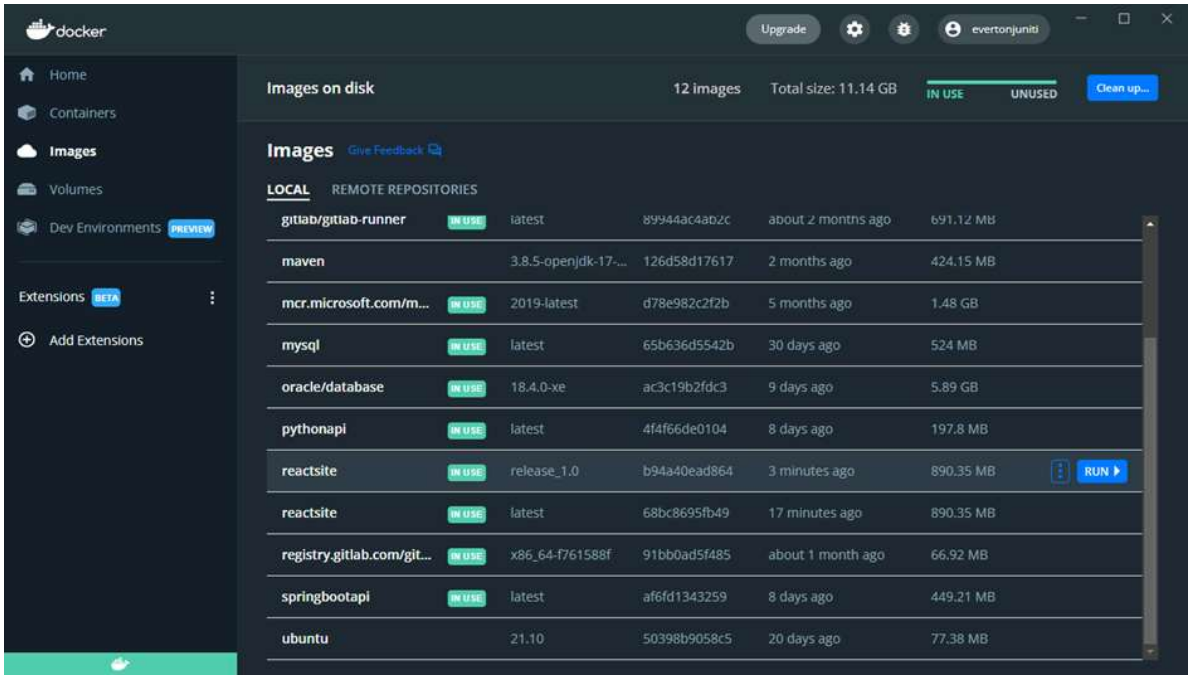
Nós estamos aqui criando um container paralelo para nosso site, só que o nome da imagem tem que ser diferente da reactsite:latest, o nome do container diferente do MeuReactSite e o número da porta no bind diferente de 3000, tudo isso pois já teremos um container rodando com essas informações que é o nosso ambiente de desenvolvimento.

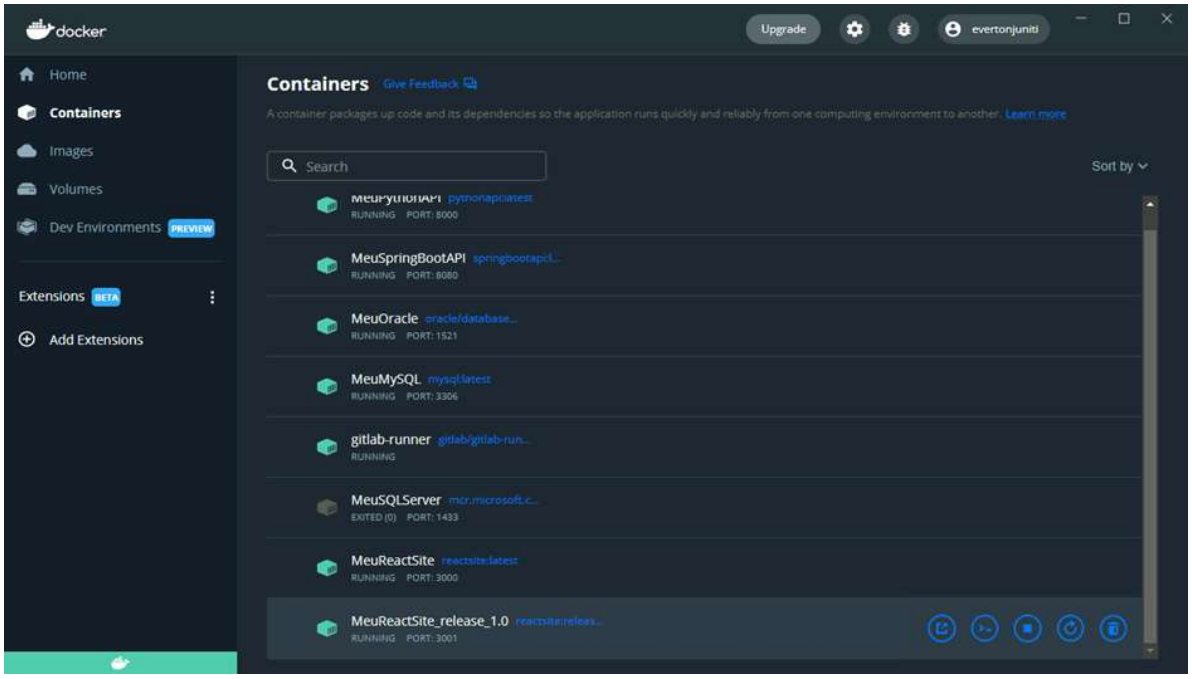
Aqui ao fazermos a subida do script e merge request para a "Release_1.0", iremos criar uma imagem chamada reactsite:Release_1.0, um container chamado MeuReactSite_Release_1.0 com um bind de porta para 3001, assim conseguiremos rodar tanto o site em desenvolvimento quanto o site mexido como se fosse um outro ambiente de Release! O $CU_COMMIT_REF_NAME é uma variável interna do Gitlab CI/CD que representa o nome da branch (no nosso exemplo: release_1.0).

Uma coisa interessante é olhar como ficou a pipeline, já que agora temos o botão de "Stop" ativo na pipeline para fazer a limpeza dessa versão de Release.

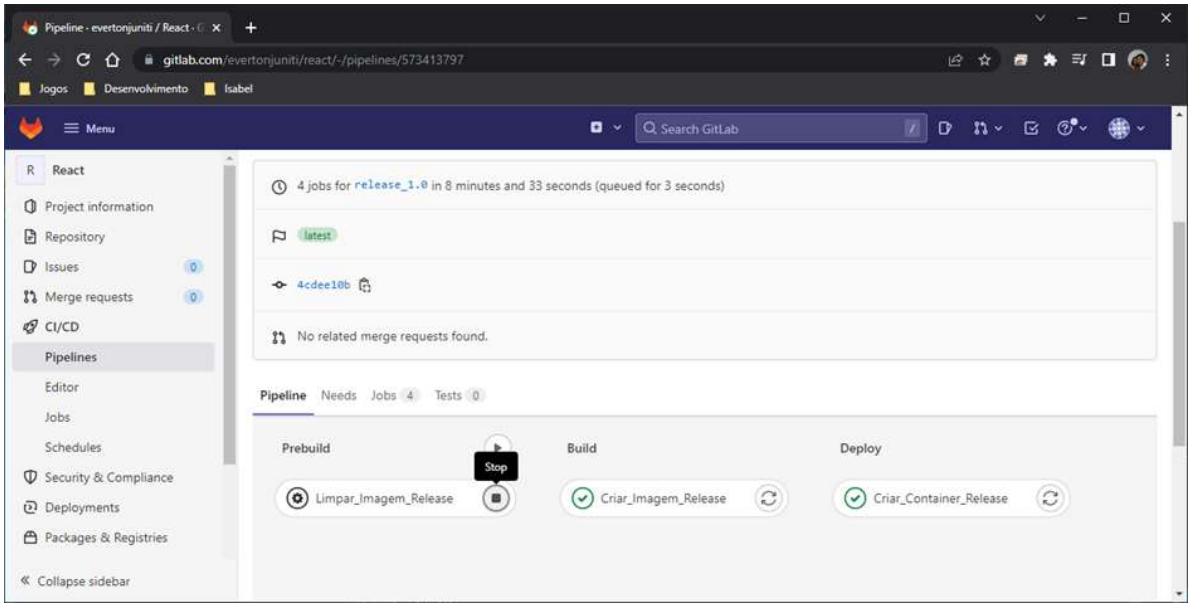


Depois que a pipeline rodar, veja como ficará no seu Docker Desktop.





Agora aperte aquele botão de “Stop” no job Limpar_Imagem_Release. No script indicamos que ele irá parar o container de release, excluir esse container e depois excluir a imagem. Esse é nosso ambiente dinâmico, não é necessário que o container nem a imagem permaneçam, já que o que deverá valer é a versão de desenvolvimento, mas isso nos permite usar as 2 versões (release e desenvolvimento) para testes distintos que podem ser feitos pelo desenvolvedor.



Geralmente a branch de produção é protegida para que ninguém consiga fazer um push direto para ela, sendo necessário um merge request (solicitação para mesclagem) para que o código saia de uma outra branch (geralmente de homologação) para ir para a produção.

Os times de desenvolvimento geralmente criam outras branches (geralmente chamadas de Feature) à partir da branch develop para desenvolverem em paralelo e depois fazem o merge (mesclagem) do que foi produzido nas várias features na branch develop ou entregam individualmente para fazer os testes de desenvolvimento.

Vamos brincar de “Git Flow”!

Faça agora algumas brincadeiras como criar outras branches à partir da “release_1.0” chamada de “feature_A” e “feature_B” e mexa nestas branches, depois faça o merge request da “feature_A” para a “release_1.0” e depois outro merge request da “feature_B” para a “release_1.0”.

Aqui fizemos o ciclo de um time de desenvolvimento, onde temos nossa Release realmente preparada para ir para o ambiente de desenvolvimento, vamos fazer o seguinte: faça um merge request da “Release” para a “Develop” e observe o pipeline.

Atividade Extra

Para se aprofundar no assunto desta aula leia o documentário de referência oficial: "Environments and deployments".

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

Referência Bibliográfica

OGURA, Everton J. **Repositório do GitLab, projeto Descomplica**. Disponível em https://gitlab.com/everton.juniti/descomplica. Acesso em 26 de junho de 2022.

.gitlab-ci.yml **keyword reference**. Disponível em https://docs.gitlab.com/ee/ci/yaml/. Acesso em 26 de junho de 2022.

Predefined variables reference. Disponível em https://docs.gitlab.com/ee/ci/variables/predefined_variables.html. Acesso em 26 de junho de 2022.

Ir para exercício