Estágios de um fluxo de pipeline

Muito bem, agora que já entendemos o que são as **stages** e as **jobs** de uma pipeline, vamos montar uma pipeline mais próxima do real com **stages** de **test**, **build** e **deploy**.

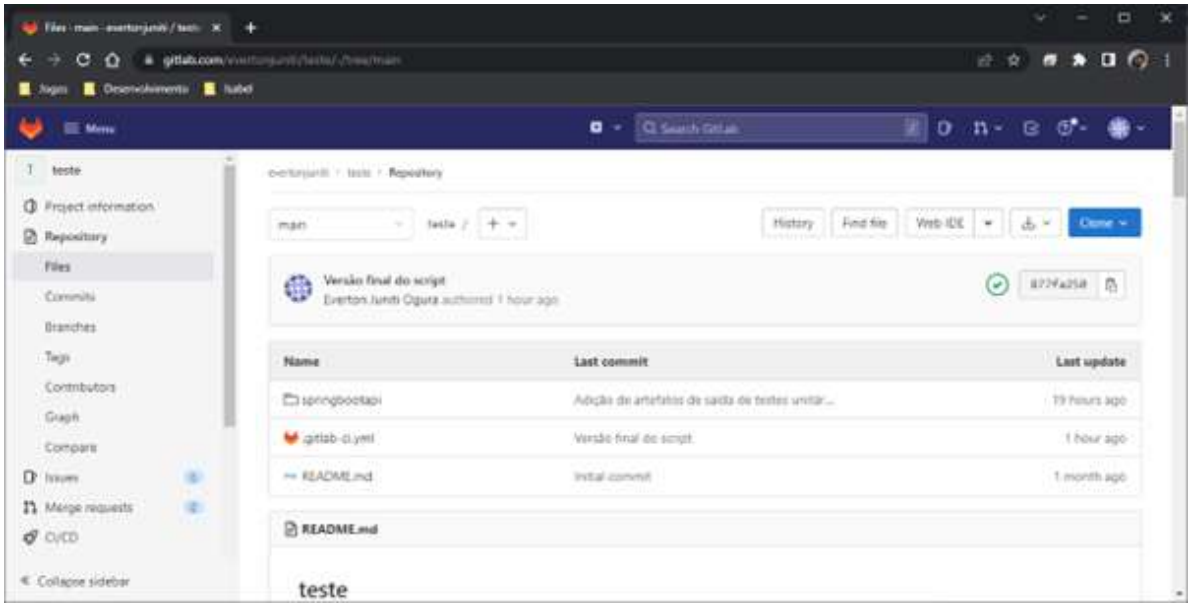
Iremos simular (na prática claro) essas etapas partindo de um código-fonte que já temos em mãos e construiremos cada etapa através do nosso script em YAML. Então a primeira coisa que faremos é acessar o repositório com o código-fonte pronto em https://gitlab.com/everton.juniti/descomplica, faça um clone do repositório (no próprio repositório tem as instruções de como fazer isso).

Faça um clone do repositório de testes que criamos também, usaremos esse repositório de testes para "brincar" de pipeline.

Após fazer o clone do repositório https://gitlab.com/everton.juniti/descomplica, vá dentro da pasta cicd_backend e copie a pasta springbootapi, cole esta cópia da pasta no diretório em que você fez o clone do seu repositório de teste.

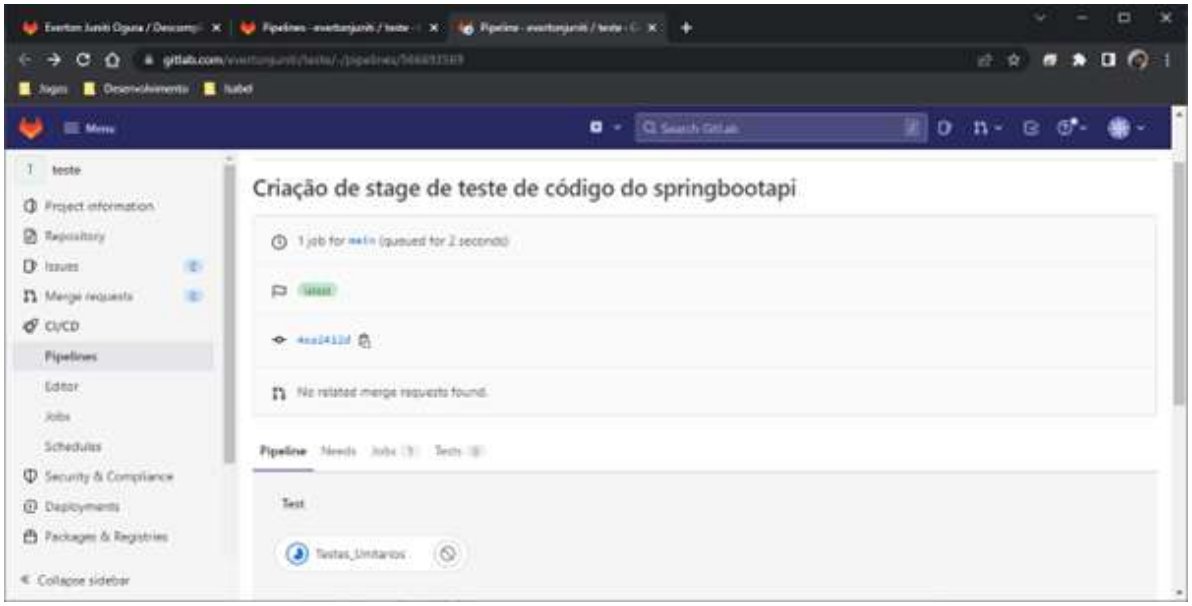
O que fizemos aqui foi pegar um código-fonte pronto e recheamos nosso repositório de testes com esse mesmo código, cópia mesmo! Agora criaremos em etapas nossa pipeline completa que irá pegar esse código-fonte do repositório, efetuaremos testes prévios antes de fazer o build, faremos o build da aplicação para gerarmos uma imagem do Docker através do Dockerfile que já temos dentro da pasta springbootapi e por fim faremos o deploy que é a subida de um container à partir da imagem recém gerada!

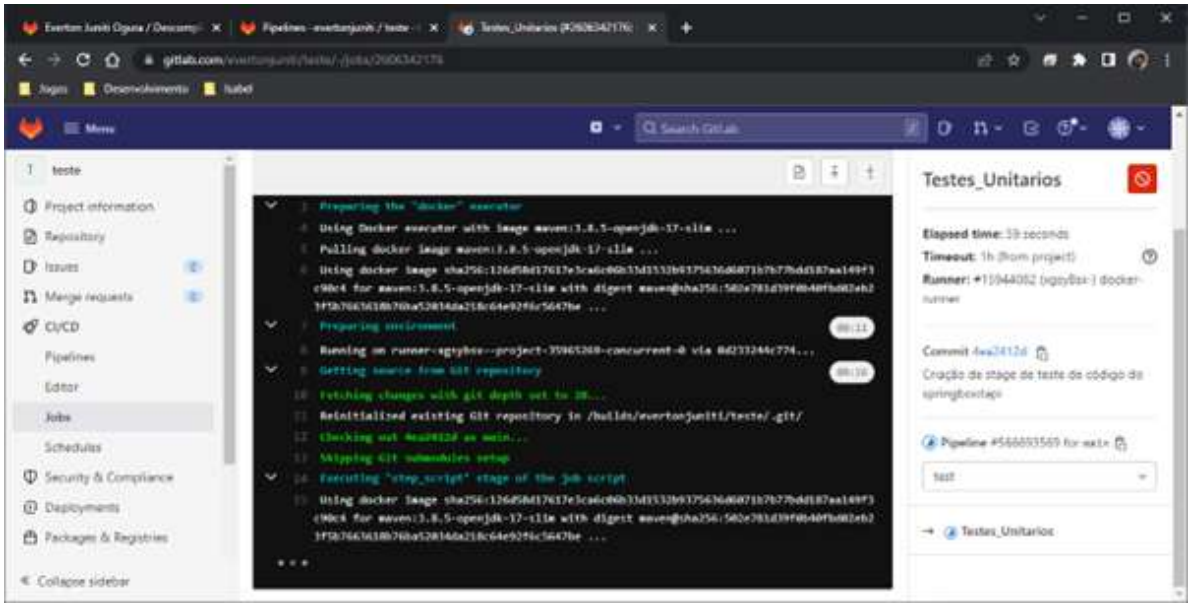
Se você fez tudo certo até aqui, seu repositório de teste deve ter ficado assim:



Agora vamos "brincar" com o nosso script de pipeline, você pode editar o .gitlab-ci.yml no bloco de notas ou via Visual Studio Code e ir fazendo o "push" das alterações.

Vamos começar criando nosso stage de teste que irá executar um comando para testar a aplicação springbootapi, para isto pegue o conteúdo seguinte no repositório https://gitlab.com/everton.juniti/descomplica, pasta cicd_generico\03-Test.gitlab-ci.yml, cole o conteúdo no .gitlab-ci.yml em seu repositório de teste (ou substitua o arquivo inteiro) e dê um "push" no git, veja que a pipeline começará a executar:





Vamos entender o que está sendo executado no script, para isto vejamos o código que alteramos:

```
variables:
    APP_PATH: springbootapi

stages:
    - test

Testes_Unitarios:
    stage: test
    image: maven:3.8.5-openjdk-17-slim
    script:
        - cd $APP_PATH
        - mvn test
```

Por enquanto é um script bastante simples em que incrementaremos com o tempo, mas observemos os trechos:

```
variables:
    APP_PATH: springbootapi
```

Aqui eu criei uma variável chamada APP_PATH com o valor springbootapi, esta variável irá nos ajudar pois como este valor será utilizado várias vezes, podemos usar uma variável que facilite nosso trabalho caso quisermos alterar o valor springbootapi para qualquer outra coisa.

```
stages:
    - test
```

Aqui indicamos que haverá apenas um estágio na pipeline, identificada como test

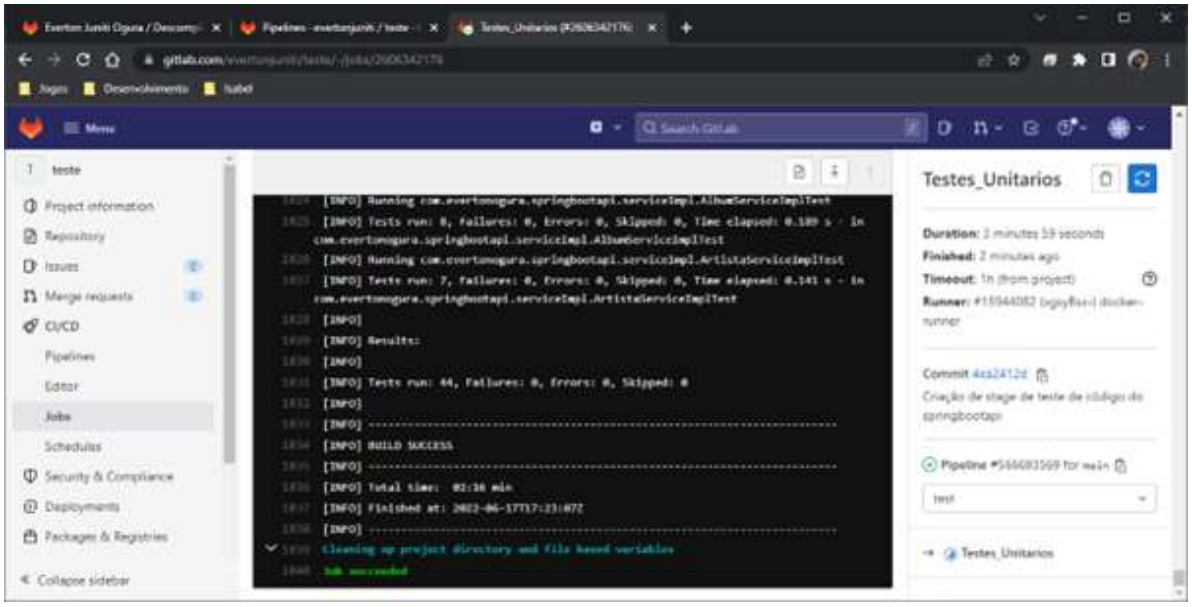
```
Testes_Unitarios:
    stage: test
    image: maven:3.8.5-openjdk-17-slim
    script:
        - cd $APP_PATH
        - mvn test
```

Aqui temos um job chamado "Testes_Unitarios", este job pertence ao **stage** chamado test (que vimos anteriormente), aqui especificamente indicamos a imagem base para processar os testes unitários da nossa solução em Java, neste caso a imagem é a "maven", versão "3.8.5-openjdk-17-slim", através dessa imagem nós conseguiremos executar os comandos que estão em **script**, sendo 2 comandos:

cd $APP_PATH: para entrar na pasta springbootapi, uma vez que este valor está na variável APP_PATH

mvn test: é o comando do Maven para executar os testes unitários presentes na solução do Java, só é possível executar este comando pois indicamos a imagem do maven como base para nosso script.

O final da execução deste job é o resultado da execução do comando “mvn test”, conforme imagem:



A pipeline fica com a indicação de execução com sucesso! Agora vamos para a próxima parte, o build.

Faremos o mesmo procedimento de alteração do .gitlab-ci.yml do nosso repositório de teste, só que desta vez teremos que alterar também o arquivo Dockerfile que está dentro da pasta springbootapi. Vamos utilizar o conteúdo já pronto do repositório https://gitlab.com/everton.juniti/descomplica, pasta cicd_generico\04-Build para atualizar tanto o arquivo .gitlab-ci.yml quanto o Dockerfile.

A diferença no arquivo Dockerfile é a inclusão de uma instrução à mais na parte de build:

```
-Dmaven.test.skip=true
```

Como já estamos realizando testes unitários na stage de test, não faz sentido executarmos os testes unitários novamente ao rodar nossa nova stage de build, e como nossa stage de build utiliza o Dockerfile, faz-se necessário alterar também o Dockerfile.

Nosso novo script ficará assim:

```
variables:
    APP_PATH: springbootapi
    DOCKER_TAG_NAME: $APP_PATH:latest

stages:
    - test
    - prebuild
    - build

Testes_Unitarios:
    stage: test
    image: maven:3.8.5-openjdk-17-slim
    script:
        - cd $APP_PATH
        - mvn test

Limpar_Imagem:
    stage: prebuild
    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
    script:
        - docker build --no-cache -t $DOCKER_TAG_NAME ./$APP_PATH
```

Sendo:

```
    DOCKER_TAG_NAME: $APP_PATH:latest
```

Incluímos uma variável nova que será a tag da nossa imagem, e esta tag será utilizada em vários locais. Note que estamos concatenando o valor de uma variável

que já existia (a APP_PATH) com o conteúdo novo desta variável, no final do dia o valor desta variável nova será: springbootapi:latest

```
    - prebuild
    - build
```

Estamos incluindo 2 estágios novos: prebuild e build, já explicaremos porque incluímos essa prebuild.

```
Limpar_Imagem:
    stage: prebuild
    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
```

Até agora nós brincamos bastante com o Docker e é provável que você ainda tenha a imagem do springbootapi presente. O que precisamos fazer é garantir que o container seja parado para que consigamos excluí-lo e, só depois de não ter nenhum container usando a imagem springbootapi, poderemos apaga-lo também.

Esse passo é necessário antes de criarmos nossa nova imagem! Então adicionamos esse passo de "prebuild". Os comando são simples, eles tentam parar e remover o container e depois remover a imagem, o "|| FAILED=true" ao final de todos os comando é proposital, pode ser que o container já não exista mais então daria um erro ao tentar parar e remover o container, então para pular erros eventuais (que para o nosso fluxo não é um erro no final do dia), incluímos esse || que indica que

caso falhe algo à mais irá acontecer, estamos preenchendo uma variável com valor true apenas para passar por este comando sem "estourar" um erro no job.

Há uma parte interessante neste estágio de prebuild:

```
    variables:
        GIT_STRATEGY: none
```

Como estamos apenas executando comando do Docker, não é necessário que tenhamos o código-fonte do repositório, esta declaração neste job faz o seguinte: ela diz ao runner que neste job não é necessário fazer o clone do repositório do git! Isso é muito útil pois cada job por padrão faz o clone do repositório inteiro e depois trabalha nele (lembre-se que o runner vai criando containers temporários para executar cada passo), então aqui para "poupar" passinhos indicamos que não é necessário esse clone neste momento.

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

Após a limpeza finalmente temos nosso job de build, o script apenas executa um comando para o Docker efetuar o build, criando uma imagem nova com uma tag chamada springbootapi:latest (contido na variável DOCKER_TAG_NAME). Como o Dockerfile está dentro da pasta springbootapi, indicamos neste comando que é para tentar efetuar o docker build em ./$APP_PATH (que será transformado em ./springbootapi, já que o valor springbootapi está contido na variável APP_PATH).

Quando você efetuar o push, acompanhe a pipeline e veja que uma nova imagem é criada no seu Docker local.

Um recurso interessante do Gitlab CI/CD é poder utilizar o resultado final da execução de um job e conseguir passar para os próximos jobs os arquivos gerados

por um job anterior, ou permitir fazer o download desses resultados.

Apresento o "artifacts", que nos permite trabalhar com esses resultados de um job. Aqui neste exemplo eu vou incluir na **stage** de test a criação de um artefato que é o resultado da execução dos testes unitários, para isso teremos que fazer alguns passos.

Vamos olhar no repositório já pronto https://gitlab.com/everton.juniti/descomplica, a pasta cicd_generico\05-Artifact. Veja que tem o arquivo pom.xml, você deverá substituir o arquivo pom.xml que está dentro da pasta springbootapi no seu repositório de testes. Há a inclusão do seguinte código no pom.xml:

```
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <configuration>
                    <redirectTestOutputToFile>true</redirectTestOutputToFile>
                </configuration>
            </plugin>
```

Esse plugin do Maven faz com que consigamos exportar o resultado de todos os testes unitários para arquivos e são estes arquivos que guardaremos como artefato para download.

Já o script .gitlab-ci.yml ficará assim:

```
variables:
    APP_PATH: springbootapi
    DOCKER_TAG_NAME: $APP_PATH:latest

stages:
    - test
    - prebuild
    - build

Testes_Unitarios:
    stage: test
    artifacts:
        paths:
            - $APP_PATH/target/surefire-reports/
    image: maven:3.8.5-openjdk-17-slim
    script:
        - cd $APP_PATH
        - mvn test

Limpar_Imagem:
    stage: prebuild
    dependencies: []
    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
    dependencies: []
    script:
        - docker build --no-cache -t $DOCKER_TAG_NAME ./$APP_PATH
```

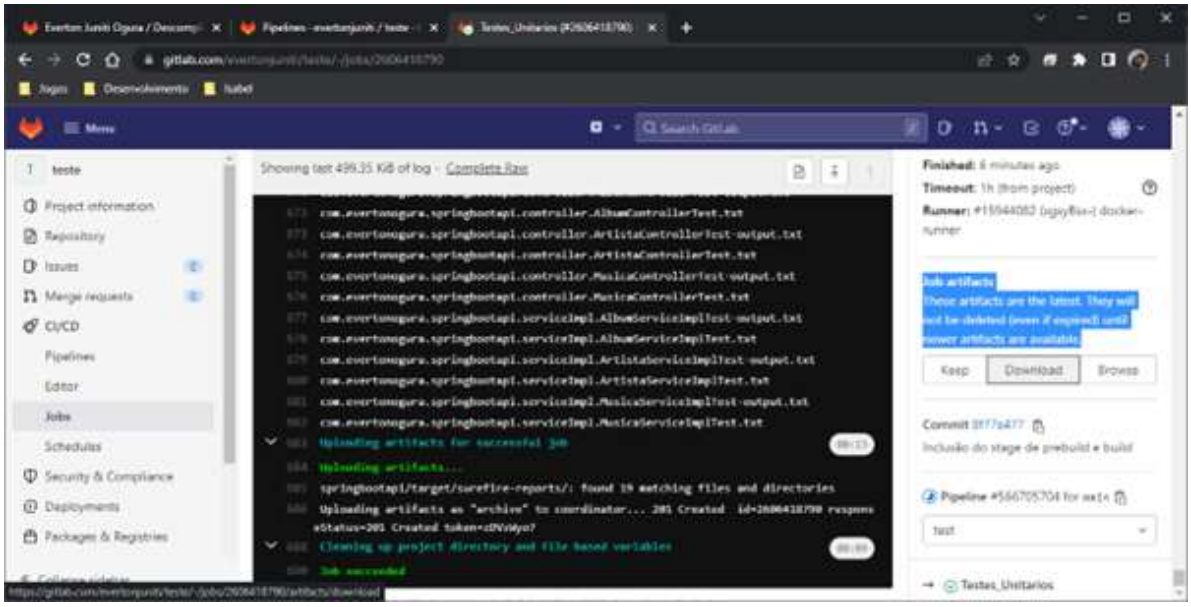
Note as seguintes inclusões:

```
    artifacts:
        paths:
            - $APP_PATH/target/surefire-reports/
```

Aqui indicamos que ao final do job Testes_Unitarios, os arquivos da pasta springbootapi/target/surefire-reports ficarão disponíveis como artefato, onde poderá ser utilizado em outros jobs e você poderá fazer o download no Gitlab.

```
    dependencies: []
```

Se eu não quero utilizar os artefatos gerados por um job anterior no meu job atual (caso dos jobs Limpar_Imagem e Criar_Imagem), eu declaro explicitamente no job essa linha, que indica que não serão copiados os artefatos (pois não usaremos).



Além dos artefatos, podemos usar o cache. A diferença é que o cache não disponibiliza arquivos para download, pois em alguns casos não queremos isso! Vejamos o uso de cache para transportar o código-fonte para os demais jobs. Faremos isso pois há 2 jobs que usam o código-fonte da pasta springbootapi: Testes_Unitarios e Criar_Imagem, só que ao invés de deixar o runner fazer o git clone 2 vezes nesses jobs, usaremos o cache para compartilhar o conteúdo da pasta springbootapi do clone feito no job Testes_Unitarios (que é o primeiro job) para ser reutilizado na job Criar_Imagem assim não precisaremos fazer o clone novamente.

Veja no repositório pronto https://gitlab.com/everton.juniti/descomplica, a pasta cicd_generico\06-Cache:

```
variables:
    APP_PATH: springbootapi
    DOCKER_TAG_NAME: $APP_PATH:latest

stages:
    - test
    - prebuild
    - build

Testes_Unitarios:
    stage: test
    artifacts:
        paths:
            - $APP_PATH/target/surefire-reports/
    cache:
        paths:
            - $APP_PATH
    image: maven:3.8.5-openjdk-17-slim
    script:
        - cd $APP_PATH
        - mvn test

Limpar_Imagem:
    stage: prebuild
    dependencies: []
    cache: []
    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
    dependencies: []
    cache:
        paths:
            - $APP_PATH
    variables:
        GIT_STRATEGY: none
    script:
```

Sendo:

```
    cache:
        paths:
            - $APP_PATH
```

Incluímos essa instrução no job Testes_Unitarios para armazenar em cache o conteúdo inteiro da pasta springbootapi (valor contido na variável APP_PATH).

```
    cache: []
```

No job Limpar_Imagem não precisaremos do código-fonte, pois só teremos os comandos do Docker, então indicamos desta forma para não trazer o conteúdo do cache.

```
    cache:
        paths:
            - $APP_PATH
    variables:
        GIT_STRATEGY: none
```

Repetimos a instrução no job Criar_Imagem para indicar que queremos fazer o uso do conteúdo do cache e indicamos o "GIT_STRATEGY: none" justamente para não precisarmos fazer o git clone nesta etapa.

Agora só falta a parte final que é o deploy! Olhemos o repositório pronto https://gitlab.com/everton.juniti/descomplica, na pasta cicd_generico\07-Deploy, olhe como ficará nosso .gitlab-ci.yml:

```yaml
variables:
    APP_PATH: springbootapi
    DOCKER_TAG_NAME: $APP_PATH:latest

stages:
    - test
    - prebuild
    - build
    - deploy

Testes_Unitarios:
    stage: test
    artifacts:
        paths:
            - $APP_PATH/target/surefire-reports/
    cache:
        paths:
            - $APP_PATH
    image: maven:3.8.5-openjdk-17-slim
    script:
        - cd $APP_PATH
        - mvn test

Limpar_Imagem:
    stage: prebuild
    dependencies: []
    cache: []
    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
    dependencies: []
    cache:
        paths:
            - $APP_PATH
    variables:
        GIT_STRATEGY: none
```

Sendo:

```
    - deploy
```

Agora sim estamos adicionando a **stage** de deploy!

```
Criar_Container:
    stage: deploy
    dependencies: []
    cache: []
    variables:
        GIT_STRATEGY: none
    script:
        - docker run --name MeuSpringBootAPI --network MinhaRede -p 8080:8080
-d $DOCKER_TAG_NAME
```

E finalmente nosso job de deploy. Após criar a imagem, podemos criar um container no Docker, indicamos não trazer nenhum artefato através da instrução "dependencies: []", e também não trazer nada do cache já que não iremos interagir com o código-fonte através da instrução "cache: []", também não precisamos fazer o git clone. O script contém o mesmo comando que usamos para subir o container do springbootapi, o que difere é o uso da variável DOCKER_TAG_NAME para indicar a tag da imagem que iremos usar no container.

ATENÇÃO: antes de dar o push neste último script que tem o deploy, deixe ligado o container que criamos anteriormente do banco de dados Oracle, assim quando o container subir ele não dará erro!

E pronto! Vimos como subir de forma simples um script completo, que testa o nosso código-fonte modelo (em Java), faz uma limpeza antes para excluir container e imagem existente, fizemos o build compilando o Java sem testar duas vezes para criar nossa imagem no repositório do Docker local e ao final fizemos o deploy subindo um container novo à partir da imagem recém criada!

Atividade Extra

Para se aprofundar no assunto desta aula leia o documento de referência: “Caching in GitLab CI/CD”. Neste material temos com maior detalhamento as diferenças entre o uso de Artefatos e Cache na pipeline.

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

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.

- GitLab CI/CD variables. Disponível em https://docs.gitlab.com/ee/ci/variables/. Acesso em 16 de junho de 2022.

- Git strategy. Disponível em https://docs.gitlab.com/ee/ci/runners/configure_runners.html#git-strategy. Acesso em 16 de junho de 2022.

- Job artifacts. Disponível em https://docs.gitlab.com/ee/ci/pipelines/job_artifacts.html. Aceso em 16 de junho de 2022.

- Caching in GitLab CI/CD. Disponível em https://docs.gitlab.com/ee/ci/caching/. Acesso em 16 de junho de 2022.

Ir para exercício