Docker

Comandos Docker:

**docker build [FLAGS] [DOCKERFILE\_PATH]:** “Builda” uma imagem baseada nas instruções do Dockerfile. O parâmetro “.” para o [DOCKERFILE\_PATH] significa que o Dockerfile encontra-se na mesma pasta do diretório onde o CMD foi aberto. Em caso de mudanças no código ou em outros arquivos, a imagem deve ser **rebuildada!**

Flags:

* –t nomeia a imagem a ser buildada

**docker run [FLAGS] [IMAGE\_NAME]:** Inicia o contâiner informado

Flags:

* –p [LOCAL\_PORT]:[DOCKER\_EXPOSED\_PORT] antes do nome da imagem informa a porta a ser utilizada localmente para acessar a aplicação
* --rm deleta o contêiner após ele ser “stoppado”
* –it roda o contêiner em modo interativo.
* --name dá um nome ao contêiner

**docker start [FLAGS] [CONTAINER\_NAME]:** Inicia um contêiner já existente, diferente do docker run, **que cria um novo**.

Flags:

* –it inicia o contêiner em modo interativo
* –a “attacha-se” ao contêiner

**docker stop [CONTAINER\_NAME]:** Para o contêiner informado

**docker ps:** Lista todos os contêineres e/ou processos rodando (ps = processes)

Flags:

* –a lista todos os processos, e não somente aqueles rodando

**docker attach [CONTAINER\_NAME]:** “Attacha-se” a um contêiner rodando, para que possa-se visualizar o output do contêiner

**docker logs [CONTAINER\_NAME]:** Printa os logs do contêiner informado

**docker inspect [CONTAINER\_NAME]:** Inspeciona as informações de um contêiner

**docker container prune:** Deleta todos os contêineres

**docker image prune:** Deleta todas as imagens que estão “untagged”

Flags:

* -a remove **TODAS** as imagens

**docker volume prune:** Deleta todos os volumes

Volumes

Volumes são pastas compartilhadas entre o contêiner e o computador local. São listados pelo comando docker volume ls

**Anonymous volumes:** Não são nomeados pelo desenvolvedor, e de tal forma, desaparecem assim que o contêiner é deletado. Quando utilizados juntamente aos “bind mounts”, eles podem “apontar” para uma pasta que não será sobreescrita por ele. Quando fazemos um “bind mount” da pasta-raiz do projeto no WORKDIR do contêiner com a pasta na máquina local, as pastas que não existem localmente somem do contêiner, **mesmo que rodemos um “npm install” no Dockerfile para gerar o node\_modules**. Neste cenário, a utilização dos “Anonymous volumes” se demonstra útil. Sua utilização é melhor debatida nas aulas 51 e 53 do curso.

* (No Dockerfile) VOLUME [ “[PATH\_INSIDE\_THE\_CONTAINER\_]” ]
* (Ao criar um contêiner) docker run –v [VOLUME\_PATH\_INSIDE\_CONTAINER]

**Named Volumes:** Utilizados quando é necessário que o volume não desapareça ao deletar o contêiner. Não estão fortemente atrelados a somente um contêiner.

* -v [VOLUME\_NAME]:[VOLUME\_PATH\_INSIDE\_CONTAINER] cria um “named volume”, usado para que os dados não se percam ao finalizar um contêiner

**Bind Mounts:** Utilizados quando precisamos que uma pasta dentro do contêiner possa ser modificada de fora dele, ou vice-versa. Por exemplo, quando não queremos re-buildar a imagem sempre que uma modificação for feita no código.

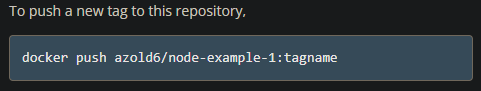
* -v [FOLDER\_PATH\_TO\_BE\_BOUND\_MYPC]:[ FOLDER\_PATH\_TO\_BE\_BOUND\_CONTEINER] cria um “bind mount”, usado para compartilhar arquivos e mudanças em uma pasta específica entre o PC e contêiner

Docker Hub

Ambiente do docker de onde puxamos e para onde enviamos imagens.

**Para enviar uma imagem ao Docker Hub:**

* Criar um repositório com o nome da imagem desejada
* Logar na CLI com o comando **docker login**
* Renomear a imagem desejada para o mesmo padrão do repositório criado, usando o seguinte comando: **docker tag [OLD\_IMAGE\_NAME] [NEW\_IMAGE\_NAME**], ou setando a tag no **docker build**
* Executar o comando **docker push [IMAGE\_NAME]:[TAG\_NAME]** contido no repositório, como segue o exemplo abaixo:



Comunicação Local-Contêiner

Deve-se adicionar a keyword ‘host.docker.internal’ no lugar de ‘localhost’, segue exemplo:

* mongodb://localhost:27017/swfavorites -> mongodb://host.docker.internal:27017/swfavorites

Comunicação Contêiner–Contêiner

É possível usar o comando docker inspect [CONTAINER\_NAME], encontrar o IP do contêiner em questão, e inserí-lo na URL de comunicação, como segue o exemplo, visando comunicar-se com um MongoDb hospedado em outro contêiner:

* mongodb://localhost:27017/swfavorites -> mongodb://[IP\_EXEMPLO]:27017/swfavorites

Entretanto, essa não é a melhor forma de fazê-lo.

Como alternativa, **podemos criar uma rede interna do docker**, que possibilita a comunicação entre contêineres que são incluídos nela.

Para criar a rede, usamos o comando:

* docker network create [NETWORK\_NAME]

Logo em seguida, podemos incluir contêineres nela. Como exemplo, usarei um contêiner contendo uma imagem do MongoDb, para que ele se comunique com outro contêiner de uma aplicação Nodejs:

* docker network create favorites
* docker run --network favorites –d --rm mongo
* docker run --network favorites –d --rm mynodejscontainer

Após isso, **a URL de conexão na aplicação Nodejs poderá usar, como host, o nome do contêiner que hospeda o banco**, como segue o exemplo:

* mongodb://[IP\_EXEMPLO]:27017/swfavorites -> mongodb://mongo:27017/swfavorites

Environment Variables

* (No Dockerfile) ENV [VAR\_NAME] [VAR\_VALUE] cria uma variável de ambiente

Com a utilização de uma **ENV**, podemos referenciá-la dentro do Dockerfile com um ‘$[VAR\_NAME]’, como segue o exemplo:

ENV PORT 80

EXPOSE $PORT

O valor setado no Dockerfile é o padrão, podendo ser alterado no comando docker run

docker run –p 3000:8000 –-env PORT 8000

Também é possível criar um arquivo ‘.env’, que contém as variáveis. Consultar curso, Aula 61, min. 7:10.

Docker Compose

Serve para subirmos diversos contêineres (services) utilizando menos comandos. Importante ressaltar que contêineres no mesmo docker-compose são colocados na mesma network, e por isso, podemos referenciar seus nomes em URLs de conexão.

**Estrutura de um docker-compose:**

version: "3.8"

services:

  mongodb:

    image: 'mongo'  #Usa uma imagem pronta

    volumes:

      - data:/data/db

    environment:

      MONGO\_INITDB\_ROOT\_USERNAME: max

      MONGO\_INITDB\_ROOT\_PASSWORD: secret

    container\_name: mongodb #Define o nome do contêiner

  backend:

    build: ./backend  #Builda uma imagem com do caminho de um Dockerfile (a partir do compose)

    environment:      #Variáveis de ambiente

      MONGODB\_USERNAME: max

      MONGODB\_PASSWORD: secret

    volumes:              #Volumes

      - logs:/app/logs    #Named volume

      - ./backend:/app    #Bind-Mount. O diretório '< :' pode ser especificado a partir do compose

      - /app/node\_modules #Anonymous volume

    ports:

      - '80:80'

    depends\_on:

      - mongodb #Esse serviço depende do mongodb para inicializar

  frontend:

    build: ./frontend

    volumes:

      - ./frontend/src:/app/src

    ports:

      - '3000:3000'

    depends\_on:

      - backend

    stdin\_open: true  #flag -it

    tty: true         #flag -it

volumes:

  data: #Named volumes devem obrigatóriamente ser especificados aqui. N vale pra bind mount ou A.V

  logs:

**Comandos do docker-compose:**

**docker-compose up [FLAGS]:** Inicia os contêineres contidos no compose

Flags:

* --build Força o ‘rebuild’ das imagens contidas no docker-compose.yml

Utility Containers

Servem para “buildarmos” o ambiente sem termos as ferramentas necessárias para tal instaladas.

**Passo a passo para gerarmos um projeto Nodejs e suas dependências sem lhe termos instalado:**

* Gerar um Dockerfile na seguinte estrutura

FROM node

WORKDIR /app

ENTRYPOINT [ "npm" ]

* Rodar **docker build –t nodeutil .** para gerar uma imagem do Dockerfile
* Rodar **docker run –v [LOCAL\_PATH]:[CONTAINER\_PATH] -it nodeutil init**
* Rodar **docker run –v [LOCAL\_PATH]:[CONTAINER\_PATH] -it nodeutil install**
* Por fim, teremos gerado o package.json e seu respectivo arquivo de dependências



**Importante:** Os comandos após o nome da imagem (destacados em **verde**) só são possíveis devido ao **ENTRYPOINT [ “npm” ]** no Dockerfile. Ele permite que executemos comandos a partir do **npm.**

**Alternativamente, podemos usar o docker-compose para utilizarmos contêineres de utilidade:**

version: '3.8'

services:

  npm:

    build: ./         #buildando e utilizando o Dockerfile do diretório

    volumes:

      - ./:/app       #Bind mount para gerar os arquivos localmente após docker-compose run

    tty: true         #flag -it

    stdin\_open: true  #flag -it

Desta forma, seguindo o que foi explicado acima, rodamos os seguintes comandos para o mesmo resultado:

* **docker-compose run npm init**
* **docker-compose run npm install**

**Importante:**

* O comando **docker-compose run [SERVICE\_NAME][COMMAND]** serve para rodar somente **um** serviço informado.
* **npm** se **refere ao nome do serviço** no docker-compose.

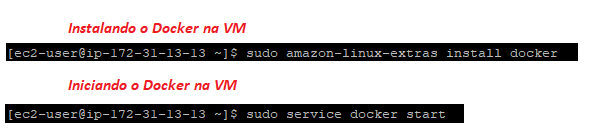
Deploy de Contêineres

EC2 (Amazon Elastic Compute Cloud):

Serve para criarmos uma VM hospedada na AWS, com qualquer OS que eles tenham a disposição.

**Conexão via SSH PuTTY – EC2:** [How to connect to EC2 w/ PuTTY (Windows) - YouTube](https://www.youtube.com/watch?v=bi7ow5NGC-U&t=235s)

Após termos o terminal da VM hospedada no AWS EC2 aberta, podemos executar os seguintes comandos para instalar e iniciar o Docker, respectivamente:



**Rodar a aplicação:** Basta subirmos a imagem desejada no DockerHub e executá-la na VM.

**Atualizar a aplicação:** Basta reuparmos a imagem atualizada no DockerHub, deletar a imagem antiga na VM, e usar a nova versão.

ECS (Amazon Elastic Container Service)

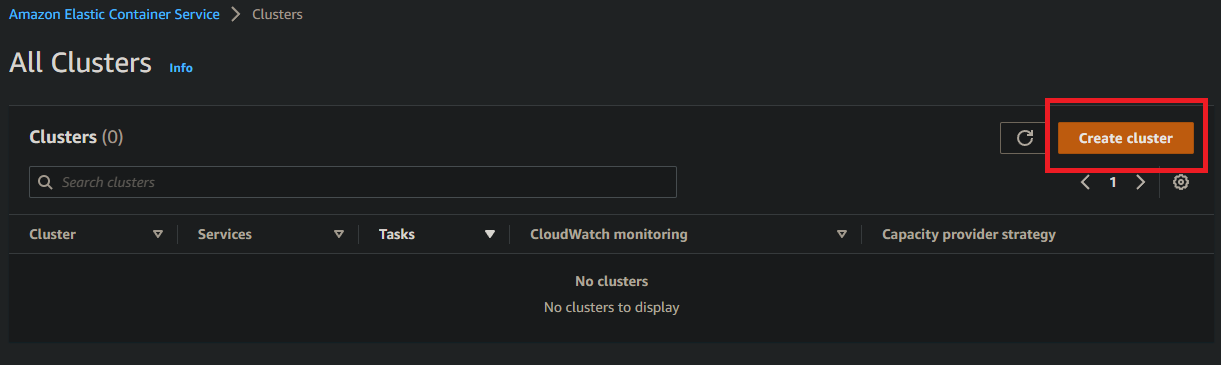
Serve para rodarmos contêineres na AWS, mas sem termos a liberdade de configurar 100% a máquina onde ele estará hospedado.

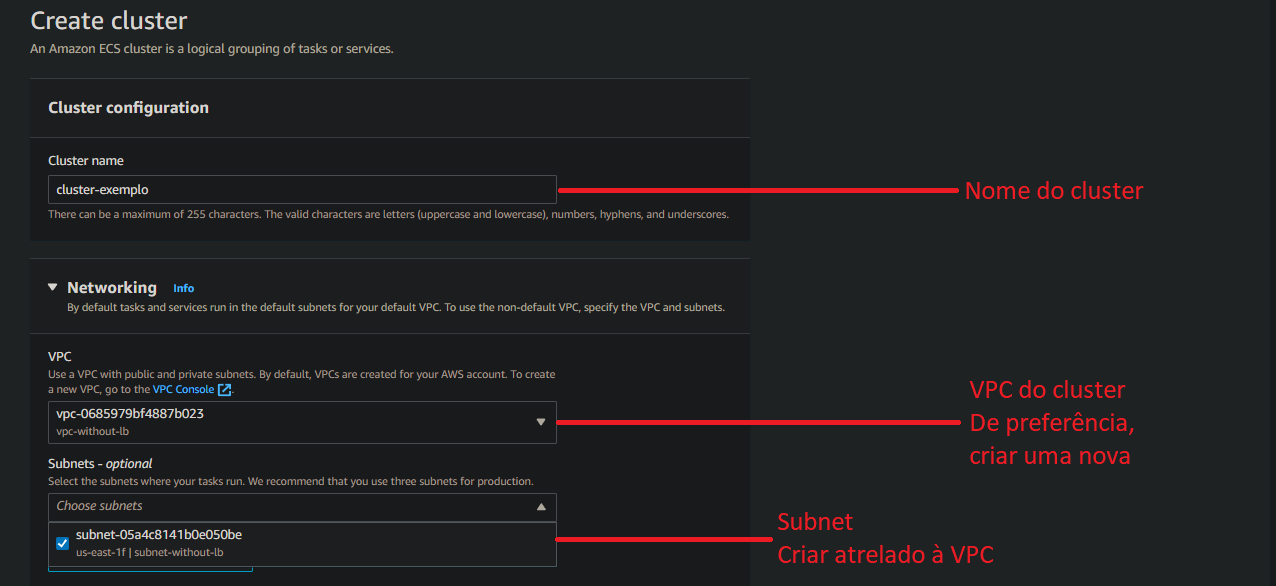
**Início das configurações:**

* Criar um **cluster**
* Criar uma **task-definition**
* Criar um **serviço** baseado na task-definition
* **No caso da adição de um load-balancer ao serviço**, devemos configurar **o Security Group (inbound e outbound rules), VPC, Subnets, e o Target Group**.
* **No caso da adição de um volume ao serviço**, devemos configurar um **EFS** na task-definition e atribuí-lo ao contêiner que possui o volume.

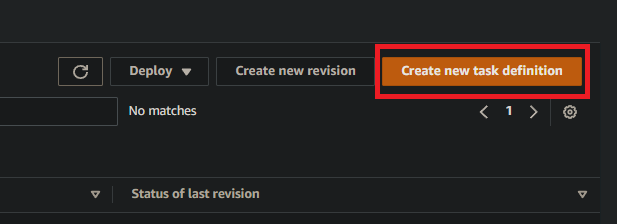
Abaixo, irei incluir um tutorial para subir dois contêineres no ECS **somente com as configurações essenciais**, com um backend em MongoDb e uma aplicação Nodejs.

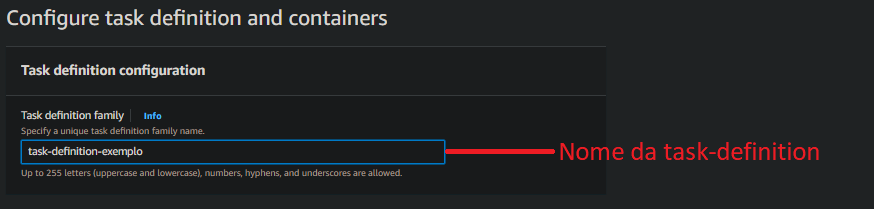
**Criando um cluster**

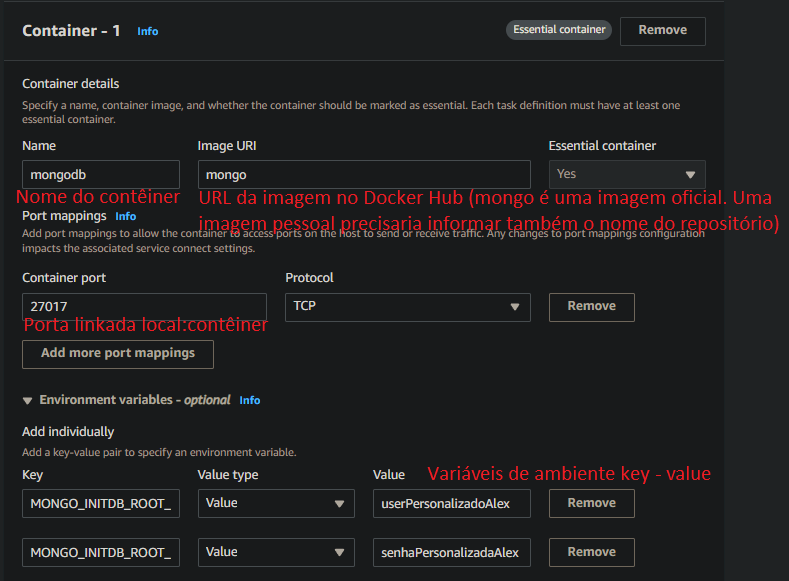
****

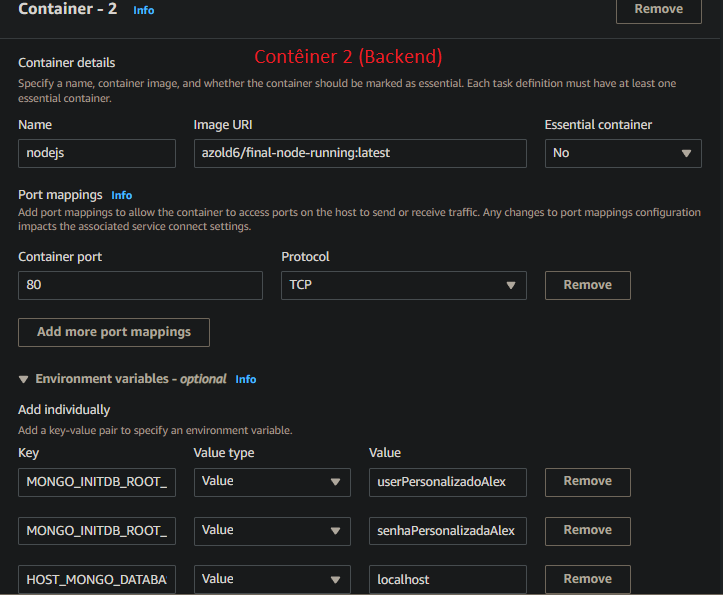
****

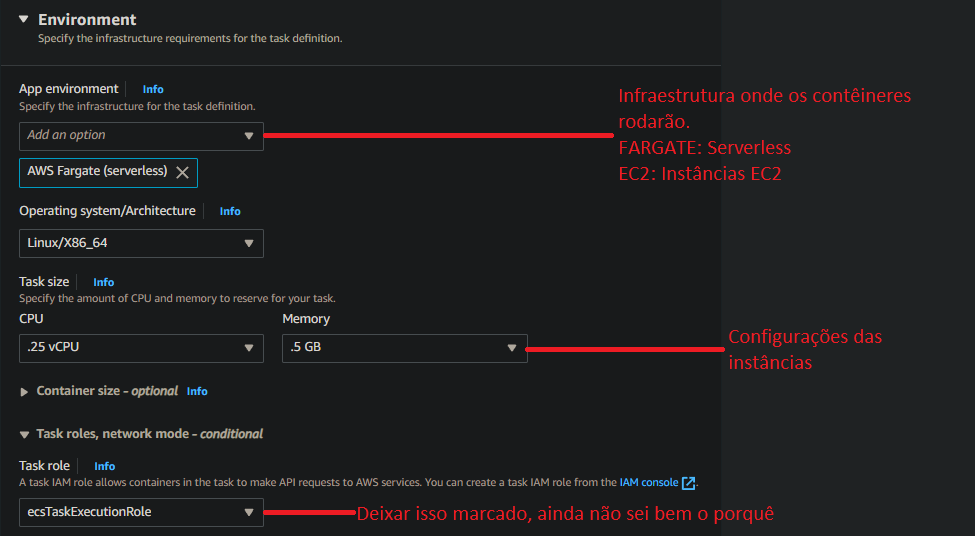
**Criando uma task-definition com os 2 contêineres**

****

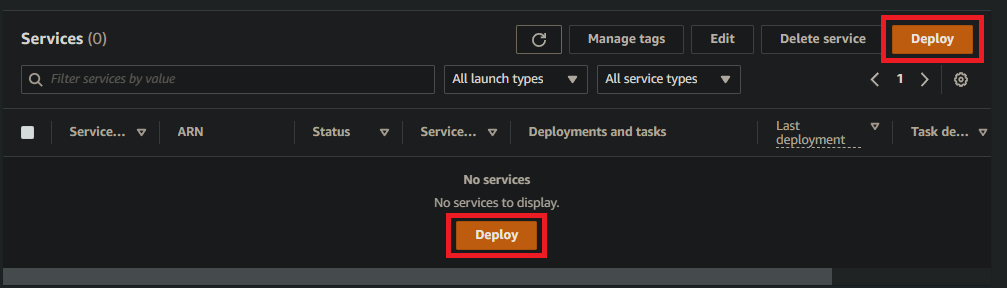
****

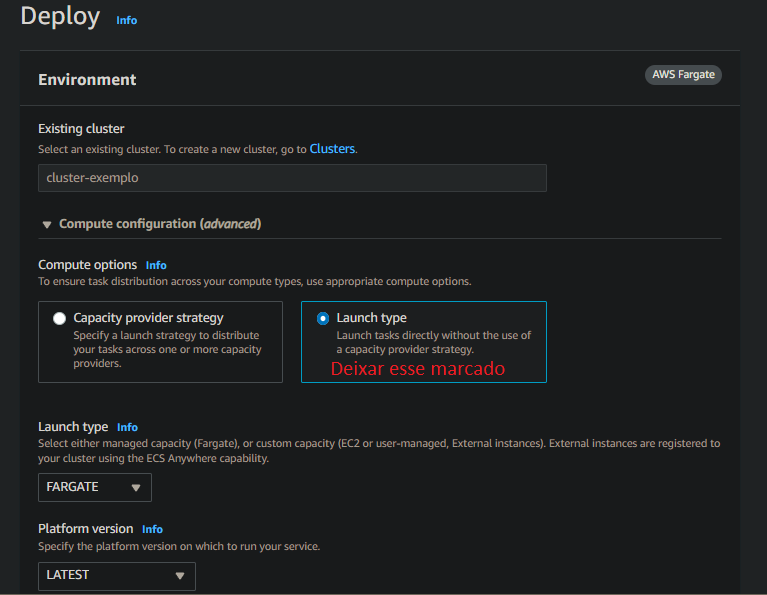
****

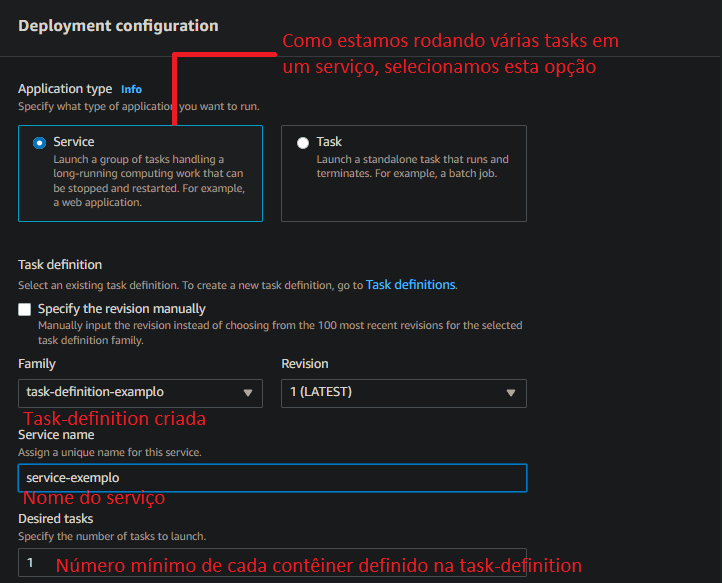
****

****

**Criando um serviço da task-definition**

****

****

****

****

Kubernetes









**Instalação Kubernetes (Aula 184)**

1. Instalar **minikube**
2. Instalar **kubectl**

**Comandos básicos minikube**

**minikube start –driver=[DRIVER\_NAME]** – Inicializa uma VM com o driver especificado. Drivers em: [Drivers | minikube (k8s.io)](https://minikube.sigs.k8s.io/docs/drivers/)

**minikube delete** – deleta a VM criada

**minikube status** – verifica o status da VM criada

**minikube dashboard** – abre uma aba no browser com um dashboard do Kubernetes

**Kubectl**

Kubectl (Kube control) é o controlador da máquina local que envia comandos ao Master Node do cluster Kubernetes.

**Comandos kubectl**

**kubectl get [ARG] –** Lista os objetos do argumento passado.

**Exemplos**

* **kubectl get deployments** – Lista todos os deployments do cluster
* **kubectl get pods** – Lista todas as Pods do cluster
* **kubectl get services** – Lista todos os serviços (objetos) criados no cluster
* **kubectl delete service [SERVICE\_NAME]** - Deleta o recurso
* **kubectl delete deployment [DEPLOYMENT\_NAME]** - Deleta o recurso

**kubectl create [OBJECT] [ARGS]** – Cria um objeto do Kubernetes.



**Objetos**

Os objetos pertencem ao ambiente Kubernetes e executam uma função nele.

**Objeto Deployments**

Serve para criarmos uma pod baseada em uma imagem do DockerHub.

**Criando um deployment**

**kubectl create deployment [DEPLOYMENT\_NAME] --image=[DEPLOYMENT\_IMAGE]** – cria um novo deployment e pod baseado na imagem informada.

**Exemplo de criação do deployment**

* **kubectl create deployment first-app --image=kub-first-app (COMANDO GERA ERRO!!!)**
* O comando acima gerará o erro **ErrImagePull**, já que os comandos do kubectl são executados dentro do cluster criado, que portanto **não terá acesso à uma imagem buildada no localhost**.
* Para que o comando execute corretamente, **é preciso que subamos a imagem para o DockerHub**, e utilizemos sua URL, como é demonstrado a seguir:
* **kubectl create deployment first-app --image=azold6/kub-first-app**

**Escalando um deployment**

**kubectl scale [OBJECT]/[OBJECT\_NAME] --replicas=[NUMBER\_OF\_DESIRED\_PODS]** - Escala um serviço (objeto kubernetes) para o número desejado de pods

**Exemplo**

* **kubectl scale deployment/first-app --replicas=3**

****

**Atualização de imagem em um Pod (deployment)**

1. Re-buildar a imagem desejada
2. Upá-la no DockerHub
3. Executar o comando **kubectl set image deployment/[DEPLOYMENT\_NAME] [CURRENT\_CONTAINER\_NAME]=[NEW\_IMAGE\_NAME\_ON\_DOCKERHUB]**

**IMPORTANTE!!!** A nova imagem será baixada **SOMENTE SE POSSUIR UM NOME OU TAG DIFERENTE DA IMAGEM ANTIGA DO POD.** Sendo assim, é uma boa prática versionar as imagens, como **imagem:1**, **imagem:2**...

1. Para visualizar o status da troca, executar **kubectl rollout status deployment/[DEPLOYMENT\_NAME]**

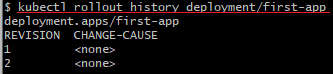
**Rollback de um deployment**

Para retornarmos para a versão anterior, devemos executar o comando abaixo:

* **kubectl rollout undo deployment/[DEPLOYMENT\_NAME]** - retorna à versão passada da Pod

Já para retornarmos para uma versão específica, devemos:

1. Executar **kubectl rollout history deployment/[DEPLOYMENT\_NAME]**

****

1. Selecionar a revisão para a qual deseja-se retornar. Pode-se verificar mais detalhes da revisão com o comando:

**kubectl rollout history deployment/[DEPLOYMENT\_NAME] --revision=[REVISION\_NUMBER]**

1. Executar **kubectl rollout undo deployment/[DEPLOYMENT\_NAME] --to-revision=[REVISION\_NUMBER]**

**Objeto Service**

Serve para agruparmos pods e atribuir a elas um IP fixo, além da possibilidade de expor as pods para a WWW.

**kubectl expose deployment [DEPLOYMENT\_NAME] –type=[TYPE] –port=[PORT]** - Cria um Service para expor uma porta do deployment informado

**Types:**

1. ClusterIP (só será ‘reachable’ de dentro do cluster)
2. NodePort (‘reachable’ de fora do cluster)
3. LoadBalancer (Gera um IP único para todas as pods e expõe ela, além de distribuir o tráfego)

Exemplo de exposição de portas de uma pod

**kubectl expose deployment first-app –type=LoadBalancer –port=8080**

Após a exposição, o comando **kubectl get services** irá expor os seguintes dados:



O **EXTERNAL-IP** é automaticamente atribuído ao serviço em um provedor cloud, mas no minikube, ainda é necessário executar o comando **minikube service first-app**

**‘Declarative Approach’**

Forma de evitar que precisemos executar múltiplos comandos para aplicar modificações no cluster. Comporta-se com um intuito similar ao docker-compose.

**Estrutura básica**

deployment.yaml

apiVersion: apps/v1

kind: Deployment ############## Tipo do objeto (neste caso, Deployment. Poderia ser um Service, Job, etc...)

metadata:

  name: second-app-deployment # Nome do objeto.

spec: ######################### ESPECIFICAÇÕES DO DEPLOYMENT

  replicas: 1 ################# Qtd. de pods do deployment.

  selector:

    matchLabels:

      app: second-app ######### Variável que auxilia o deployment a encontrar as pods que deve gerenciar.

      tier: backend ########### Neste caso, O deployment buscará pelo valor 'nginx' em spec.template.metadata.labels.app.

  ########### Se existir, funcionará adequadamente

template: ################################### **ESPECIFICAÇÕES DO POD. Em caso de confusão, consultar aula 198.**

    metadata:

      labels:

        app: second-app ##################### Local onde o selector.matchLabels busca pelas labels das aplicações.

        tier: backend

    spec:

      containers: ########################### Declaração dos contêineres que irão para as Pods (neste caso, 3)

        - name: second-node ################# Nome do contêiner

          image: academind/kub-first-app:2 ## Imagem a ser usada

        # - name: ... (Caso eu queira adicionar mais contêineres)

        #   image: ...

Para aplicar as definições descritas dentro do arquivo **deployment.yaml** acima, executamos o seguinte comando:

* **kubectl apply -f=[FILE\_NAME]**

Agora, precisamos de um objeto Service para expor as portas do Pod criado pelo **deployment** acima:

backend.yaml

apiVersion: v1

kind: Service ############## Tipo do objeto

metadata:

  name: backend ############ Nome do objeto

spec:

  selector:

    app: second-app ######## Selector para definir qual pod terá suas portas expostas por esse serviço.

  ports:            ######## O second-app foi definido no arquivo deployment.yaml

    - protocol: 'TCP'

      port: 80 ############# Porta externa

      targetPort: 8080 ##### Porta interna a ser exposta

  type: LoadBalancer ####### Tipo do serviço. Poderia ser ClusterIP, NodePort, etc...

Após executarmos **kubectl apply -f=[YAML\_NAME]**, basta executar **minikube service [EXPOSURE\_SERVICE\_NAME]** para acessarmos o Pod localmente.



Para atualizar as pods com uma nova imagem, ou adicionar mais replicas, basta alterar o a yaml do deployment e executar **kubectl apply -f=[YAML\_NAME]**

**Junção de objetos no mesmo arquivo**

Com o divisor ‘---’, é possivel colocar múltiplas instruções em um mesmo arquivo yaml, como segue o exemplo:

  # SERVICE DEFINITIONS…

  # ...

  ---

  # DEPLOYMENT DEFINITIONS…

  # ...

---

  # MORE DEFINITIONS…

  # ...

**Liveness probe**

Podemos sobreescrever a forma ‘padrão’ de verificar a saúde do contêiner. Para isso, utilizamos o **LivenessProbe**.

#ESPECIFICAÇÕES DO OBJETO ACIMA...

template: ################################# ESPECIFICAÇÕES DO POD.

    metadata:

      labels:

        app: second-app

        tier: backend

    spec:

      containers:

        - name: second-node

          image: azold6/kub-first-app:2

          livenessProbe:

            httpGet:

              path: / # Caminho onde a requisição "verificadora de saúde" deve ser enviada

              port: 8080 # Porta para a requisição

            periodSeconds: 10 # Intervalo de verificação após o início

            initialDelaySeconds: 5 #Intervalo de verificação inicial

**Volumes**

O Kubernetes permite que escolhamos uma infinidade de tipos de volumes.

**emptyDir**

O emptyDir permite que os dados de um diretório não sejam perdido na reiniciação de um contêner, criando um diretório vazio no diretório desejado ao inicializar.

# ...

    spec:

      containers:

        - name: story-backend

          image: azold6/kub-deploy-stor

          volumeMounts:

            - mountPath: /app/story # Caminho que não pode ser perdido no reiniciamento do contêiner

              name: story-volume # Linkagem do diretório desejado ao volume criado abaixo

      volumes:

        - name: story-volume # Nome do volume

          emptyDir: {} # Tipo do volume

**Problemática:** O emptyDir **inicializa atrelado a somente um pod**. Isso significa que, quando um pod precisa ser reiniciado, o dado não poderá ser acessado até que o Kubernetes o substitua por outra pod.

**hostPath**

Funciona como um **bind-mount**. Uma pasta no cluster é linkada a um diretório no contêiner.

# ...

    spec:

      containers:

        - name: story-backend

          image: azold6/kub-deploy-stor

          volumeMounts:

            - mountPath: /app/story # Caminho que não pode ser perdido no reiniciamento do contêiner

              name: story-volume # Linkagem do diretório desejado ao volume criado abaixo

      volumes:

        - name: story-volume # Nome do volume

          hostPath:

path: /data # Pasta dentro do cluster onde o bind-mount será feito

type: DirectoryOrCreate # Estratégia do bind-mount. Se o diretório /data não existir, ele será criado.

**Problemática:** O **hostPath** é independente de pods, mas não de nodes. Ou seja, o **volume armazenado aqui poderá ser utilizado por todas as pods daquele worker node, mas não por pods de outros worker nodes**.