

Aluno: Gabriel Sebold dos Santos;

Turma: 30;

Curso: Sistemas de Informação;

Atividade valendo 1 ponto na nota de atividades complementares.

1 - Criar uma aplicação utilizando multi-stage builds para otimizar a imagem final.

Passos:

Criar um Dockerfile que compila um programa Go em uma etapa e o executa em uma imagem base alpine na etapa final.

Testar o container gerado.

2 - Criar uma rede personalizada e permitir que dois containers se comuniquem usando nomes DNS.

Passos:

Criar uma rede bridge personalizada chamada minha-rede.

Executar dois containers: um usando a imagem nginx e outro usando a imagem busybox.

Testar a comunicação entre os containers utilizando ping.

3 - Configurar um volume para armazenar dados persistentes e reutilizá-lo entre containers.

Passos:

Criar um volume chamado dados-persistentes.

Criar um container que salva uma mensagem no volume em um arquivo.

Parar o container e iniciar outro container que acessa a mesma mensagem.

4 - Criar um site personalizado e funcional usando IA Generativa e em seguida executar usando Docker.

Parte 1:

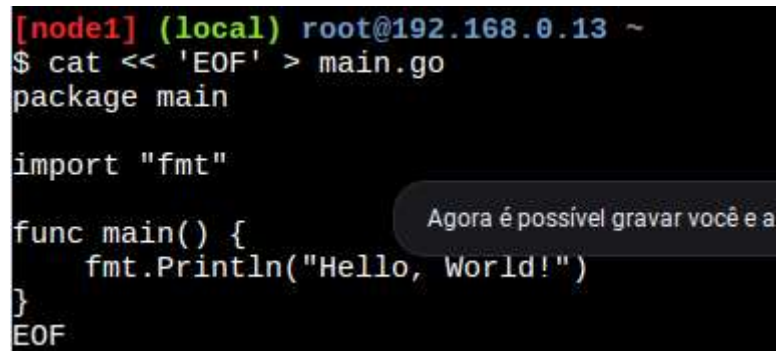
*** Criar uma aplicação utilizando multi-stage builds para otimizar a imagem final.**

Passos:

- Criar um Dockerfile que compila um programa Go em uma etapa e o executa em uma imagem base alpine na etapa final.

- Testar o container gerado.

Comandos



```
[node1] (local) root@192.168.0.13 ~
$ cat << 'EOF' > main.go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
EOF
```

Agora é possível gravar você e a

- CAT : está sendo usado para criar um arquivo com conteúdo digitado diretamente no terminal;
- << 'EOF' : Ele permite que você forneça várias linhas de texto como entrada para o comando cat, até encontrar uma linha que contenha somente EOF.
- > main.go : Redireciona a saída do cat para um novo arquivo chamado main.go. Se o arquivo já existir, ele será sobrescrito. O .go indica que o arquivo go está sendo criado na linguagem goolang.

Logo, tudo que estiver entre << 'EOF' e a palavra EOF será escrito dentro do arquivo main.

- package main : Todo programa executável em Go deve ter esse pacote principal. Indica que esse é o ponto de entrada da aplicação.
- import "fmt" : Importa o pacote fmt, que é usado para formatar e imprimir texto no terminal.
- func main() : A função principal onde o programa começa a ser executado. Toda aplicação Go precisa ter essa função main() no pacote main.
- fmt.Println("Hello, World!") : Comando que imprime a mensagem "Hello, World!" no terminal com uma quebra de linha no final.

```
$ cat << 'EOF' > Dockerfile
# Etapa 1: Compilação
FROM golang:1.18-alpine AS builder
RUN apk update && apk add --no-cache git
WORKDIR /app
COPY . .
RUN go mod init myapp || true
RUN go build -o myapp

# Etapa 2: Imagem final reduzida
FROM alpine:latest
WORKDIR /app
COPY --from=builder /app/myapp .
CMD ["/myapp"]
EOF
```

É criado um novo arquivo nomeado de Dockerfile.

Etapa 1:

- **FROM golang:1.18-alpine AS builder :**
Usa a imagem oficial do Golang baseada em Alpine Linux, e nomeia esta etapa de builder;
- **RUN apk update && apk add --no-cache git :**
Instala o git dentro do container.
apk é o gerenciador de pacotes do Alpine.
--no-cache evita armazenar cache para deixar a imagem menor.
- **WORKDIR /app :**
Define o diretório de trabalho no container como /app.
- **COPY . . :**
Copia todo o conteúdo do diretório atual para dentro do container na pasta /app
- **RUN go mod init myapp || true :**
Tenta inicializar o módulo Go com o nome **myapp**.
O (|| true) serve para não quebrar o build caso o módulo já exista.
É uma forma de garantir que o comando continue mesmo se der erro.
- **RUN go build -o myapp :**
Compila o código Go e gera um binário chamado **myapp**.

Etapa 2:

- **FROM alpine:latest:**
Começa uma nova imagem para rodar só o executável.
- **WORKDIR /app:**
Define o diretório de trabalho no container.
- **COPY --from=builder /app/myapp:**
Copia o binário gerado na etapa anterior para esta nova imagem.
- **CMD ["/myapp"]:**
Define o comando que será executado quando o container rodar.

```
[node1] (local) root@192.168.0.13 ~
$ docker build -t my-go-app .
[+] Building 20.3s (15/15) FINISHED                                docker:default
=> [internal] load build definition from Dockerfile                0.0s
=> => transferring dockerfile: 328B                                0.0s
=> [internal] load metadata for docker.io/library/alpine:latest    1.7s
=> [internal] load metadata for docker.io/library/golang:1.18-alpine 2.7s
=> [internal] load .dockerignore                                    0.0s
=> => transferring context: 2B                                       0.0s
=> [builder 1/6] FROM docker.io/library/golang:1.18-alpine@sha256:77f25981bd57e60a510165f3be89c901aec904 12.7s
=> => resolve docker.io/library/golang:1.18-alpine@sha256:77f25981bd57e60a510165f3be89c901aec90453fd0f1c5 0.0s
=> => sha256:a2f8637abd914a8a62416e027a351293d0472bc4b4f44383c6f425fd0e03861c 284.81kB / 284.81kB 0.5s
=> => sha256:4ba80a8cd2c7b1695ffb2166c58c8d0f0d4562c943fdb929d22467df250536bb 115.40MB / 115.40MB 2.4s
=> => sha256:77f25981bd57e60a510165f3be89c901aec90453fd0f1c5a45691f6cb1528807 1.65kB / 1.65kB 0.0s
=> => sha256:ab5685692564e027aa84e2980855775b2e48f8fc82c1590c0e1e8cbc2e716542 1.16kB / 1.16kB 0.0s
=> => sha256:a77f45e5f987fb7def8755903ad89fe37a38105dcf475be26550d7d86364e166 5.01kB / 5.01kB 0.0s
=> => sha256:8921db27df2831fa6eaa85321205a2470c669b855f3ec95d5a3c2b46de0442c9 3.37MB / 3.37MB 0.4s
=> => extracting sha256:8921db27df2831fa6eaa85321205a2470c669b855f3ec95d5a3c2b46de0442c9 0.2s
```

- **docker build -t my-go-app . :**
docker build :
É o comando que constrói uma imagem Docker a partir de um Dockerfile.
- **-t my-go-app :**
-t serve para dar um nome à imagem criada.
Esse nome será usado mais tarde para rodar o container com `docker run <nome da imagem>`.
- **.(ponto) :**
O ponto indica o diretório atual como contexto de build.
Isso significa que o Docker vai procurar o Dockerfile nesse diretório e copiar os arquivos que estão ali.

```
[node1] (local) root@192.168.0.13 ~
$ docker run --rm my-go-app
Hello, World!
```

- **docker run --rm my-go-app:**

Cria e executa um container temporário a partir da imagem my-go-app.
Roda o programa Go que você compilou (o binário myapp).
O container é automaticamente removido após a execução, graças ao --rm.

Assim se encerra a primeira parte retornando “Hello, World!”.

Parte 2:

*** Criar uma rede personalizada e permitir que dois containers se comuniquem usando nomes DNS.**

Passos:

- Criar uma rede bridge personalizada chamada minha-rede.
- Executar dois containers: um usando a imagem nginx e outro usando a imagem busybox.
- Testar a comunicação entre os containers utilizando ping.

Comandos

```
[node2] (local) root@192.168.0.13 ~  
$ docker network create minha-rede  
6fdc83420edcd99b7fe3ea529a111d6f571020370cab3b2ac194f4775e153648
```

- **docker network create:**

Cria uma rede virtual interna do Docker, onde containers podem se comunicar entre si de forma isolada. (no caso criei com o nome “minha-rede”)

```
[node2] (local) root@192.168.0.13 ~  
$ docker run -d --name meu-nginx --network minha-rede nginx  
Unable to find image 'nginx:latest' locally  
latest: Pulling from library/nginx  
6e909acdb790: Pull complete  
5eaa34f5b9c2: Pull complete  
417c4bccf534: Pull complete  
e7e0ca015e55: Pull complete  
373fe654e984: Pull complete  
97f5c0f51d43: Pull complete  
c22eb46e871a: Pull complete  
Digest: sha256:124b44bfc9ccd1f3cedf4b592d4d1e8bddb78b51ec2ed5056c5  
Status: Downloaded newer image for nginx:latest  
a010fac5b2feb58e0f558c3df1fac06dffb03c6e58c0ca04366f0aa90ede5a62  
[node2] (local) root@192.168.0.13 ~
```

- **docker run -d --name meu-nginx --network minha-rede nginx:**

-d

executa o container em segundo plano.

Você pode continuar digitando comandos enquanto o Nginx roda por trás.

--name meu-nginx

Dá um nome ao container: meu-nginx.

Isso facilita para referenciar o container depois, sem precisar usar o ID.

--network minha-rede

Conecta o container à rede Docker personalizada “minha-rede”.

Isso permite que ele se comunique com outros containers na mesma rede.

nginx

Nome da imagem oficial do Nginx.

O Docker vai baixar essa imagem e iniciar o Nginx.

```
[node2] (local) root@192.168.0.13 ~  
$ docker run -it --name busybox-teste --network minha-rede busybox  
Unable to find image 'busybox:latest' locally  
latest: Pulling from library/busybox  
97e70d161e81: Pull complete  
Digest: sha256:37f7b378a29ceb4c551b1b5582e27747b855bbfaa73fa11914fe0df028dc581f  
Status: Downloaded newer image for busybox:latest
```

- **docker run -it --name meu-busybox --network minha-rede busybox:**

-it:

Mantém o terminal interativo, permitindo que você insira comandos no container.

--name meu-busybox:

Atribui o nome meu-busybox ao container.

--network minha-rede:

Conecta o container à rede minha-rede.

busybox:

Imagem utilizada para criar o container.


```
--- meu-nginx ping statistics ---
4 packets transmitted, 4 packets received, 0% packet loss
round-trip min/avg/max = 0.105/0.147/0.198 ms
/ # ping -c 4 meu-nginx
PING meu-nginx (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: seq=0 ttl=64 time=0.137 ms
64 bytes from 172.20.0.2: seq=1 ttl=64 time=0.116 ms
64 bytes from 172.20.0.2: seq=2 ttl=64 time=0.106 ms
64 bytes from 172.20.0.2: seq=3 ttl=64 time=0.118 ms
```

- **ping -c 4 meu-nginx:**

ping:

Comando utilizado para testar a conectividade com outro host.

-c 4:

Parâmetro que indica ao ping para enviar exatamente 4 pacotes ICMP.

meu-nginx:

Nome do container de destino que você deseja alcançar.

Parte 3:

*** Configurar um volume para armazenar dados persistentes e reutilizá-lo entre containers.**

Passos:

- Criar um volume chamado dados-persistentes.
- Criar um container que salva uma mensagem no volume em um arquivo.
- Parar o container e iniciar outro container que acessa a mesma mensagem.

Comandos

```
[node1] (local) root@192.168.0.13 ~
$ docker volume create dados-persistentes
dados-persistentes
```

- **docker volume create dados-persistentes:**

docker volume create:

Comando utilizado para criar um volume no Docker.

dados-persistentes:

Nome atribuído ao volume que será criado.

- **docker run --name escritor -v dados-persistentes:/data busybox sh -c "echo 'Olá, mundo! Mensagem persistente.' > /data/mensagem.txt":**

docker run

Inicia um container.

--name escritor

Dá o nome escritor ao container.

-v dados-persistentes:/data

Usa o volume Docker chamado dados-persistentes.

Esse volume é montado no diretório /data dentro do container.

Tudo o que for gravado em /data será salvo no volume, ou seja, persiste mesmo depois que o container for removido.

busybox

É a imagem usada.

sh -c "echo '...' > /data/mensagem.txt"

Esse é o comando que o container executa:

Ele abre um shell (**sh -c**)

E executa `echo 'Olá, mundo! Mensagem persistente.' > /data/mensagem.txt`

Isso grava o texto dentro do volume, no arquivo /data/mensagem.txt

```
[node1] (local) root@192.168.0.13 ~  
$ docker run --rm -v dados-persistentes:/data busybox sh -c "cat /data/mensagem.txt"  
Olá, mundo! Mensagem persistente.
```

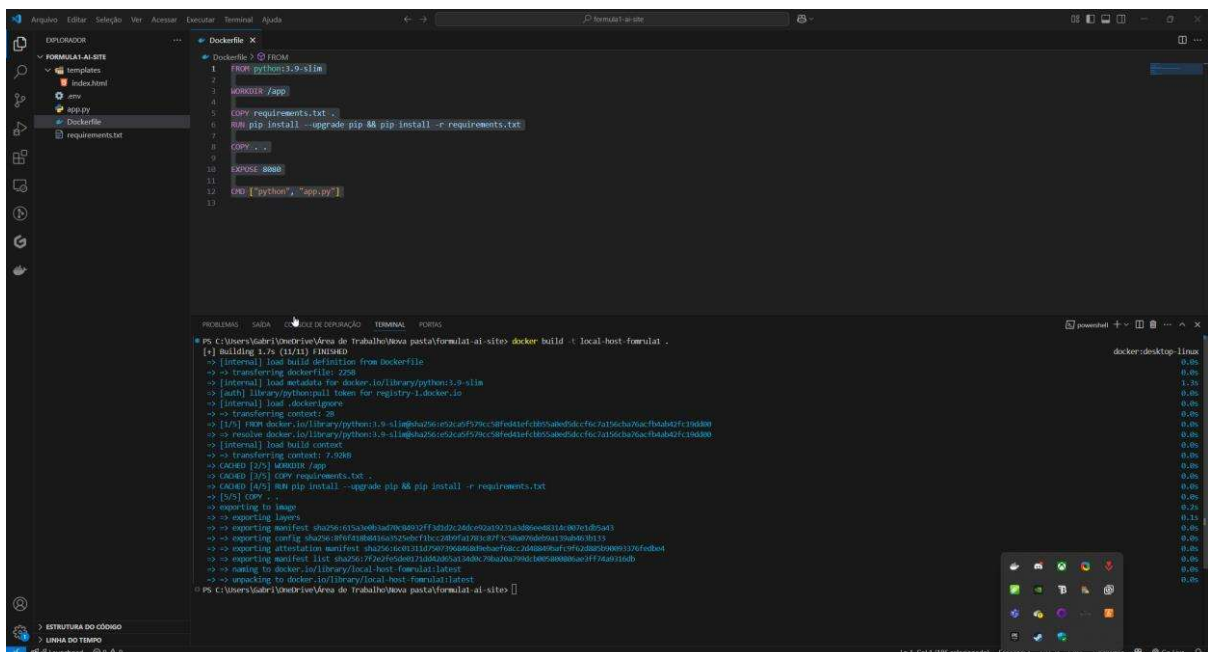

- **`docker run --rm -v dados-persistentes:/data busybox sh -c "cat > /data/mensagem.txt":`**

Aqui ele só chama o arquivo mensagem.txt, executa o que tem nele e remove o container após a execução.

Parte 4:

* Criar um site personalizado e funcional usando IA Generativa e em seguida executar usando Docker.

Comandos



Com base na requisição da parte quatro solicitei para a IA que a mesma gerasse um site simples relacionado a fórmula 1:

Ela tentou, criar algo com integração de API com o chat gpt, para que gerasse notícias relacionadas á fórmula 1 assim que apertasse o botão. (Mal sucedido).

O Dockerfile ficou assim:

“FROM python:3.9-slim

```
WORKDIR /app
```

COPY requirements.txt .

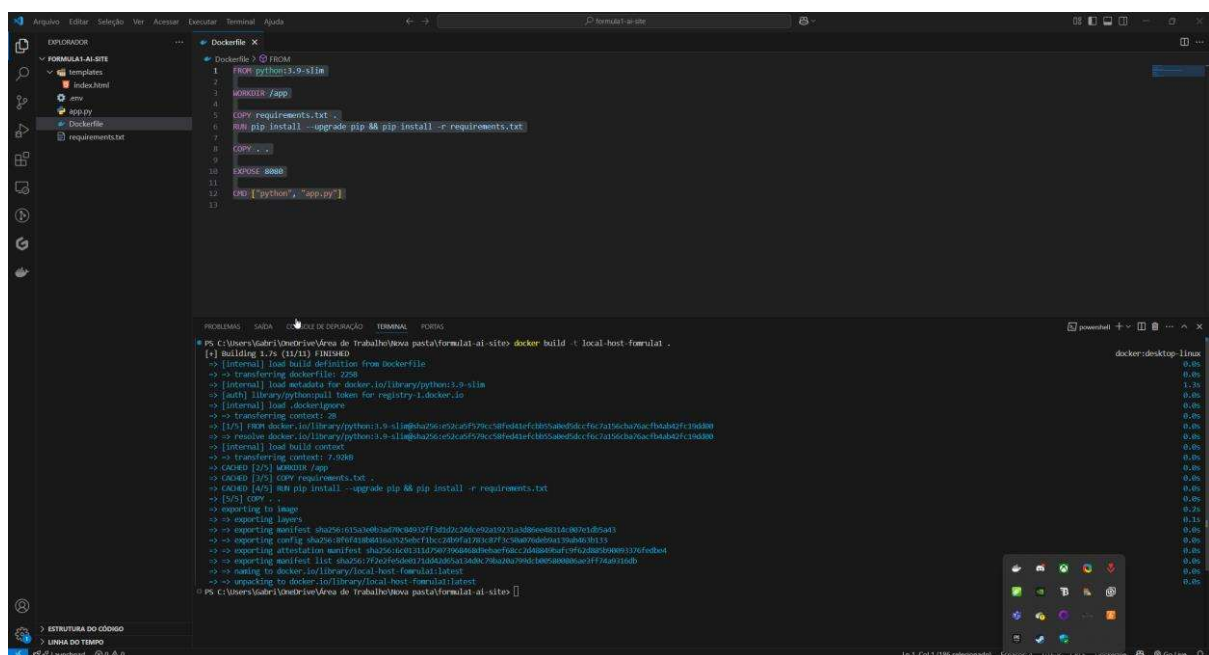
RUN `pip install --upgrade pip && pip install -r requirements.txt`

COPY ..

EXPOSE 8080

CMD ["python", "app.py"]

- Ele começa usando uma imagem do **Python 3.9**. Depois, ele define que o diretório principal dentro do container vai ser o **/app**, que é onde tudo vai acontecer.
- Aí ele copia o arquivo **requirements.txt**, que é onde geralmente ficam listadas as bibliotecas que o projeto precisa. Em seguida, ele atualiza o **pip** e instala tudo que tá no **requirements.txt**.
- Depois disso, ele copia todos os arquivos do projeto pra dentro do container. Ele também avisa que a aplicação vai rodar na **porta 8080**.
- Por fim, quando o container for iniciado, ele deve rodar o arquivo **app.py** com o **Python**.



- **docker build -t local-host-formula1 . :**

Esse comando cria uma imagem Docker personalizada a partir do seu projeto local, usando as instruções de um Dockerfile.

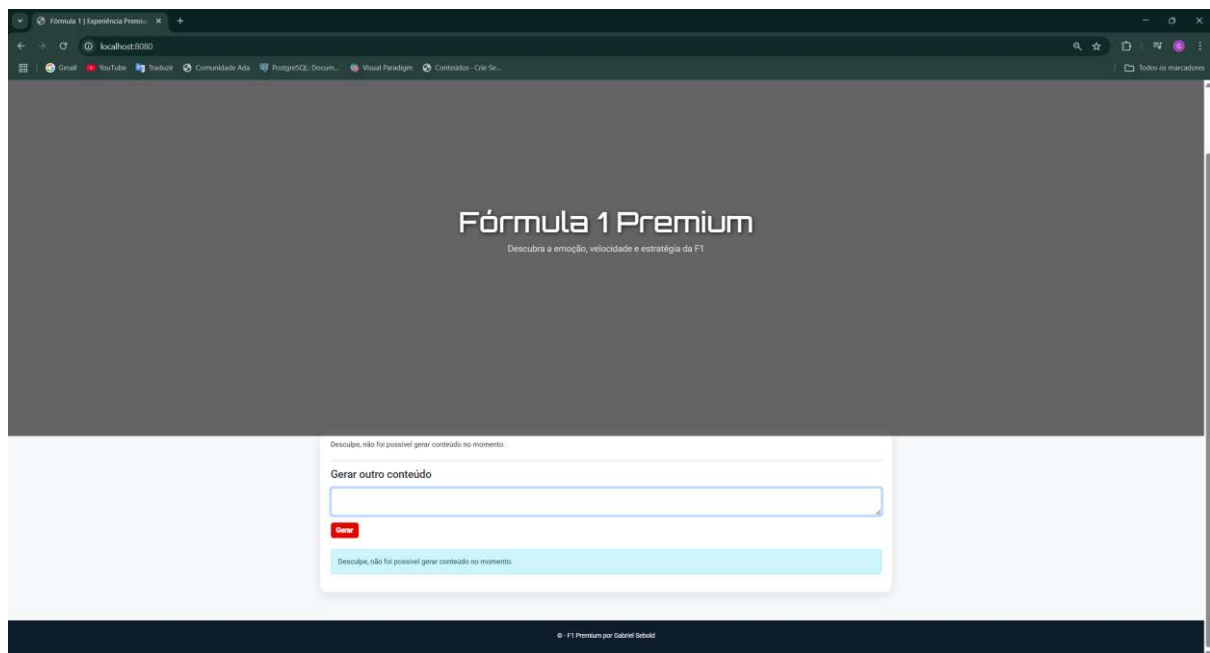
```
PS C:\Users\Gabri\OneDrive\Área de Trabalho\Nova pasta\formula1-ai-site> docker run -d -p 8080:8080 --name formula1-container formula1-ai-site 1c4d648365f9b31a5e877694ce088917083d66303de0762e140d4afb2e775337
```

- **docker run -d -p 8080:8080 --name formula1-container local-host-fomrula1:**

Esse comando executa um container Docker com sua aplicação já empacotada, deixando ela rodando em segundo plano e acessível pela porta 8080 do seu computador. O container recebe o nome formula1-container.

Após isso é só acessar a URL (<http://localhost:8080>) e temos um ambiente de testes rodando localmente.

E esse é o resultado:



E no Docker Desktop:

