Sistema Operacional de Redes 2
Avaliação-15

Nome: Carlos Abimael Oliveira do Nascimento

Turma: P8 de Informática

Pesquise e Responda as perguntas abaixo.

1) Como definir um volume no *Docker Compose* para persistir os dados do banco de dados *PostgreSQL* entre as execuções dos *containers*?

Utilizando a chave ***volumes***. O comando ***volumes*** permite mapear um diretório ou um volume para um caminho dentro do *container*.

Este é um exemplo de como fazer:

```
version: '3'
services:
  db:
    image: postgres
    volumes:
      - ./data:/var/lib/postgresql/data
    environment:
      - POSTGRES_PASSWORD=senha
```

Neste exemplo, estou criando um serviço chamado ***db*** usando a imagem do *PostgreSQL*. O volume é definido na seção ***volumes***, onde especificamos o caminho do diretório local ***./data*** e o caminho dentro do *container* ***/var/lib/postgresql/data***. Dessa forma, os dados do banco de dados serão persistidos no diretório local ***./data***.

Quando eu executo o comando ***docker-compose up***, o *Docker Compose* criará o volume e o mapeará para o caminho especificado dentro do *container*. Isso significa que os dados do banco de dados *PostgreSQL* serão armazenados no diretório local

./data e estarão disponíveis mesmo que você pare e inicie novamente os *containers*.

2) Como configurar variáveis de ambiente para especificar a senha do banco de dados *PostgreSQL* e a porta do servidor *Nginx* no *Docker Compose*?

Usando a chave ***environment*** dentro da definição de serviço no arquivo ***docker-compose.yml***.

Exemplo:

```
version: '3'
services:
  db:
    image: postgres
    environment:
      - POSTGRES_PASSWORD=minha_senha

  nginx:
    image: nginx
    ports:
      - 8080:80
    environment:
      - NGINX_PORT=80
```

No exemplo acima, tem dois serviços: ***db*** e ***nginx.*** Para o serviço ***db***, defini a variável de ambiente ***POSTGRES_PASSWORD*** e atribuí a senha desejada, ***minha_senha***. Essa variável será passada para o *container* do *PostgreSQL* como uma variável de ambiente.

Para o serviço ***nginx***, defini a variável de ambiente ***NGINX_PORT*** e atribuí o valor ***80***. Além disso, mapeei a porta 8080 do *host* para a porta 80 do *container* usando a chave ***ports***. Isso significa que o serviço *Nginx* estará acessível em ***http://localhost:8080***.

3) Como criar uma rede personalizada no *Docker Compose* para que os *containers* possam se comunicar entre si?

Use a chave ***networks*** no arquivo ***docker-compose.yml***.

Exemplo:

```yaml
version: '3'
services:
  app:
    image: minha_app
    networks:
      - minha_rede

  db:
    image: postgres
    networks:
      - minha_rede

networks:
  minha_rede:
    driver: bridge
```

Nesse exemplo, tem dois serviços: ***app*** e ***db***. O serviço ***app*** usa a imagem ***minha_app***` e o serviço `db` usa a imagem do PostgreSQL. Ambos os serviços estão associados à rede personalizada chamada ***minha_rede*** usando a chave ***networks***.

Além disso, na seção ***networks***, definimos a configuração para a rede ***minha_rede*** usando o driver ***bridge***. O driver ***bridge*** cria uma rede interna que permite a comunicação entre os *containers*.

Ao executar ***docker-compose up***, o *Docker Compose* criará a rede ***minha_rede*** e associará os serviços a ela. Os containers poderão se comunicar usando os nomes de serviço como *host*, por exemplo, o serviço ***app*** poderá se conectar ao serviço ***db*** usando o nome ***db*** como *host*.

4) Como configurar o *container Nginx* para atuar como um *proxy* reverso para redirecionar o tráfego para diferentes serviços dentro do *Docker Compose*?

Use um arquivo de configuração personalizado para o Nginx e monte-o como um volume dentro do container.

Exemplo:

1. Crie um arquivo chamado ***nginx.conf*** com a configuração do proxy reverso.

```
http {
    upstream app {
        server app1:8000;
        server app2:9000;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://app;
            proxy_set_header Host $host;
            proxy_set_header X-Real-IP $remote_addr;
        }
    }
}
```

Neste exemplo, estou configurando o Nginx para atuar como um proxy reverso redirecionando o tráfego para dois serviços, ***app1*** na porta 8000 e ***app2*** na porta 9000.

2. No arquivo ***docker-compose.yml***, adicione o serviço do *Nginx* e monte o arquivo ***nginx.conf*** como um volume:

```yaml
version: '3'
services:
  nginx:
    image: nginx
    ports:
      - 80:80
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - app1
      - app2

  app1:
    image: minha_app1
    # configurações do serviço app1

  app2:
    image: minha_app2
    # configurações do serviço app2
```

Neste exemplo, criei um serviço chamado ***nginx*** usando a imagem do *Nginx*. Mapeei a porta 80 do *host* para a porta 80 do *container* e montei o arquivo ***nginx.conf*** como ***/etc/nginx/nginx.conf*** dentro do *container*.

Certifique-se de ajustar as configurações dos serviços ***app1*** e ***app2*** de acordo com suas necessidades.

Ao executar ***docker-compose up***, o *Docker Compose* iniciará os serviços ***app1***, ***app2*** e ***nginx***. O *Nginx* atuará como um *proxy* reverso, redirecionando o tráfego para os serviços ***app1*** e ***app2*** com base nas configurações definidas no arquivo ***nginx.conf***.

Essa configuração permite que o *Nginx* funcione como um ponto de entrada para seus serviços dentro do *Docker Compose*, redirecionando solicitações com base nas regras definidas no arquivo de configuração.

5) Como especificar dependências entre os serviços no *Docker Compose* para garantir que o banco de dados *PostgreSQL* esteja totalmente inicializado antes do *Python* iniciar?

Pode-se usar a chave ***depends_on*** no arquivo ***docker-compose.yml***. No entanto, é importante destacar que o ***depends_on*** só controla a ordem de inicialização dos serviços, mas não garante que o serviço dependente esteja totalmente pronto antes que o serviço dependente seja iniciado.

Exemplo de como usar o ***depends_on*** no seu arquivo ***docker-compose.yml***:

```yaml
version: '3'
services:
  db:
    image: postgres
    # configurações do serviço db

  python:
    build: ./python
    depends_on:
      - db
    # configurações do serviço python
```

Neste exemplo, tem dois serviços: ***db*** e ***python***. O serviço ***python*** depende do serviço ***db*** porque precisa do banco de dados *PostgreSQL* para estar disponível antes de iniciar.

Ao usar ***depends_on***, o *Docker Compose* garantirá que o serviço ***db*** seja iniciado antes do serviço ***python***. No entanto, isso não garante que o banco de dados *PostgreSQL* esteja totalmente inicializado e pronto para aceitar conexões antes de o serviço ***python*** ser iniciado.

Uma abordagem comum para lidar com essa situação é escrever algum código no serviço ***python*** para verificar se o banco de dados está pronto antes de continuar a execução. Isso pode ser feito, por

exemplo, tentando se conectar ao banco de dados e aguardando até que a conexão seja estabelecida com sucesso.

Existem ferramentas como o ***wait-for-it*** ou scripts personalizados que podem ser usados para esperar até que um serviço esteja pronto antes de iniciar o próximo. Essas ferramentas podem ser adicionadas ao seu *container Python* como parte do processo de *build* ou execução.

Dessa forma, você pode garantir que o banco de dados ***PostgreSQL*** esteja totalmente inicializado antes de o ***Python*** iniciar, além de controlar a ordem de inicialização usando o ***depends_on***.

6) Como definir um volume compartilhado entre os *containers Python* e *Redis* para armazenar os dados da fila de mensagens implementada em *Redis*?

você pode usar a chave ***volumes*** no arquivo ***docker-compose.yml***. Isso permitirá que ambos os *containers* acessem e armazenem os dados no mesmo diretório.

Aqui está um exemplo de como fazer isso:

```yaml
version: '3'
services:
  python:
    build: ./python
    volumes:
      - mydata:/app/data
    # configurações do serviço python

  redis:
    image: redis
    volumes:
      - mydata:/data
    # configurações do serviço redis

volumes:
  mydata:
```

Neste exemplo, temos dois serviços: ***python*** e ***redis***. Ambos os serviços estão usando o mesmo volume chamado ***mydata***. O volume é definido na seção ***volumes*** fora da definição dos serviços.

Dessa forma, o volume ***mydata*** será criado e compartilhado entre os *containers* ***python*** e ***redis***. No serviço ***python***, o volume está mapeado para o diretório ***/app/data***, enquanto no serviço ***redis***, o volume está mapeado para o diretório ***/data***.

Isso permitirá que tanto o *container Python* quanto o *container Redis* acessem e armazenem os dados da fila de mensagens no mesmo diretório, facilitando a comunicação e o compartilhamento de dados entre os dois serviços.

7) Como configurar o *Redis* para aceitar conexões de outros *containers* apenas na rede interna do *Docker Compose* e não de fora?

Você pode ajustar a configuração de rede e as opções de segurança do Redis.

Aqui está um exemplo de como fazer isso no seu arquivo ***docker-compose.yml***:

```yaml
version: '3'
services:
  redis:
    image: redis
    command: redis-server --bind 0.0.0.0 --protected-mode yes
    ports:
      - 6379:6379
    networks:
      - minha_rede

networks:
  minha_rede:
    driver: bridge
```

Neste exemplo, o serviço Redis é configurado com as seguintes opções:

- O comando ***redis-server --bind 0.0.0.0*** é usado para fazer o Redis aceitar conexões de qualquer endereço IP. Ao usar ***0.0.0.0***, ele aceitará conexões de qualquer IP na rede interna do Docker.
- ***--protected-mode yes*** é definido para habilitar o modo protegido do Redis, o que significa que as conexões externas são desabilitadas por padrão.
- A porta ***6379*** é mapeada para o host para permitir o acesso externo ao Redis (opcionalmente, você pode remover essa linha se não precisar de acesso externo).

Além disso, é importante definir uma rede personalizada para os serviços no Docker Compose. No exemplo acima, foi criada uma rede chamada ***minha_rede*** com o driver ***bridge***.

Com essa configuração, o Redis estará configurado para aceitar conexões apenas dos containers dentro da rede interna definida pelo Docker Compose. As conexões externas serão desabilitadas por padrão devido ao modo protegido.

8) Como limitar os recursos de CPU e memória do container Nginx no Docker Compose?

Você pode usar as opções ***cpu_shares***, ***cpu_quota***, ***mem_limit*** e ***mem_reservation*** na definição do serviço Nginx no arquivo ***docker-compose.yml***.

Aqui está um exemplo de como fazer isso:

```yaml
version: '3'
services:
  nginx:
    image: nginx
    ports:
      - 80:80
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 256M
        reservations:
          cpus: '0.2'
          memory: 128M
```

Neste exemplo, estamos configurando limites de recursos para o serviço Nginx:

- ***limits.cpus: '0.5'*** define um limite de 0,5 núcleos de CPU para o container Nginx. Isso significa que o Nginx terá acesso a, no máximo, 50% de um núcleo de CPU.
- ***limits.memory: 256M*** define um limite de memória de 256 megabytes para o container Nginx.
- ***reservations.cpus: '0.2'*** define uma reserva de 0,2 núcleos de CPU para o container Nginx. Isso garante que o Nginx tenha acesso a pelo menos 20% de um núcleo de CPU.
- ***reservations.memory: 128M*** define uma reserva de memória de 128 megabytes para o container Nginx. Isso garante que o Nginx tenha acesso a pelo menos 128 megabytes de memória.

Essas configurações permitem limitar e reservar os recursos de CPU e memória do container Nginx no Docker Compose.

Ao usar o comando `docker-compose up`, o Docker Compose usará essas configurações para limitar os recursos do container Nginx de acordo com as especificações fornecidas.

9) Como configurar o container Python para se conectar ao Redis usando a variável de ambiente correta especificada no Docker Compose?

Você precisa definir a variável de ambiente corretamente no arquivo ***docker-compose.yml*** e acessá-la no código do seu aplicativo Python.

Aqui está um exemplo de como fazer isso:

1. No arquivo ***docker-compose.yml***, defina a variável de ambiente para a conexão do Redis no serviço Python:

```yaml
version: '3'
services:
  python:
    build: ./python
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
    # outras configurações do serviço python

  redis:
    image: redis
    # configurações do serviço redis
```

Neste exemplo, estamos definindo duas variáveis de ambiente no serviço Python: ***REDIS_HOST*** e ***REDIS_PORT***. O ***REDIS_HOST*** é definido como ***redis*** para usar o nome do serviço Redis como host, e o ***REDIS_PORT*** é definido como ***6379*** para especificar a porta do Redis.

2. No código do seu aplicativo Python, você pode acessar essas variáveis de ambiente para obter as informações de conexão do Redis. Dependendo do framework ou biblioteca que você está

usando para acessar o Redis, o método exato pode variar. Aqui está um exemplo genérico:

```
import os
import redis

redis_host = os.getenv('REDIS_HOST')
redis_port = os.getenv('REDIS_PORT')

# Conectar ao Redis usando as informações obtidas das variáveis de ambiente
r = redis.Redis(host=redis_host, port=redis_port)

# Use o objeto "r" para interagir com o Redis
```

Neste exemplo, usamos o módulo ***os*** para acessar as variáveis de ambiente ***REDIS_HOST*** e ***REDIS_PORT*** definidas no Docker Compose. Em seguida, usamos essas informações para criar um objeto ***Redis*** e nos conectarmos ao Redis.

Certifique-se de ajustar o código do seu aplicativo para usar corretamente as informações de conexão obtidas das variáveis de ambiente.

Ao executar o ***docker-compose up***, o Docker Compose definirá as variáveis de ambiente no serviço Python conforme especificado no arquivo ***docker-compose.yml***, permitindo que o container Python se conecte corretamente ao Redis usando as informações fornecidas.

10) Como escalar o container Python no Docker Compose para lidar com um maior volume de mensagens na fila implementada em Redis?

Você pode usar a funcionalidade de escalabilidade do Docker Compose, que permite executar várias instâncias do serviço Python simultaneamente.

Aqui está um exemplo de como fazer isso:

1. No seu arquivo ***docker-compose.yml***, adicione a opção ***scale*** ao serviço Python, especificando o número desejado de instâncias:

```
version: '3'
services:
  python:
    build: ./python
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
    # outras configurações do serviço python

  redis:
    image: redis
    # configurações do serviço redis
```

2. Execute o seguinte comando para escalar o serviço Python:

```
docker-compose up --scale python=<número_de_instâncias>
```

Substitua ***<número_de_instâncias>*** pelo número desejado de instâncias que você deseja executar. Por exemplo, se você quiser executar três instâncias do serviço Python, você pode usar:

```
docker-compose up --scale python=3
```

Dessa forma, o Docker Compose criará e iniciará várias instâncias do serviço Python, permitindo que elas lidem com um maior volume de mensagens na fila implementada em Redis.

Cada instância do serviço Python terá acesso às mesmas variáveis de ambiente definidas no ***docker-compose.yml***, incluindo as informações de conexão do Redis. Certifique-se de que o código do seu aplicativo Python seja projetado para lidar corretamente com várias instâncias e a distribuição do trabalho.

Ao dimensionar o serviço Python dessa forma, você estará distribuindo a carga de trabalho entre várias instâncias, permitindo que você processe um maior volume de mensagens na fila.