Lucas da Silva Pereira - P8 - INFORMÁTICA
Sistemas Operacionais de Redes 2

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

Resposta:

Para definir um volume persistente (não volátil) no Docker Compose para armazenar (persistir) os dados de um banco de dados PostgreSQL, você pode usar a seção "volumes" no arquivo "docker-compose.yml".

Por exemplo:

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

Os dados ficarão armazenados no ./data do host, assim continuarão persistindo lá mesmo quando os containers forem desligados, reiniciados, etc.

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

Resposta:

No Docker Compose, utiliza-se a propriedade "environment" e dentre outras dentro do arquivo "docker-compose.yml" para configurar variáveis de ambiente com o fim de especificar a senha do banco de dados PostgreSQL, bem como a porta do servidor Nginx. Como no exemplo:

```
version: '3'
services:
  db:
    image: postgres
    environment:
      POSTGRES_PASSWORD: mypassword

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

Na variável "POSTGRES_PASSWORD: mypassword" você pode alterar a senha do Banco de Dados Postgres.
E na variável "NGINX_PORT: 80" é possível alterar a o mapeamento das portas do Server Nginx.

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

Resposta:

Para criar uma rede personalizada no Docker Compose e que permita a comunicação entre os containers, utiliza-se a propriedade networks dentro do arquivo "docker-compose.yml". Por exemplo:

```
version: '3'
services:
  db:
    image: postgres
    networks:
      - mynetwork
nginx:
    image: nginx
    networks:
      - mynetwork
networks:
  mynetwork:
```

No exemplo acima está sendo criada uma rede personalizada chamada mynetwork usando a seção networks no nível superior do "arquivo docker-compose.yml". Atribui-se o serviço "db" e o serviço "nginx" a essa rede usando a propriedade networks em cada serviço. Conectando, assim os containers "db" e "nginx" a essa mesma rede personalizada, possibilitando a comunicação.

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

Resposta:

Para configurar o container Nginx como um proxy reverso que redirecione o tráfego para diferentes serviços dentro do Docker Compose, precisa-se criar um arquivo de configuração para o Nginx que define as regras de redirecionamento.

Primeiro cria-se um arquivo de configuração do Nginx chmado "nginx.conf". Como no exemplo:

```
worker_processes auto;

events {
    worker_connections 1024;
}

http {
    server {
        listen 80;

        location /service1 {
            proxy_pass http://service1:8000;
        }

        location /service2 {
```

```
        proxy_pass http://service2:9000;
    }
  }
}
```

No exemplo está sendo criado um server nginx que atende na porta 80 e em seguida configuradas duas localizações "service1" e "service2" na porta 8000 e 9000, respectivamente.

Atualiza-se o arquivo docker-compose.yml para incluir o serviço do Nginx e montar o arquivo de configuração personalizado. Exemplo:

```
version: '3'
services:
  nginx:
    image: nginx
    ports:
      - 80:80
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - service1
      - service2
service1:
    # Configurações do serviço 1
service2:
    # Configurações do serviço 2
```

Nas configurações acima, define-se o serviço nginx que usa a imagem do Nginx. A propriedade ports mapeia a porta 80 do container Nginx para a porta 80 do host. A propriedade volumes monta o arquivo nginx.conf local no caminho /etc/nginx/nginx.conf dentro do container Nginx. Além disso, deve-se substituir o "service1" e o "service2" pelos nomes escolhidos.

Após essas etapas, executa-se o comando "docker-compose up" para iniciar os serviços definidos no arquivo docker-compose.yml.
Esses serviços incluirão o nginx configurado como proxy reverso.

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?

Resposta:

Para especificar dependências entre serviços no Docker Compose e ainda garantir que o banco de dados PostgreSQL esteja totalmente inicializado antes do Python iniciar, é usada a propriedade "depends_on" no arquivo "docker-compose.yml".

O depends_on não aguarda a inicialização completa do serviço dependente, apenas garante a ordem de

inicialização. Exemplo:

```
version: '3'
services:
  db:
    image: postgres
    # Configurações do banco de dados PostgreSQL

  python:
    build: ./python-app
    depends_on:
      - db
    # Configurações da aplicação Python
```

No exemplo, o python está dependente do serviço "db" (banco de dados postgres totalmente inicializado). O "depends_on" fará com que o "db" inicie sempre antes do python.

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

Resposta:

Para definir um volume compartilhado entre os containers Python e Redis no Docker Compose e armazenar os dados da fila de mensagens implementada em Redis, pode se usar a propriedade "volumes" no arquivo "docker-compose.yml". Exemplo:

```
version: '3'
services:
  python:
    build: ./python-app
    volumes:
      - message_queue:/app/data

  redis:
    image: redis
    volumes:
      - message_queue:/data

volumes:
  message_queue:
```

No exemplo há dois serviços: python e redis. A propriedade volumes está sendo usada para criar um volume chamado "message_queue". Esse volume é compartilhado entre os

dois serviços, permitindo que eles acessem o mesmo local de armazenamento.

No serviço python, o volume "message_queue" está sendo montado em /app/data, o que significa que qualquer dado armazenado nesse diretório dentro do container será persistido no volume compartilhado.

No serviço redis, o volume message_queue está sendo montado em /data, permitindo que o Redis armazene seus dados da fila de mensagens nesse diretório, que também é compartilhado com o serviço python. Dessa forma, os dados da fila de mensagens implementada no Redis serão armazenados no volume compartilhado, permitindo que tanto o container Python quanto o container Redis acessem e manipulem esses dados.

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

Resposta:

**É necessário definir o arquivo "docker-compose.yml"
Certificar-se de que o arquivo docker-compose.yml esteja configurado corretamente para criar o serviço Redis.
Exemplo:**

```
version: '3'
services:
  redis:
    image: redis
    ports:
      - '6379:6379'
```

A porta 6379 do container Redis é mapeada para a porta 6379 do host. Para restringir as conexões ao Docker Compose, não precisa-se fazer o mapeamento de porta para o host.

Para impedir as conexões externas ao Redis, é necessário remover o mapeamento de porta do serviço Redis no arquivo "docker-compose.yml". Modificando a seção ports do serviço Redis, o resultado é:

```
services:
  redis:
```

```
    image: redis
    # ports:
    #   - '6379:6379'
```

Ao comentar esse mapeamento com o "#", ele é removido, evitando conexões externas.

Agora cria-se uma rede personalizada, precisa criar uma rede personalizada no Docker Compose para que os containers possam se comunicar internamente. As linhas a seguir devem ser adicionadas ao arquivo "docker-compose.yml":

```
networks:
  redis_network:
```

Conecta-se o serviço Redis à rede personalizada. Por fim, conecta-se o serviço Redis à rede personalizada que foi criada. Modificando a seção do serviço Redis para incluir a opção networks como no exemplo:

```
services:
  redis:
    image: redis
    networks:
      - redis_network
    # ports:
    #   - '6379:6379'
```

Assim, o Redis está conectado apenas à rede personalizada.

Por fim, é necessário executar o comando "docker-compose up" para iniciar os serviços.

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

Resposta:

Para limitar os recursos de CPU e memória do container Nginx no Docker Compose, usa-se as opções "cpus" e "mem_limit".

Dentro da seção do serviço Nginx no arquivo "docker-compose.yml", adiciona-se as opções "cpus" e "mem_limit" para definir os limites de CPU e memória, respectivamente. Por exemplo, limitar o Nginx a 0,5 CPUs e 512 MB de memória.

```
services:
  nginx:
    image: nginx
    ports:
      - '80:80'
    cpus: '0.5'
    mem_limit: 512m
```

Por fim, inicia-se os serviços Após adicionar as opções de

limite de recursos no arquivo "docker-compose.yml", é necessário salvar o arquivo e iniciar seus serviços usando o comando "docker-compose up -d".

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

Resposta:

Certificar-se de que o arquivo "docker-compose.yml" está configurado corretamente para criar os serviços do Redis e do Python. Por exemplo:

```
version: '3'
services:
  redis:
    image: redis

  python:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - REDIS_HOST=redis
```

Nesse exemplo, há dois serviços: redis e python. O serviço python usa um arquivo Dockerfile para criar a imagem do Python e também define a variável de ambiente REDIS_HOST com o valor redis, que é o nome do serviço Redis definido no

arquivo "docker-compose.yml".

Acessar a variável de ambiente no código Python. No código Python, pode-se acessar a variável de ambiente REDIS_HOST para se conectar ao Redis corretamente. Exemplo:

```
import os
import redis

redis_host = os.getenv('REDIS_HOST', 'localhost')
redis_port = 6379

# Conecte-se ao Redis
r = redis.Redis(host=redis_host, port=redis_port)

# Realize operações no Redis, se necessário
```

A variável redis_host obtém o valor da variável de ambiente REDIS_HOST usando os.getenv('REDIS_HOST', 'localhost'). Se a variável de ambiente não estiver definida, o valor padrão localhost será usado. E necessário instalar as dependências:

```
COPY requirements.txt .
RUN pip install -r requirements.txt
```

Após isso, inicia-se os serviços com o comando "docker-compose up".

10) Como escalar o container Python no Docker Compose

para lidar com um maior volume de mensagens na fila implementada em Redis?

Resposta:

Para dimensionar o serviço Python, usar a opção "scale "no Docker Compose. Adiciona-se as seguintes linhas de comando abaixo da definição do serviço python no arquivo docker-compose.yml:

```
services:
  python:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      - REDIS_HOST=redis
    deploy:
      replicas: 3
```

No exemplo, há 3 como número de réplicas, o que significa 3 instâncias do serviço Python em execução.

Inicia-se os serviços escalados após adicionar a opção de escala no arquivo "docker-compose.yml", salva e inicia os serviços escalados usando o comando "docker-compose up -d –scale python=3". Isso instrui o Docker Compose a iniciar três instâncias do serviço Python.