Nome : Abraão Rodrigues de Lima
Informática - SOR2 - Taveira

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

Adicione a seguinte configuração ao seu arquivo ‘docker-compose.yml’:

```
version: '3'
services:
  db:
    image: postgres
    volumes:
      - db_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_DB=mydatabase
      - POSTGRES_USER=myuser
      - POSTGRES_PASSWORD=mypassword

volumes:
  db_data:
```

1. A seção ‘services’ define os serviços do Docker Compose, neste caso, apenas o banco de dados PostgreSQL (‘db’).

2. A imagem do PostgreSQL é especificada com ‘image: postgres’.

3. A configuração ‘volumes’ é usada para definir um volume chamado ‘db_data’ que será mapeado para o diretório ‘/var/lib/postgresql/data’ dentro do contêiner. É nesse diretório que os dados do PostgreSQL serão armazenados.

4. A seção ‘environment’ é usada para definir as variáveis de ambiente do PostgreSQL, como o nome do banco de dados, usuário e senha.

5. A seção ‘volumes’ fora do escopo do 'services' é usada para declarar o volume ‘db_data’.

Dessa forma, o volume ‘db_data’ será persistido mesmo quando os containers forem reiniciados ou recriados, garantindo que os dados do banco de dados sejam mantidos entre as execuções.

Certifique-se de executar o comando ‘docker-compose up’ na mesma pasta em que o arquivo 'docker-compose.yml' está localizado para iniciar os serviços definidos no Docker Compose.

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

Adicione a seguinte configuração ao seu arquivo 'docker-compose.yml':

```
version: '3'
services:
  db:
    image: postgres
    environment:
      - POSTGRES_PASSWORD=${DB_PASSWORD}

  nginx:
    image: nginx
    ports:
      - ${NGINX_PORT}:80
    environment:
      - NGINX_PORT=${NGINX_PORT}
```

1. Na seção 'services', temos dois serviços: 'db' para o PostgreSQL e 'nginx' para o Nginx.

2. No serviço 'db', a configuração 'environment' é usada para definir a variável de ambiente 'POSTGRES_PASSWORD' com o valor '${DB_PASSWORD}'. Essa sintaxe '${DB_PASSWORD}' indica que o valor será fornecido por meio de uma variável de ambiente externa ao arquivo 'docker-compose.yml'.

3. No serviço 'nginx', a configuração 'ports' é usada para mapear a porta '${NGINX_PORT}' do host para a porta 80 do contêiner Nginx. Novamente, a sintaxe '${NGINX_PORT}' indica que o valor será fornecido por uma variável de ambiente externa.

4. A configuração 'environment' do serviço 'nginx' é usada para definir a variável de ambiente 'NGINX_PORT' com o valor '${NGINX_PORT}'.

Ao executar o comando 'docker-compose up', você precisará garantir que as variáveis de ambiente 'DB_PASSWORD' e 'NGINX_PORT' estejam definidas no ambiente em que você está executando o Docker Compose. Você pode definir essas variáveis de ambiente antes de executar o comando ou armazená-las em um arquivo '.env' na mesma pasta do 'docker-compose.yml'.

Exemplo de um arquivo '.env':

```
DB_PASSWORD=secretpassword
NGINX_PORT=8080
```

Dessa forma, as variáveis de ambiente serão configuradas corretamente no Docker Compose, permitindo que você especifique a senha do banco de dados PostgreSQL e a porta do servidor Nginx de forma flexível e personalizada.

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

Adicione a seguinte configuração ao seu arquivo ‘docker-compose.yml’:

```
version: '3'
 services:
    service1:
       image: myimage1
       networks:
          - mynetwork

service2:
       image: myimage2
       networks:
          - mynetwork

networks:
   mynetwork:
```

1. Na seção 'services', temos dois serviços: ‘service1’ e ‘service2’. Substitua ‘myimage1’ e ‘myimage2’ pelas imagens Docker que você deseja usar para cada serviço.
2. Em cada serviço, a configuração ‘networks’ é usada para especificar a rede à qual o contêiner pertence. No exemplo acima, ambos os serviços (‘service1’ e ‘service2’) estão conectados à rede ‘mynetwork’.
3. A seção ‘networks’ fora do escopo dos serviços é usada para declarar a rede personalizada ‘mynetwork’.

Ao usar essa configuração, o Docker Compose criará automaticamente a rede ‘mynetwork’ e conectará os contêineres dos serviços ‘service1’ e ‘service2’ a essa rede. Os contêineres poderão se comunicar entre si usando os nomes dos serviços como endereços de host, pois o Docker Compose define automaticamente um sistema de DNS interno para os serviços.

Dessa forma, os containers nos serviços ‘service1’ e ‘service2’ podem se comunicar entre si usando os nomes dos serviços como hosts, por exemplo, ‘http://service’1 ou ‘http://service2’.

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

Segue estas etapas:

1. Crie um arquivo de configuração personalizado para o Nginx que definirá as regras de proxy reverso. Por exemplo, crie um arquivo chamado 'nginx.conf' com o seguinte conteúdo:

```
events {}

http {
  server {
    listen 80;

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

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

Neste exemplo, definimos duas regras de proxy reverso usando as diretivas 'location'. O tráfego recebido em '/service1' será redirecionado para 'http://service1:8000', e o tráfego recebido em '/service2' será redirecionado para 'http://service2:9000'. Certifique-se de substituir 'service1' e 'service2' pelos nomes dos serviços reais do seu Docker Compose e as portas pelos respectivos serviços.

2. No seu arquivo 'docker-compose.yml', adicione o serviço do Nginx e monte o arquivo de configuração personalizado:

```
version: '3'
services:
  nginx:
    image: NGINX
    doors:
      - 80:80
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf
    depends_on:
      - service1
      - service2

service1:
    image: MyImage1
```

```
  service2:
    image:MyImage2
```

Neste exemplo, adicionamos o serviço ‘nginx’ ao Docker Compose, vinculamos a porta ‘80’ do host à porta ‘80’ do contêiner Nginx e montamos o arquivo de configuração personalizado ‘nginx.conf’ em ‘/etc/nginx/nginx.conf’ dentro do contêiner.

Certifique-se de substituir ‘myimage1’ e ‘myimage2’ pelas imagens Docker reais que você deseja usar para seus serviços.

3. Inicie o Docker Compose executando o comando ‘docker-compose up’.

O Nginx será iniciado e configurado como um proxy reverso de acordo com as regras definidas no arquivo ‘nginx.conf’. O tráfego recebido em ‘/service1’ será redirecionado para o serviço ‘service1’, e o tráfego recebido em ‘/service2’ será redirecionado para o serviço ‘service2’.

Certifique-se de que seus serviços ‘service1’ e ‘service2’ estejam definidos corretamente no Docker Compose para que o Nginx possa redirecionar o tráfego corretamente.

Lembre-se de que você pode personalizar as regras de proxy reverso no arquivo ‘nginx.conf’ de acordo com suas necessidades, adicionando ou modificando as diretivas ‘location’.

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?

Use a diretiva ‘depends_on’ no seu arquivo 'docker-compose.yml'. No entanto, vale ressaltar que o ‘depends_on’ não garante a disponibilidade total do serviço dependente, apenas o inicia antes do serviço que depende dele.

Aqui está um exemplo de como você pode configurar as dependências:

```
version: '3'
Services:
  db:
   image: Postgres
   environment:
    - POSTGRES_DB=mydatabase
    - POSTGRES_USER=myuser
    - POSTGRES_PASSWORD=mypassword

python:
   build: ./path/to/python
   depends_on:
    - db
```

Neste exemplo, temos dois serviços: ‘db’ para o PostgreSQL e ‘python’ para a aplicação Python. O serviço ‘python’ tem uma dependência declarada em relação ao serviço ‘db’ usando a diretiva ‘depends_on’. Isso significa que o serviço ‘db’ será iniciado antes do serviço ‘python’.

No entanto, o ‘depends_on’ não verifica se o serviço dependente está pronto para aceitar conexões. É importante ter em mente que o banco de dados PostgreSQL pode levar algum tempo para ser totalmente inicializado e aceitar conexões.

Dessa forma, o serviço Python será iniciado após o serviço PostgreSQL, mas é recomendado que sua aplicação Python implemente uma lógica de espera ou tentativas de conexão para garantir que o banco de dados esteja pronto para receber conexões antes de iniciar sua aplicação.

Você pode usar bibliotecas como ‘psycopg2’ ou ‘sqlalchemy’ no seu código Python para lidar com as tentativas de conexão e garantir que o banco de dados esteja pronto antes de estabelecer a conexão.

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

Adicione a seguinte configuração ao seu arquivo ‘docker-compose.yml’:

```
version: '3'
services:
  Python:
    build: ./path/to/python
    volumes:
      - message_queue_data:/app/date

redis:
    Image: Redis
    volumes:
      - message_queue_data:/date

volumes:
  message_queue_data:
```

1. Na seção ‘services’, temos dois serviços: 'python' para o contêiner Python e ‘redis’ para o Redis.

2. No serviço ‘python’, a configuração ‘volumes’ é usada para definir um volume chamado ‘message_queue_data’ que será montado no diretório ‘/app/data’ dentro do contêiner Python. Esse diretório será usado para armazenar os dados da fila de mensagens.

3. No serviço 'redis', também usamos a configuração ‘volumes’ para montar o mesmo volume ‘message_queue_data’ no diretório ‘/data’ dentro do contêiner Redis. Assim, os dados da fila de mensagens serão compartilhados entre os dois containers.

4. A seção ‘volumes’ for a do escopo dos serviços é usada para declarar o volume ‘message_queue_data’.

Dessa forma, tanto o container Python quanto o container Redis terão acesso ao mesmo volume ‘message_queue_data’, permitindo que os dados da fila de mensagens sejam compartilhados entre eles. Isso significa que qualquer alteração nos dados da fila feita por um container será refletida e acessível pelo outro container.

Certifique-se de executar o comando ‘docker-compose up’ na mesma pasta em que o arquivo ‘docker-compose.yml’ está localizado para iniciar os serviços definidos no Docker Compose.

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

Adicione a seguinte configuração ao seu arquivo 'docker-compose.yml':

```
version: '3'
services:
  redis:
    image: Redis
    command: redis-server --bind 0.0.0.0
    ports:
      - 6379:6379
```

1. Na seção 'services', temos apenas o serviço 'redis' para o Redis.

2. A imagem do Redis é especificada com 'image: redis'.

3. A configuração 'command' é usada para executar o comando 'redis-server --bind 0.0.0.0' dentro do contêiner Redis. Isso faz com que o Redis escute em todas as interfaces de rede, incluindo a interface de rede interna do Docker Compose.

4. A configuração 'ports' é usada para mapear a porta '6379' do host para a porta '6379' do contêiner Redis. Isso permite que você acesse o Redis a partir do host usando 'localhost:6379' ou '127.0.0.1:6379'.

Com essa configuração, o Redis estará disponível para conexões somente dentro da rede interna do Docker Compose. Os outros containers no mesmo Docker Compose poderão se conectar ao Redis usando o nome do serviço 'redis' como host e a porta '6379'.

No entanto, vale ressaltar que a porta '6379' do Redis estará exposta no host, o que significa que outros processos ou serviços em execução no host também poderão acessar o Redis usando 'localhost:6379' ou '127.0.0.1:6379'. Se você deseja restringir o acesso do Redis apenas aos containers no Docker Compose e não ao host, pode remover a configuração 'ports' do serviço 'redis'. Dessa forma, o Redis será acessível apenas dentro do Docker Compose.

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

Utilize as configurações ‘cpu_quota’, ‘cpu_period’, ‘mem_limit’ e ‘mem_reservation’ no seu arquivo ‘docker-compose.yml’. Aqui está um exemplo de como adicionar essas configurações:

```
version: '3'
services:
  nginx:
    image:Nginx
    ports:
      - 80:80
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: 512M
        reservations:
          cpus: '0.2'
          memory: 256M
```

1. Na seção ‘services’, temos o serviço ‘nginx’ para o container Nginx.

2. A imagem do Nginx é especificada com ‘image: nginx’.

3. A configuração ‘ports’ é usada para mapear a porta ‘80’ do host para a porta ‘80’ do contêiner Nginx.

4. Dentro do serviço ‘nginx’, a configuração ‘deploy’ é usada para definir as configurações de recursos.

5. A seção ‘resources’ define os limites e reservas de recursos.
    - ‘limits’ define os limites máximos de recursos:
        1. ‘cpus’ define a quantidade máxima de CPUs que o container Nginx pode utilizar, sendo '0.5' equivalente a 50% de uma CPU.
        2. ‘memory’ define o limite máximo de memória que o container Nginx pode utilizar, sendo '512M' equivalente a 512 megabytes.
    - ‘reservations’ define as reservas mínimas de recursos:
        1. ‘cpus’ define a quantidade mínima de CPUs que o container Nginx reserva para garantir a disponibilidade, com '0.2' sendo equivalente a 20% de uma CPU.
        2. ‘memory’ define a reserva mínima de memória que o container Nginx reserva para garantir a disponibilidade, com '256M' sendo equivalente a 256 megabytes.
        3.

Dessa forma, você está definindo limites e reservas de CPU e memória para o container Nginx no Docker Compose. Essas configurações ajudarão a controlar o uso de recursos pelo Nginx e garantir que ele não exceda os limites definidos, fornecendo uma distribuição equilibrada de recursos no ambiente de execução do Docker Compose.

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

Segue estes passos:

1. No seu arquivo 'docker-compose.yml', adicione uma variável de ambiente para o serviço Python, especificando a conexão com o Redis. Por exemplo:

```
version: '3'
services:
  python:
    build: ./path/to/python
    environment:
      - REDIS_HOST=Redis
      - REDIS_PORT=6379
```

Neste exemplo, adicionamos duas variáveis de ambiente: 'REDIS_HOST' e 'REDIS_PORT'. Definimos 'REDIS_HOST' como 'redis', que é o nome do serviço Redis no Docker Compose, e 'REDIS_PORT' como '6379', a porta em que o Redis está sendo executado.

2. No código Python da sua aplicação, você pode acessar essas variáveis de ambiente para obter as informações de conexão com o Redis. Aqui está um exemplo de como você pode fazer isso usando a biblioteca 'redis-py':

```
Import os
Import redis

redis_host = os.environ.get('REDIS_HOST')
redis_port = int(os.environ.get('REDIS_PORT'))

r = redis. Redis(host=redis_host, port=redis_port)
```

Neste exemplo, estamos usando a biblioteca 'redis-py' para criar uma conexão com o Redis. As variáveis 'REDIS_HOST' e 'REDIS_PORT' são obtidas das variáveis de ambiente usando 'os.environ.get()'.

Certifique-se de que a biblioteca 'redis-py' esteja instalada no seu ambiente Python. Você pode instalá-la usando o 'pip':

```
pip install redis
```

Dessa forma, você está configurando o container Python para se conectar ao Redis usando as variáveis de ambiente especificadas no Docker Compose. Isso permite que você defina as informações de conexão com o Redis de forma flexível e reutilizável, facilitando a alteração dessas informações sem a necessidade de modificar diretamente o código Python.

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

Use a funcionalidade de dimensionamento horizontal do Docker Compose. Isso permitirá que você crie várias instâncias do serviço Python para distribuir a carga de trabalho.

Aqui está um exemplo de como escalar o serviço Python:

```
version: '3'
services:
  python:
    build: ./path/to/python
    environment:
      - REDIS_HOST=redis
      - REDIS_PORT=6379
    deploy:
      replicas: 3
```

1. Na seção 'services', temos o serviço python para o container Python.

2. A imagem do Python e o ambiente estão especificados com 'build' e 'environment', respectivamente.

3. Dentro do serviço 'python', a configuração 'deploy' é usada para definir as opções de escalabilidade.

4. A configuração 'replicas' é definida como '3', o que significa que o serviço Python será replicado em três instâncias.

Ao usar o comando 'docker-compose up' com essa configuração, o Docker Compose iniciará três instâncias do serviço Python, permitindo que você distribua a carga de trabalho e aumente a capacidade de processamento para lidar com um maior volume de mensagens na fila.

Cada instância do serviço Python estará usando as mesmas variáveis de ambiente, permitindo que todas se conectem ao mesmo Redis e consumam as mensagens da fila de forma distribuída.

Certifique-se de que sua aplicação Python esteja preparada para lidar com a escalabilidade horizontal. Dependendo da sua implementação específica, você pode precisar implementar estratégias de balanceamento de carga e coordenação entre as instâncias para garantir o processamento adequado das mensagens da fila.