

# MÓDULO 15 – DOCKER –

## ATIVIDADE 06

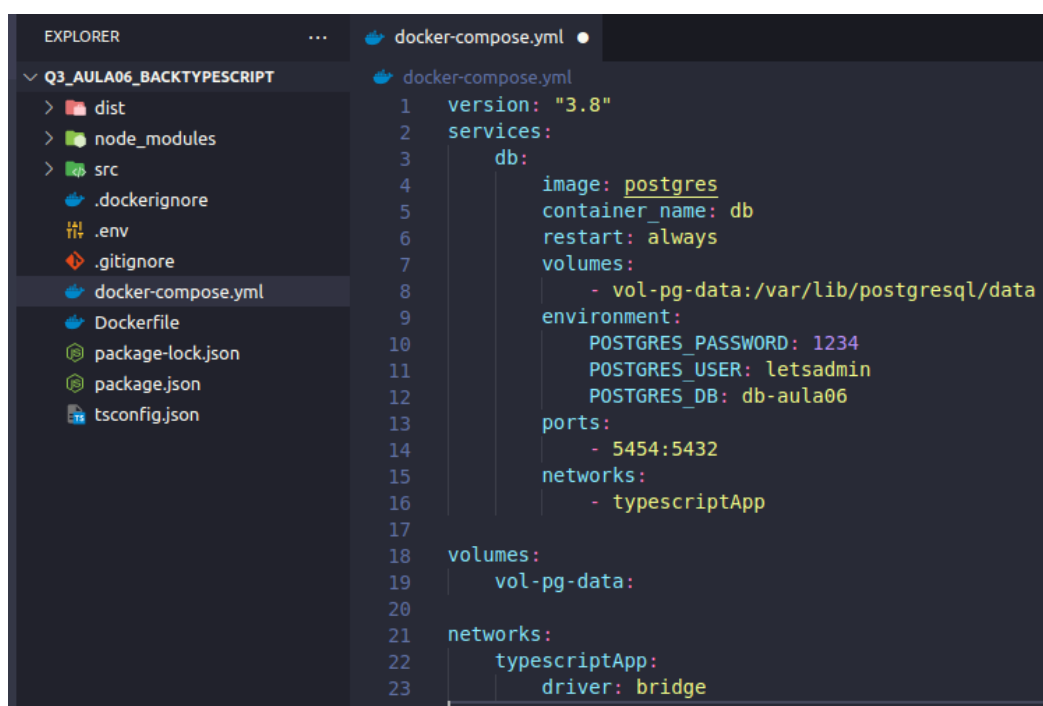
### 1.OBJETIVO

Vamos rodar o backend em typescript (feito na aula 06 de typescript) com banco de dados Postgresql, mas desta vez usando `docker compose` para subir os dois serviços sem necessidade de usar `docker run` separadamente para cada um. Para isso, siga os passos abaixo (muitos detalhes omitidos). Você deve mostrar todos os comandos realizados e o resultado deles (mesmo que não explicitamente pedido), de sorte que seja possível acompanhar passo-a-passo sua execução e verificar que tudo funcionou.

### 2.RESULTADOS E DISCUSSÃO

#### 2.1. SERVIÇO 1 – BANCO DE DADOS POSTGRES

O primeiro passo foi criar um arquivo docker-compose.yml na raiz do diretório da aplicação backend-typescript e escrever a configuração de um serviço “db responsável pelo database postgresQL:



```
1 version: "3.8"
2 services:
3   db:
4     image: postgres
5     container_name: db
6     restart: always
7     volumes:
8       - vol-pg-data:/var/lib/postgresql/data
9     environment:
10      POSTGRES_PASSWORD: 1234
11      POSTGRES_USER: letsadmin
12      POSTGRES_DB: db-aula06
13     ports:
14       - 5454:5432
15     networks:
16       - typescriptApp
17
18 volumes:
19   vol-pg-data:
20
21 networks:
22   typescriptApp:
23     driver: bridge
```

Explicando as informações contidas no docker-compose.yml:

A configuração version define a versão do docker compose, o que é importante por questão de compatibilidade. Por exemplo, a versão '3.8' é compatível com Docker Engine v19.03.0 ou superior (ver <https://docs.docker.com/compose/compose-file/compose-versioning/#version-3>). Define-se um serviço chamado "db", que não deve ser confundido com o nome do container. Por praticidade, optou-se por colocar o mesmo nome no container. Define-se a imagem base como postgres, disponível publicamente no Docker Hub. Caso o container pare de funcionar, ele reinicia automaticamente. Um volume é criado para persistir os dados mesmo que o container seja destruído ou reiniciado. Através da propriedade environment, define-se variáveis de ambiente que serão passadas. Trata-se de um padrão do postgres informar o PASSWORD, as demais não são obrigatórias. A propriedade ports mapeia a porta 5454 do host (máquina onde o container está sendo executado) para a porta 5432 dentro do container, permitindo o acesso pelo host. Cria-se uma rede do tipo bridge e o container recebe o nome dessa rede, fazendo parte dela. É importante destacar que caso não tivéssemos informado uma rede, o próprio docker compose cria uma rede e permite troca de informações entre os containers através do nome do serviço.

Com o terminal na mesma posição (raiz do diretório do projeto), para iniciar o serviço definido no arquivo docker-compose.yml, em segundo plano, aplicou-se o comando a seguir:

### **docker compose up -d**

```
letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/repositorioHard/Alpha-edtech-cycle01/Module15-Docker/Aula06/Q3_aula06_backtypescript$ docker compose up -d
[+] Running 3/3
  :: Network q3_aula06_backtypescript_typescriptApp Created 0.9s
  :: Volume "q3_aula06_backtypescript_vol-pg-data" Created 0.1s
  :: Container db Started 8.9s
letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/re
```

Como foi utilizada a flag "-d", não temos os logs do processo e após concluída a inicialização dos serviços, o terminal fica disponível para uso. A próxima imagem ilustra o comportamento quando não aplicamos a flag "-d".

```
letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/repositorioHard/Alpha-edtech-cycle01/Module15-Docker/Aula06/Q3_aula06_backtypescript$ docker compose up
[+] Running 3/3
  :: Network q3_aula06_backtypescript_typescriptApp Created 0.6s
  :: Container backend-web Created 7.9s
  :: Container db Created 8.3s
Attaching to backend-web, db
db
db | PostgreSQL Database directory appears to contain a database; Skipping initialization
db
db | 2023-05-22 12:21:29.656 UTC [1] LOG: starting PostgreSQL 15.2 (Debian 15.2-1.pgdg110+1) on x86_64-pc-linux-gnu, compiled by gcc (Debian 10.2.1-6) 10.2.1 20210110, 64-bit
db | 2023-05-22 12:21:29.657 UTC [1] LOG: listening on IPv4 address "0.0.0.0", port 5432
db | 2023-05-22 12:21:29.657 UTC [1] LOG: listening on IPv6 address ":::", port 5432
db | 2023-05-22 12:21:29.875 UTC [1] LOG: listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
db | 2023-05-22 12:21:30.074 UTC [29] LOG: database system was shut down at 2023-05-22 12:09:28 UTC
db | 2023-05-22 12:21:30.200 UTC [1] LOG: database system is ready to accept connections
backend-web | Server is running on https://0.0.0.0:80
db | 2023-05-22 12:26:30.171 UTC [27] LOG: checkpoint starting: time
db | 2023-05-22 12:26:32.871 UTC [27] LOG: checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.653 s, sync=0.341 s, total=2.701 s; sync files=2, longest=0.171 s, average=0.171 s; distance=0 kB, estimate=0 kB
█
```

Para confirmar que o container foi criado e está em execução, podemos executar esse comando:

### **docker compose ps**

```

letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/repositorioHard/Alpha-edtech-cycle01/Module15-Dock
ktypescript$ docker compose ps
NAME                IMAGE              COMMAND                  SERVICE      CREATED          STATUS
db                  postgres          "docker-entrypoint.s..." db           2 minutes ago   Up 2 minutes
2/tcp, :::5454->5432/tcp

```

Nota-se que o estado está up.

Para acessar o client psql dentro do container “db”, utiliza-se o comando a seguir:

**docker exec -it db psql -U letsadmin -d db-aula06**

Detalhando o comando acima:

**docker exec** permite usar um comando dentro de um container ativo; a **flag “-it”** habilita a interação via terminal, permitindo a inserção de comandos e visualização da saída; o **“db”** é o nome do container (poderia ter sido usado o ID do container, por exemplo) que desejamos conectar; **psql** é um cliente que permite interagir com bancos de dados que são geridos pelo sistema de gerenciamento de banco de dados relacional PostgreSQL; a **flag “-U”** especifica o nome do usuário usado na conexão, nesse caso optou-se pelo superusuário definido anteriormente; a **flag “-d”** aponta o nome do banco de dados que desejamos conectar e que está dentro do container.

```

letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/repositorioHard/Alpha-edtech-cy
ktypescript$ docker exec -it db psql -U letsadmin -d db-aula06
psql (15.2 (Debian 15.2-1.pgdg110+1))
Type "help" for help.

db-aula06=#

```

No schema da tabela accounts, há uma chave primária do tipo UUID. Optou-se por adicionar um valor padrão através da função `uuid_generate_v4()` que atribui um valor válido de UUID caso não tenha sido informado algum valor para esse campo na hora da adição de um novo elemento. Por conta disso, antes de criar a tabela, faz-se necessária a adição de uma extensão que permitirá o uso dessa função. No terminal do psql utilizou-se esse comando:

**CREATE EXTENSION IF NOT EXISTS "uuid-oss";**

```

db-aula06=# CREATE EXTENSION IF NOT EXISTS "uuid-oss";
CREATE EXTENSION
db-aula06=#

```

A próxima etapa é copiar e colar os comandos definidos anteriormente e que contém o nome da tabela do nosso banco de dados e a adição de elementos.

CREATE TABLE accounts (

```

    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),

```

```

updated_at TIMESTAMP,
deleted_at TIMESTAMP,
deleted BOOLEAN DEFAULT FALSE
);
INSERT INTO accounts (
    name,
    email,
    password
)
VALUES
('lets','lets@gmail.com','123'),
('maria','maria@gmail.com','222'),
('joao','joao@gmail.com','333'),
('felipe','felipe@gmail.com','444');
SELECT * FROM accounts;

```

As próximas imagens mostram o resultado obtido no terminal do psql:

```

db-aula06=# CREATE TABLE accounts (
    id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
    name TEXT NOT NULL UNIQUE,
    email TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP,
    deleted_at TIMESTAMP,
    deleted BOOLEAN DEFAULT FALSE
);
CREATE TABLE
db-aula06=# 

```

```

db-aula06=# INSERT INTO accounts (
    name,
    email,
    password
)
VALUES
('lets','lets@gmail.com','123'),
('maria','maria@gmail.com','222'),
('joao','joao@gmail.com','333'),
('felipe','felipe@gmail.com','444');
INSERT 0 4
db-aula06=# 

```

Através do comando “**SELECT \* FROM accounts;**” é possível visualizar uma tabela com todas as informações da tabela accounts.

```
db-aula06=# SELECT * FROM accounts;
```

id	name	email	password	created_at	updated_at
71e429df-c850-40ce-b90b-b3d837a00177	lets	lets@gmail.com	123	2023-05-22 02:15:06.976012	
6b675d5f-9879-4430-ac16-e50fc7e0d224	maria	maria@gmail.com	222	2023-05-22 02:15:06.976012	
5676b1d1-1c53-4031-bcb8-2dc765d3ba8e	joao	joao@gmail.com	333	2023-05-22 02:15:06.976012	
25b37958-4fe6-4a63-936d-6607fa0715d2	felipe	felipe@gmail.com	444	2023-05-22 02:15:06.976012	

```
(4 rows)
db-aula06=#
```

Caso necessário limpar o terminal do psql, basta utilizar “\! clear”. Para encerrar a conexão via psql, podemos usar “\q” .

Mostrou-se que o banco de dados está funcionando e a tabela “accounts” foi criada com alguns dados. Para derrubar o container, aplica-se o comando:

**docker compose down**

Para demonstrar que foi destruído com sucesso, pode-se utilizar esse comando:

**docker ps**

```
letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/repositorio
ktypescript$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
--------------	-------	---------	---------	--------	-------	-------

```
letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/repositorio
ktypescript$
```

Na figura acima, nota-se que o container não está na lista, portanto, não está em execução.

## 2.2. SERVIÇO 2 – BACKEND EM TYPESCRIPT

A próxima etapa é modificar o arquivo docker-compose.yml para que seja criado também o serviço do backend. Pede-se, entre outras coisas, que o nome do serviço seja “backend-web”. A seguir, apresenta-se o arquivo com a adição do novo serviço.

```

17     backend-web:
18         build: .
19         container_name: backend-web
20         restart: always
21         depends_on:
22             - db
23         volumes:
24             - vol-pg-data:/var/lib/postgresql/data
25         environment:
26             - PORT=80
27             - HOST=0.0.0.0
28             - DBUSER=letsadmin
29             - DBPASSWORD=1234
30             - DBHOST=db
31             - DBPORT=5432
32             - DBNAME=db-aula06
33         ports:
34             - 80:80
35         networks:
36             - typescriptApp
37
38     volumes:
39         vol-pg-data:
40
41     networks:
42         typescriptApp:
43             driver: bridge

```

Explicando o conteúdo da imagem acima:

A propriedade build indica que deve ser utilizada um arquivo Dockerfile presente no diretório atual como base. Nota que não precisamos usar uma configuração command, por exemplo: command: sh -c "npm install && node dist/server.js", porque já está definido no arquivo Dockerfile. O backend-web é uma serviço que depende do banco de dados, então é possível especificar essa dependência. Desse modo, o docker compose vai executar primeiro o serviço “db”. As demais configurações são parecidas com a mostrada anteriormente.

Com os dois serviços definidos, podemos ativar os serviços através do seguinte comando:

**docker compose up -d**

```

letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/repositorioHard/
ktypescript$ docker compose up -d
[+] Building 4.2s (1/2)
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 174B
=> [internal] load .dockerignore
=> => transferring context: 59B
[+] Running 3/3
  :: Network q3_aula06_backtypescript_typescriptApp Created
  :: Container backend-web Started
  :: Container db Started
letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/repositorioHar
ktypescript$

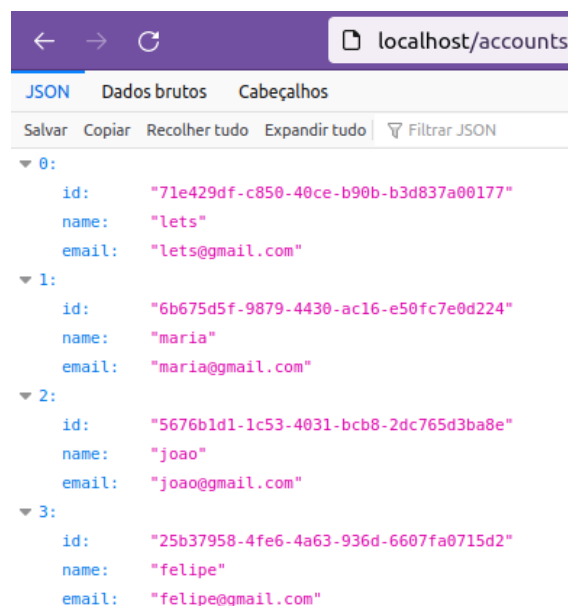
```

Para demonstrar que temos dois containers em execução, aplicou-se o comando:

## docker ps

```
letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/repositorioHard/Alpha-edtech-cycle01/Module15-Docker/Aula06
typescript$ docker ps
CONTAINER ID   IMAGE                                COMMAND                  CREATED        STATUS        PORTS
dd045587d111   postgres                            "docker-entrypoint.s..." 2 minutes ago  Up 2 minutes  0.0.0.0:5454->5432
993a98ed96c9   q3_aula06_backtypescript-backend-web "docker-entrypoint.s..." 2 minutes ago  Up 2 minutes  0.0.0.0:80->80/tcp
letonio@letonio-Inspiron-15-3567:~/Documentos/alphaedtech/hardSkills/repositorioHard/Alpha-edtech-cycle01/Module15-Docker/Aula06
```

Nota-se que os dois containers estão funcionando. Antes de empregar o Postman há uma rota que pode ser acessada via navegador (GET – <http://localhost:80/accounts>). A porta 80 é padrão do http, portanto, uma alternativa é escrever apenas <http://localhost/accounts> no navegador.



Percebe-se que está funcionando corretamente.

Vamos utilizar os comandos “docker inspect db” e “docker inspect backend-web” e para verificar detalhes dos containers

## docker inspect db

```
NetworkID: "9c577e4eb84cd1f0caafba028b33b1000554b53aa1a877916546e74c1d34fdb1",
"Networks": {
  "q3_aula06_backtypescript_typescriptApp": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": [
      "db",
      "db",
      "dd045587d111"
    ],
    "NetworkID": "9c577e4eb84cd1f0caafba028b33b1000554b53aa1a877916546e74c1d34fdb1",
    "EndpointID": "0d7b0796197206e61da37d4440c6aa75a758a134a5bd3ce62ce8647c02afea58",
    "Gateway": "172.27.0.1",
    "IPAddress": "172.27.0.2",
    "MacAddress": "02:00:14:00:00:00"
  }
}
```

## docker inspect backend-web

```

"Networks": {
  "q3_aula06_backtypescript_typescriptApp": {
    "IPAMConfig": null,
    "Links": null,
    "Aliases": [
      "backend-web",
      "backend-web",
      "993a98ed96c9"
    ],
    "NetworkID": "9c577e4eb84cd1f0caafba028b33b1000554b53aa1a877916546e74c1d34fdb1",
    "EndpointID": "71bdbe296975314dc20902c7eb6a339911ed1d3d316fbf1b049482033e669230",
    "Gateway": "172.27.0.1",
    "IPAddress": "172.27.0.3",
    "IPSubnet": "172.27.0.0/16"
  }
}

```

Nota-se que uma rede foi criada, cujo nome é formado pela combinação do nome do diretório da raiz do projeto combinado com typescriptApp informado no dockerfile-compose.yml. Nota-se que o gateway é o mesmo e cada container tem um IPAddress distinto.

## 2.3. TESTANDO A APLICAÇÃO

Utilizou-se o cliente Postman para verificar o funcionamento da aplicação.

**Caso 1: POST (<http://localhost:80/accounts/login>)** - Faz o login a partir de uma conta já registrada, pois algumas rotas exigem autenticação.

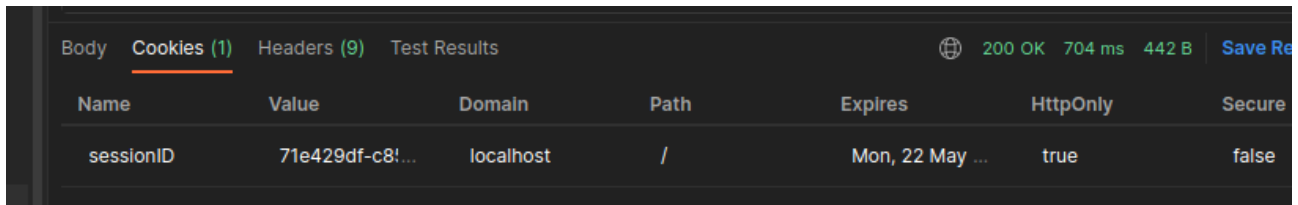
Body:

```

{
  "email": "lets@gmail.com",
  "password": "123"
}

```

Response:



Name	Value	Domain	Path	Expires	HttpOnly	Secure
sessionID	71e429df-c8!...	localhost	/	Mon, 22 May ...	true	false

**Caso 2: POST (<http://localhost:80/accounts>)** – Registrar uma nova container

Body:

```

{
  "name": "newUserCompose",
  "email": "newUserCompose@gmail.com",
  "password": "555"
}

```

Response:



```
1  {
2    "id": "e12518b9-09b8-4c98-a7b8-4784c593d895",
3    "name": "newUserCompose",
4    "email": "newUserCompose@gmail.com"
5  }
```

Para confirmar que foi adicionado com sucesso no banco de dados, vamos usar a rota GET (<http://localhost:80/accounts/>), onde é possível observar no final a conta recém-adicionada:

```
18  {
19    "id": "25b37958-4fe6-4a63-936d-6607fa0715d2",
20    "name": "felipe",
21    "email": "felipe@gmail.com"
22  },
23  {
24    "id": "e12518b9-09b8-4c98-a7b8-4784c593d895",
25    "name": "newUserCompose",
26    "email": "newUserCompose@gmail.com"
27  }
```

**Caso 3: PATCH (<http://localhost:80/accounts/>)** - Atualiza as informações de um usuário. Apenas o usuário que está logado pode fazer alterações na sua conta.

Body:

```
{
  "name": "letsUpdatedCompose",
  "email": "letsUpdatedCompose@gmail.com",
  "password": "111"
}
```

Response:

```
1  {
2    "id": "71e429df-c850-40ce-b90b-b3d837a00177",
3    "name": "letsUpdatedCompose",
4    "email": "letsUpdatedCompose@gmail.com"
5  }
```

Acessando a rota GET mais uma vez, pode-se notar que o usuário foi atualizado com sucesso:

```

18     "id": "e12518b9-09b8-4c98-a7b8-4784c593d895",
19     "name": "newUserCompose",
20     "email": "newUserCompose@gmail.com"
21   },
22   {
23     "id": "71e429df-c850-40ce-b90b-b3d837a00177",
24     "name": "letsUpdatedCompose",
25     "email": "letsUpdatedCompose@gmail.com"
26   }
27 ]

```

**Caso 4: DELETE (<http://localhost:80/accounts/:id>)** – Deletar a conta de um usuário baseado no ID.

Optou-se por fazer o delete do usuário que foi criado, cujo id é:

**e12518b9-09b8-4c98-a7b8-4784c593d895**

**DELETE**



<http://localhost:80/accounts/e12518b9-09b8-4c98-a7b8-4784c593d895>

Response:

Retorna os dados do usuário que foi deletado

```

1  [
2    {
3      "id": "e12518b9-09b8-4c98-a7b8-4784c593d895",
4      "name": "newUserCompose",
5      "email": "newUserCompose@gmail.com"
6    }
7  ]

```

Utilizou-se a rota GET para confirmar que o usuário não está presente mais.

```

13     "id": "25b37958-4fe6-4a63-936d-6607fa0715d2",
14     "name": "felipe",
15     "email": "felipe@gmail.com"
16   },
17   {
18     "id": "71e429df-c850-40ce-b90b-b3d837a00177",
19     "name": "letsUpdatedCompose",
20     "email": "letsUpdatedCompose@gmail.com"
21   }
22 ]

```

Por fim, utiliza-se a rota de logout para fazer o logout do usuário

**POST (http://localhost:80/accounts/logout)**

Response:

```
1  [24]  
2  "message": "[logout] Cookie removido e logout efetuado"  
3  [24]
```

Finalizando a verificação e comprovando que está funcionando normalmente.

### 3. QUESTÃO TEÓRICA

**Por fim, uma pergunta teórica. Se o seu docker-compose.yml usa a configuração “network”, apague-a porque deve continuar funcionando sem ela (se já não tinha isso, deixe como está). Também, se o seu dockercompose.yml publica a porta do banco, retire essa configuração (ou seja, não publique a porta) porque o backend deve continuar tendo acesso ao banco sem isso. A pergunta é: na aula passada quando rodamos o banco de dados sem publicar a porta dele, o backend mesmo assim conseguia acessar o banco porque os dois contêineres estavam na mesma rede (do tipo bridge, que criamos “na mão”), e estar na mesma rede faz com que um container tenha acesso ao outro. Em contraste, nesta aula nós não configuramos explicitamente rede nenhuma, e ainda assim o backend consegue acessar o banco (mesmo sem publicar a porta do banco). Por que o backend consegue acessar o banco ? (sugestão: a documentação sobre “network” do docker compose tem a resposta).**

Realmente, após fazer as modificações sugeridas sobre essa questão teórica, continuou funcionando normalmente. A explicação é que o próprio Docker Compose por padrão coloca os vários serviços que foram definidos numa mesma rede, criada implicitamente por ele. Mesmo sem configurar explicitamente, o Docker Compose cria uma rede interna e todos os serviços definidos no arquivo são conectados a essa mesma rede, e os containers podem usar o nome do serviço para se comunicar. Por isso que no serviço “backend-web”, a variável de ambiente foi definida como DBHOST=db, isto é, o host do banco de dados foi definido como o nome do serviço e funciona como hostname, permitindo a conexão.