Exercises 2.6 - 2.10

Volumes in action

Next we're going to set up the project management application Redmine, a PostgreSQL database and Adminer, a graphical interface for database administration.

All of the above have official Docker images available as we can see from Redmine, Postgres and Adminer respectively. The officiality of the containers is not that important, just that we can expect that it will have some support. We could also, for example, setup Wordpress or a MediaWiki inside containers in the same manner if you're interested in running existing applications inside Docker. You could even set up an application monitoring tool such as Sentry.

In https://hub.docker.com/_/redmine there is a list of different tagged versions:

Supported tags and respective `Dockerfile` links

- `5.1.2`, `5.1`, `5`, `latest`, `5.1.2-bookworm`, `5.1-bookworm`, `5-bookworm`, `bookworm`
- `5.1.2-alpine3.18`, `5.1-alpine3.18`, `5-alpine3.18`, `alpine3.18`, `5.1.2-alpine`, `5.1-alpine`, `5-alpine`, `alpine`
- `5.0.8`, `5.0`, `5.0.8-bookworm`, `5.0-bookworm`
- `5.0.8-alpine3.18`, `5.0-alpine3.18`, `5.0.8-alpine`, `5.0-alpine`

We can most likely use any of the available images.

From the section *Environment Variables* we can see that all versions can use `REDMINE_DB_POSTGRES` environment variable to set up a Postgres database. So before moving forward, let's setup Postgres for us.

In https://hub.docker.com/_/postgres there's a sample compose file under the section "via docker-compose or docker stack deploy". Let's strip that down as follows

```
version: "3.8"

services:
  db:
    image: postgres:13.2-alpine
    restart: unless-stopped
    environment:
      POSTGRES_PASSWORD: example
    container_name: db_redmine
```

Note:

- `restart: always` was changed to `unless-stopped`, that will keep the container running unless we explicitly stop it. With `always` the stopped container is started after reboot, for example, see here for more.

Under the section Where to store data, we can see that the `/var/lib/postgresql/data` should be mounted separately to preserve the data.

There are two options for doing the mounting. We could use a bind mount like previously and mount an easy-to-locate directory for storing the data. Let us now use the other option, a Docker managed volume.

Let's run the Docker Compose file without setting anything new:

```
$ docker compose up

  ✔ Network redmine_default  Created
  ✔ Container db_redmine     Created
  Attaching to db_redmine
  db_redmine  | The files belonging to this database system will be owned by user "postgres".
  db_redmine  | This user must also own the server process.
  ...
  db_redmine  | 2024-03-11 14:05:52.340 UTC [1] LOG:  starting PostgreSQL 13.2 on aarch64-unknown-linux
  db_redmine  | 2024-03-11 14:05:52.340 UTC [1] LOG:  listening on IPv4 address "0.0.0.0", port 5432
  db_redmine  | 2024-03-11 14:05:52.340 UTC [1] LOG:  listening on IPv6 address "::", port 5432
  db_redmine  | 2024-03-11 14:05:52.342 UTC [1] LOG:  listening on Unix socket "/var/run/postgresql/.s.
  db_redmine  | 2024-03-11 14:05:52.345 UTC [46] LOG:  database system was shut down at 2024-03-11 14:0
  db_redmine  | 2024-03-11 14:05:52.347 UTC [1] LOG:  database system is ready to accept connections
```

The image initializes the data files in the first start. Let's terminate the container with ^C. Compose uses the current directory as a prefix for container and volume names so that different projects don't clash (The prefix can be overridden with `COMPOSE_PROJECT_NAME` environment variable if needed).

Let's **inspect** if there was a volume created with `docker container inspect db_redmine | grep -A 5 Mounts`

```
"Mounts": [
    {
        "Type": "volume",
        "Name": "2d86a2480b60743147ce88e8e70b612d10b4c4151779b462baf4e81b84061ef5",
        "Source": "/var/lib/docker/volumes/2d86a2480b60743147ce88e8e70b612d10b4c4151779b462baf4e81b8406
        "Destination": "/var/lib/postgresql/data",
```

An indeed there is one! So despite us **not** configuring one explicitly, an anonymous volume was automatically created for us.

Now if we check out `docker volume ls` we can see that a volume with the name "2d86a2480b60743147ce88e8e70b612d10b4c4151779b462baf4e81b84061ef5" exists.

```
$ docker volume ls
  DRIVER              VOLUME NAME
  local     2d86a2480b60743147ce88e8e70b612d10b4c4151779b462baf4e81b84061ef5
```

There may be more volumes on your machine. If you want to get rid of them you can use `docker volume prune`. Let's put the whole "application" down now with `docker compose down`.

Instead of the randomly named volume we better define one explicitly. Let us change the definition as follows:

```
version: "3.8"

services:
  db:
    image: postgres:13.2-alpine
    restart: unless-stopped
    environment:
      POSTGRES_PASSWORD: example
    container_name: db_redmine
    volumes:
      - database:/var/lib/postgresql/data

volumes:
  database:
```

Now, after running `docker compose up` again, let us check what it looks like:

```
$ docker volume ls
  DRIVER              VOLUME NAME
  local               redmine_database

$ docker container inspect db_redmine | grep -A 5 Mounts
"Mounts": [
    {
        "Type": "volume",
        "Name": "redmine_database",
        "Source": "/var/lib/docker/volumes/ongoing_redminedata/_data",
        "Destination": "/var/lib/postgresql/data",
```

Ok, looks a bit more human-readable! Now when the Postgres is running, it is time to add Redmine.

The container seems to require just two environment variables.

```
redmine:
  image: redmine:5.1-alpine
  environment:
    - REDMINE_DB_POSTGRES=db
    - REDMINE_DB_PASSWORD=example
  ports:
    - 9999:3000
  depends_on:
    - db
```

Notice the depends_on declaration. This makes sure that the `db` service is started first. `depends_on` does not guarantee that the database is up, just that it is started first. The Postgres server is accessible with the DNS name "db" from the Redmine service as discussed in the section Docker networking.

Now when you run `docker compose up` you will see a bunch of database migrations running first.

```
redmine_1  | I, [2024-03-03T10:59:20.956936 #25]  INFO -- : Migrating to Setup (1)
redmine_1  | == 1 Setup: migrating ==========================================================
...
redmine_1  | [2024-03-03 11:01:10] INFO  ruby 3.2.3 (2024-01-30) [x86_64-linux]
redmine_1  | [2024-03-03 11:01:10] INFO  WEBrick::HTTPServer#start: pid=1 port=3000
```

As the documentation mentions, the image creates files to `/usr/src/redmine/files` and those are better to be persisted. The Dockerfile has this line where it declares that a volume should be created. Again Docker will create the volume, but it will be handled as an anonymous volume that is not managed by the Docker Compose, so it's better to create it explicitly.

With that in mind, our configuration changes to this:

```
version: "3.8"

services:
  db:
    image: postgres:13.2-alpine
    restart: unless-stopped
    environment:
      POSTGRES_PASSWORD: example
    container_name: db_redmine
    volumes:
      - database:/var/lib/postgresql/data
  redmine:
    image: redmine:4.1-alpine
    environment:
      - REDMINE_DB_POSTGRES=db
      - REDMINE_DB_PASSWORD=example
    ports:
      - 9999:3000
    volumes:
      - files:/usr/src/redmine/files
    depends_on:
      - db

volumes:
  database:
  files:
```

Now we can use the application with our browser through http://localhost:9999. After some changes inside the application, we can inspect the changes that happened in the image and check that no extra meaningful files got written to the container:

```
$ docker container diff $(docker compose ps -q redmine)
  C /usr/src/redmine/config/environment.rb
  ...
  C /usr/src/redmine/tmp/pdf
```

Probably not.

We could use command `psql` inside the Postgres container to interact with the database by running

```
docker container exec -it db_redmine psql -U postgres
```

The same method can be used to create backups with pg_dump

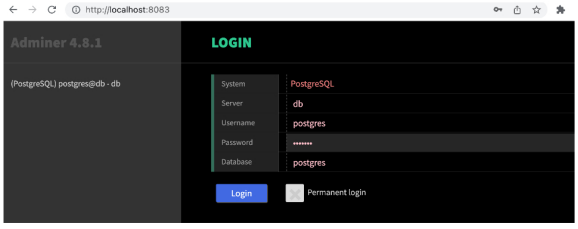
```
docker container exec db_redmine pg_dump -U postgres > redmine.dump
```

Rather than using the archaic command line interface to access Postgres, let us now set up the database Adminer to the application.

After a look at the documentation, the setup is straightforward:

```
adminer:
  image: adminer:4
  restart: always
  environment:
    - ADMINER_DESIGN=galkaev
  ports:
    - 8083:8080
```

Now when we run the application we can access the adminer from http://localhost:8083:



Setting up the adminer is straightforward since it will be able to access the database through the Docker network. You may wonder how the adminer finds the Postgres database container. We provide this information to Redmine using an environment variable:

```
  redmine:
    environment:
      - REDMINE_DB_POSTGRES=db
```

Adminer actually assumes that the database has DNS name *db* so with this name selection, we did not have to specify anything. If the database has some other name, we have to pass it to adminer using an environment variable:

```
  adminer:
    environment:
      - ADMINER_DEFAULT_SERVER=database_server
```

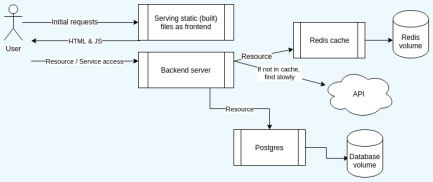
Exercises 2.6 - 2.10

> **EXERCISE 2.6**
>
> Let us continue with the example app that we worked with in Exercise 2.4.
>
> Now you should add a database to the example backend.
>
> Use a Postgres database to save messages. For now, there is no need to configure a volume since the official Postgres image sets a default volume for us. Use the Postgres image documentation to your advantage when configuring: https://hub.docker.com/_/postgres/. Especially part *Environment Variables* is a valuable one.
>
> The backend README should have all the information needed to connect.
>
> There is again a button (and a form!) in the frontend that you can use to ensure your configuration is done right.

Submit the docker-compose.yml

TIPS:

- When configuring the database, you might need to destroy the automatically created volumes. Use commands `docker volume prune`, `docker volume ls` and `docker volume rm` to remove unused volumes when testing. Make sure to remove containers that depend on them beforehand.
- `restart: unless-stopped` can help if the Postgres takes a while to get ready



EXERCISE 2.7

Postgres image uses a volume by default. Define manually a volume for the database in a convenient location such as in `./database` so you should use now a bind mount. The image documentation may help you with the task.

After you have configured the bind mount volume:

- Save a few messages through the frontend
- Run `docker compose down`
- Run `docker compose up` and see that the messages are available after refreshing browser
- Run `docker compose down` and delete the volume folder manually
- Run `docker compose up` and the data should be gone

> TIP: To save you the trouble of testing all of those steps, just look into the folder before trying the steps. If it's empty after `docker compose up` then something is wrong.

Submit the docker-compose.yml

The benefit of a bind mount is that since you know exactly where the data is in your file system, it is easy to create backups. If the Docker managed volumes are used, the location of the data in the file system can not be controlled and that makes backups a bit less trivial...

TIPS FOR MAKING SURE THE BACKEND CONNECTION WORKS

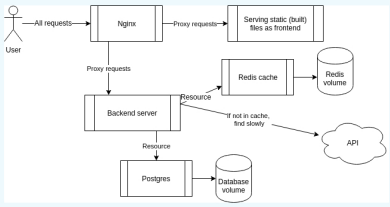
In the next exercise try using your browser to access http://localhost/api/ping and see if it answers pong

It might be Nginx configuration problem. Ensure there is a trailing / on the backend URL as specified under the location /api/ context in the nginx.conf.

EXERCISE 2.8

In this exercise, you shall add Nginx to work as a reverse proxy in front of the example app frontend and backend.

According to Wikipedia *a reverse proxy is a type of proxy server that retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client, appearing as if they originated from the reverse proxy server itself.*



So in our case, the reverse proxy will be the single point of entry to our application, and the final goal will be to set both the React frontend and the Express backend behind the reverse proxy.

The idea is that a browser makes *all* requests to *http://localhost*. If the request has a URL prefix *http://localhost/api*, Nginx should forward the request to the backend container. All the other requests are directed to the frontend container.

So, at the end, you should see that the frontend is accessible simply by going to http://localhost. All buttons, except the one labeled *Exercise 2.8* may have stopped working, do not worry about them, we shall fix that later.

The following file should be set to */etc/nginx/nginx.conf* inside the Nginx container. You can use a file bind mount where the contents of the file is the following:

```
events { worker_connections 1024; }

http {
  server {
    listen 80;

    location / {
      proxy_pass _frontend-connection-url_;
    }

    # configure here where requests to http://localhost/api/...
    # are forwarded
    location /api/ {
      proxy_set_header Host $host;
      proxy_pass _backend-connection-url_;
    }
  }
}
```

Nginx, backend and frontend should be connected in the same network. See the image above for how the services are connected. You find Nginx-documentation helpful, but remember, the configuration you need is pretty straightforward, if you end up doing complex things, you are most likely doing something wrong.

If and when your app "does not work", remember to have a look in the log, it can be pretty helpful in pinpointing errors:

```
2_7-proxy-1  | /docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
2_7-proxy-1  | /docker-entrypoint.sh: Configuration complete; ready for start up
2_7-proxy-1  | 2023/03/05 09:24:51 [emerg] 1#1: invalid URL prefix in /etc/nginx/nginx.conf:8
2_7-proxy-1 exited with code 1
```

Submit the docker-compose.yml

EXERCISE 2.9

Most of the buttons may have stopped working in the example application. Make sure that every button for exercises works.

Remember to take a peek into the browser's developer consoles again like we did back part 1, remember also this and this.

The buttons of the Nginx exercise and the first button behave differently but you want them to match.

If you had to make any changes explain what you did and where.

Submit the docker-compose.yml and both Dockerfiles.

PUBLISHING PORTS TO HOST NETWORK

There is an important lesson about Docker networking and ports to be learned in the next exercise.

When we do a port mapping, in `docker run -p 8001:80 ...` or in the Docker Compose file, we publish a container port to the host network to be accessible in localhost.

The container port is there within the Docker network accessible by the other containers that are in the same network even if we do not publish anything. So publishing the ports is only for exposing ports outside the Docker network. If no direct access outside the network is not needed, then we just do not publish anything.

EXERCISE 2.10

Now we have the reverse proxy up and running! All the communication to our app should be done through the reverse proxy and direct access (eg. accessing the backend with a GET to http://localhost:8080/ping) should be prevented.

Use a port scanner, eg https://hub.docker.com/r/networkstatic/nmap to ensure that there are no extra ports open in the host.

It might be enough to just run

```
$ docker run -it --rm --network host networkstatic/nmap localhost
```

If you have an M1/M2 Mac, you might need to build the image yourself.

The result looks like the following (I used a self-built image):

```
$ docker run -it --rm --network host nmap localhost
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-05 12:28 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000040s latency).
Other addresses for localhost (not scanned): ::1
Not shown: 996 closed tcp ports (reset)
PORT     STATE    SERVICE
80/tcp   filtered http
111/tcp  open     rpcbind
5000/tcp filtered commplex-link
8080/tcp filtered http-proxy

Nmap done: 1 IP address (1 host up) scanned in 1.28 seconds
```

As we see, there are two suspicious open ports: 5000 and 8080. So it is obvious that the frontend and backend are still directly accessible in the host network. This should be fixed!

You are done when the port scan report looks something like this:

```
Starting Nmap 7.93 ( https://nmap.org ) at 2023-03-05 12:39 UTC
Nmap scan report for localhost (127.0.0.1)
Host is up (0.0000040s latency).
Other addresses for localhost (not scanned): ::1
Not shown: 998 closed tcp ports (reset)
PORT    STATE    SERVICE
80/tcp  filtered http
111/tcp open     rpcbind

Nmap done: 1 IP address (1 host up) scanned in 1.28 seconds
```

Edit this page

Previous
« Docker networking

Next
Containers in development »