T7 - Web Development

T-WEB-700

Bootstrap

The Count of Money

1.1

For your web platform, you need to set up one widespread application architecture called **3-tier** :

1. a database, containing your data
2. a backend, serving the services you offer through an API
3. a frontend, to serve your static resources (HTML pages, Javascript scripts, CSS style sheet, fonts, images, …) and proxyfier your API

Developing such an application in a collaborative way requires rigor.
It could be helpful (/necessary) to virtualize and orchestrate each of the 3 services mentioned above.

DOCKER

You MUST at least be able to list, create, delete, stop and start containers, access their logs,…
You also need **docker-compose**, and the use of the following commands :

```
docker-compose ps|down|stop|start|restart|logs|rm
```

If you're not comfortable with this tool, ask your peers, Google or JARVISS.

1. Database

A database should ensure the persistence, integrity and protection of the data you handle.
Using the official image, create a container to be named *my_sql_database* (to be launched as a *daemon*), with your first service: the **MySQL database**.
Also define the corresponding **docker-compose** service in a *docker-compose.yml* file and run it.

Creation of the database service

In a *migration.sql* file, create the *my_database_name* database that will be used by default.
Respect the following tree structure:

Terminal

```
~/T-WEB-700> tree
.
+-- database
|    +-- migration.sql
+-- docker-compose.yml
1 directory, 2 files
```

Then create a *tasks* table (for instance with an id, a title, begin and end dates and a status), and insert some recordings. Here is an example :

Terminal

```
~/T-WEB-700> tail -n6 database/migration.sql
-- SEED THE DATABASE
INSERT INTO task
    (title, begin, end, status)
VALUES
    ('task 1', '2020-01-01 20:30:00', '2020-04-10 21:30:00', 'not started'),
    ('task 2', NOW(), '2020-04-07 23:42:00', 'in progress');
```

Modify your *docker-compose.yml* file so that this migration file is executed each time the container is started.

Environment variables are your friends, don't avoid them!

Log-files keep track of everything, they might be useful!

Data persistence

What happens to your data when you stop or delete your container ?

Using **docker-compose**:

- stop and suppress running containers,
- start the stack in daemon mode,
- insert some new records to your database service's container,
- stop the stack, then reboot it.

Check your data on your service's container.
Still using **docker-compose**, stop and suppress the stack, then reboot it.
Re-check your data on your service's container.
What are you witnessing ? Probably few things.
What did you expect ? Surely a lot more!

You are not the first and only one trying to run a database in a docker container. There must be a solution…

Complying with the tree below, create two scripts (*mysql_dump* and *mysql_restore*) to back up and restore your databases hosted in your container. Each script must take the name of the dump file as a parameter.

Terminal

```
~/T-WEB-700> tree
.
+-- database
|   +-- migration.sql
+-- docker-compose.yml
+-- scripts
|   +-- mysql-dump.sh
|   +-- mysql-restore.sh
2 directories, 4 files
```

Test your scripts by deleting and restoring your data, then check for its persistence.

ADMINISTRATION

Connecting to the mysql client of your container each time you need to introspect your database, is not very convenient…

Define a new *db_admin* service within your *docker-compose.yml* file.

This service must rely on the official *phpmyadmin* image and will depend on your *database* service.

Identify your administrative service with the user of your database, and connect the port 8090 of your host machine to port 80 of this container.
Test it: restart your service stack, and log into *localhost:8090* from your browser.

2. Backend - API with expressJS

The second part of the *3-tier architecture* is the backend, serving the services you offer through an API.

If you're not comfortable with those concepts, ask your peers, Google or JARVISS.

Defining routes

In your Dockerfile, define a *backend* service that will serve your API.
It will listen on a port configured by environment variable. Connect the 8080 host machine's port to your container's port, named *my_backend*.

```
services:
    ...
    backend:
        build:
            context: backend
            args:
                port: 1234
        container_name: my_backend
        ports:
            - 8080: 1234
        environment:
            PORT: 1234
```

Then create a directory *backend* in which you put the necessary files at your service.

Terminal

```
~/T-WEB-700> tree
.
+-- backend
|   +-- app.js
|   +-- Dockerfile
+-- database
|   +-- migration.sql
+-- docker-compose.yml
+-- scripts
|   +-- mysql-dump.sh
|   +-- mysql-restore.sh
3 directories, 6 files
```

Your Dockerfile should just install necessary libraries in a nodeJS environment *(we use LTS)* and start the server on the specified port.

Right now, your API only provides the following *CRUD routes* concerning the tasks.

- GET /api/tasks
- POST /api/tasks
- GET /api/tasks/:id
- PUT /api/tasks/:id
- DELETE /api/tasks/:id

Handle these routes in an app.js file.

For the moment, it is not necessary to implement these routes. We will just return a JSON message.

Terminal

```
~/T-WEB-700> tail -n5 backend/app.js
app.use('/api/tasks', taskRouter);
var port = process.env.PORT || 8080;
app.listen(port, function() {
    console.log('Example app listening on port', port);
});
```

Rebuild and restart your application stack.
Keep track of your logs.

Test your routes with **curl** the following way.
Are they built correctly and accessible?

```
curl -H "Content-Type: application/json" -X GET http://localhost:8080/api/tasks
curl -d '{"key1":"value1", "key2":"value2"}' -H "Content-Type: application/json"
                                -X POST http://localhost:8080/api/tasks
curl -H "Content-Type: application/json" -X GET http://localhost:8080/api/tasks/1
curl -d '{"key1":"value1-bis", "key2":"value2-bis"}'
                                -H "Content-Type: application/json"
                                -X PUT http://localhost:8080/api/tasks/1
curl -H "Content-Type: application/json" -X DELETE http://localhost:8080/api/tasks/1
```

COMMUNICATION

Create a connection to the database from your backend, and implement your routes to establish communication between your API and your database.

docker-composer starts parallel containers by default.
Pay attention to services' starting order, and be aware that you can't access a booting database.

TESTS

It is essential to test the API (possibly via *phpmyadmin*): how well do your actions work? does the API provide expected responses?

Using **curl**, test the following features:

- list the tasks
- create a task (success and error)
- recover a task (success and error)
- modify a task (success and error)
- delete a task

These tests aren't enough! The more you implement, the more robust your API will be.

3. Frontend - ReactJS

Almost there!
You have to build a nice, functional front-end with an astonishing interface providing the most advanced user experience (without which your application would be unusable).

Basic application

You should always make your work easier by using tools that automates the recurring development tasks. We will use here create-react-app.

Read the doc to understand what this tool can provide you.

Make sure that *create-react-app* is installed globally on your machine.

Using *create-react-app*, create a *ReactJS* application, named ***frontend***, at the root of your project.
Then code this application to allow listing, creating, deleting and modifying the tasks.

Feel free to use external librairies, but remember they need to be installed automatically via ***npm***.

API is served from the same domain's name as your front.

You do not need to actually connect to the API in order to develop the frontend.
You could mock the API during your development phase.

Mutli-stage building

Now, you have to ***'dockerize'*** your front end.

Notice the container needs an entire NodeJS ecosystem to build your web application (.js, .css, index.html, assets, etc.) and a web server to serve them.

It is wise not to put all your eggs in one basket!

Docker provides ***build multi-stage***, a mechanism to handle your type of situation. Implement it, using (in the same ***Dockerfile***) :

- a *node container*, used as a development context to build your ReactJS application
- a *nginx container*, used as a production service.

Official nginx docker image and its documentation might be convenient.

Reverse proxy

You beautiful frontend is now accessible from any browser.
Nevertheless, you probably already noticed that all your API calls are failing.
Why?

Do all roads lead to Rome?

Ending

Connect the rights ports together and make sure nothing and no one is listening.

```
sudo netstat -tunelp | grep 80
```

Stop the service that listens on this port (for example apache server).

```
sudo systemctl stop apache2.service
```

Add the following line to your */etc/hosts* file:

```
127.0.0.1       www.my-fake-domain.com
```

Relaunch your application stack and visit *www.my-fake-domain.com* on any browser.

CONGRATULATIONS, you learned basic recipees to cook your own web application.
We might say you are almost a full-stack developer!