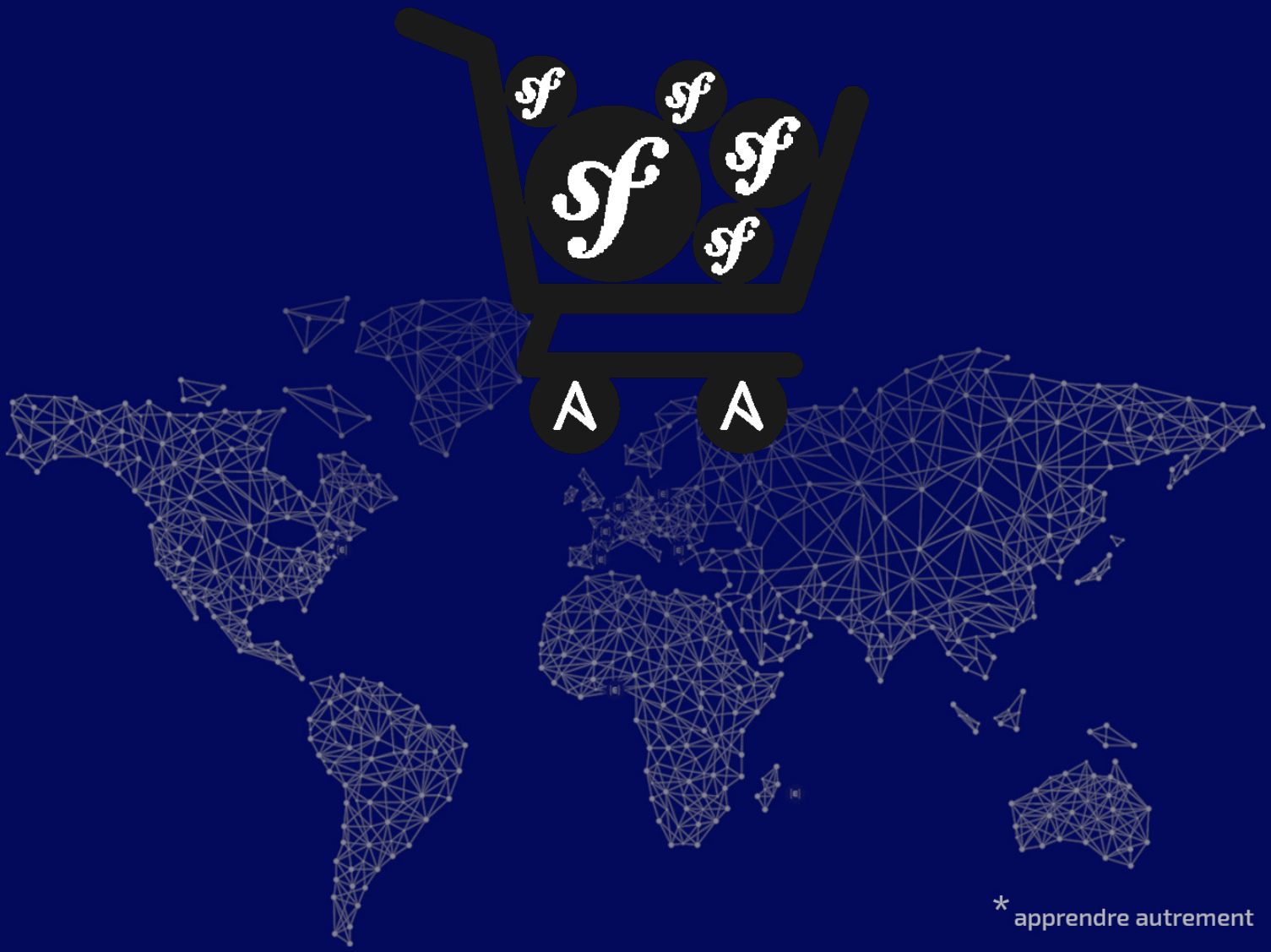
{EPITECH}
LEARN DIFFERENT*

E-COMMERCE

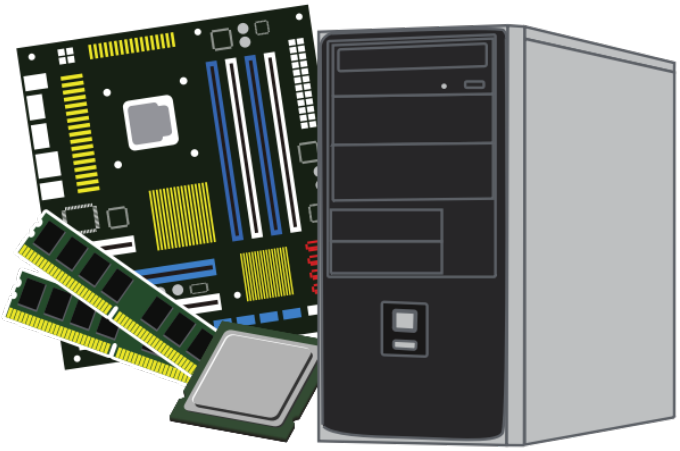
BOYS' FINNY MEALS



* apprendre autrement

E-COMMERCE

Your objective is to create a generic API for e-commerce merchant sites with a maximum of features. In our example, we will sell computer components.



- ✓ **Back-end**
  You **must** use the (latest) Symfony framework to create this API.
- ✓ **Front-end**
  You **must** use React library with TypeScript.
- ✓ **Deployment**
  You **must** use Docker to deploy your project.

Your project, and especially the back-end, are located in an `app` folder at the root of your repository. Thus, we should be able to launch it using:

```
Terminal
~/T-WEB-600> cd app/ && symfony server:start
```

Back-end, front-end and deployment must be dockerised and be ready to deploy!

API

Your API must respect the **REST** standard and using JSON format to represent data.

Tools as api-platform that generate path, URL or http verb are prohibited.

Each endpoint must use the **most** appropriate HTTP verb (GET, POST, PUT or DELETE).
The HTTP code associated with the response from the server must be consistent with the response sent (200 when all is well, 404 when a resource cannot be found, ...) and as precise as possible (201 when a resource is created for instance).

Your API must be implemented using the power of the Symfony framework and use the JSON format for data exchange (in requests and responses bodies).

Below you'll find all the endpoints that we ask for and their uses in the following format "- what the endpoint is used for (`/api/endpoint/{variablePart}`)", where:

- ✓ `{variablePart}` is to be replaced by a variable that'll be used in a request (such as an id) ;
- ✓ `/api/endpoint/{variablePart}` is the URI to access the endpoint on the host.

If the host is `localhost` we must be able to access it using `http://localhost/api/endpoint/{variablePart}` (with the required HTTP verb).

Start by making a simple but functional API before adding more advanced features!

In case of errors, the body must be as follows:

```
{
    "error": "The error message explaining what went wrong."
}
```

Users

A *user* is represented as such:

```
{
    "login": "foobar",
    "password": "mypassword",
    "email": "my@email.com",
    "firstname": "Foo",
    "lastname": "Bar"
}
```

- ✓ registration of user (`/api/register`)
- ✓ connection of user, retrieving the authentication token (`/api/login`)

The login process, in case of success must return:

```
{
    "token": "XXXXXXXXXX"
}
```

This token will then be needed in the `authorization` header of the requests that needs authentication (indicated by the **AUTHED** flag):

- ✓ update current user information (`/api/users`) – **AUTHED**
- ✓ display current user information (`/api/users`) – **AUTHED**

When displaying the information, is it secure to send the password?

Catalog

A *product* is represented as such:

```
{
    "id": 1,
    "name": "Item 3000",
    "description": "Best item in the shop!",
    "photo": "https://path/to/image.png",
    "price": 13.37
}
```

- ✓ Retrieve list of products (`/api/products`)
- ✓ Retrieve information on a specific product (`/api/products/{productId}`)
- ✓ Add a product (`api/products`) – **AUTHED**

- ✓ Modify and delete a product (`/api/products/{productId}`) – **AUTHED**
- ✓ Add a product to the shopping cart. (`/api/carts/{productId}`) – **AUTHED**
- ✓ Remove a product to the shopping cart. (`/api/carts/{productId}`) – **AUTHED**
- ✓ State of the shopping cart (list of products in the cart). (`/api/carts`) – **AUTHED**
- ✓ Validation of the cart (aka converting the cart to an order) (`/api/carts/validate`) – **AUTHED**

Permissions are very important, not everyone should be able to add and remove items.

Orders

An **order** is represented as such:

```
{
    "id": 1,
    "totalPrice": 42.01,
    "creationDate": "2021-04-01 08:32:00Z",
    "products": [
        {
            "id": 1,
            "name": "Item 3000",
            "description": "Best item in the shop!",
            "photo": "https://path/to/image.png",
            "price": 13.37
        },
        {
            "id": 2,
            "name": "Another item",
            "description": "Still good",
            "photo": "https://path/to/image2.png",
            "price": 28.64
        }
    ]
}
```

- ✓ recover all orders of the current user (`/api/orders/`) – **AUTHED**
- ✓ Get information about a specific order (`/api/orders/{orderId}`) – **AUTHED**
- ✓ Is only authorized if the order belong to the logged user.

Storage

To store the data you need to install a database.
In this project, we will use Postgresql.

The following environment variable **must** be used to connect the database:
`DATABASE_URL` (eg: 'postgresql://db_user:db_password@127.0.0.1:5432/db_name')

Frontend

For the frontend part, you must use React with Typescript.

You need to implement the frontend parts that correspond to the backend API routes

Payment

An e-commerce without a payment it's just a website.

As a final part, you will have to implement stripe. It's important to understand all the process before implementing the payment method. Get it right, it can be damaging for your company if payment are misconfigured.

Development environment

Independently of the deployment part (see below), we **must** be able to launch your Symfony application (aka the API) using PHP's built-in web server with this command (only for development environments):

```
Terminal
~/T-WEB-600> symfony server:start
```

Deployment

You **must** be able to deploy your API using the latest version of Docker.

You will have 3 services: PHP, web server and database.

Web server as apache2 or Nginx.

The container of the web server and the php server must be separated

Have you heard about php-fpm ?

To orchestrate the containers, you must use docker compose

Docker compose is now included natively in docker

You must have two files at the root of your repository:

- ✓ the `Dockerfile` for build your custom image of PHP server.
- ✓ the `compose.yml` to run all your services.

Organization

It is without saying that we cannot really test your deployment if there's no application to deploy. But the deployment is an important part of the project and should not be considered at the end. Do the whole project gradually, by step (and don't forget that you're in group, so you can work in parallel on several things).

Here's to help you, the beginning of an organization scheme:

- ✓ Work on parallel on the API and on the deployment.
- ✓ When you have a basic API (for instance when the retrieval, creation, deletion and updating of products is done), focus your effort on the deployment part.
- ✓ When the deployment part is done, then focus on adding the rest of your API.

Focussing on something don't necessary mean to dedicate all your resources to it. It depends o your particular team organizations and the skills of each one of you.

Focussing on something don't necessary mean to dedicate all your ressources to it. It depends of your particular team organizations and the skills of each one of you.

* apprendre autrement