{EPITECH}

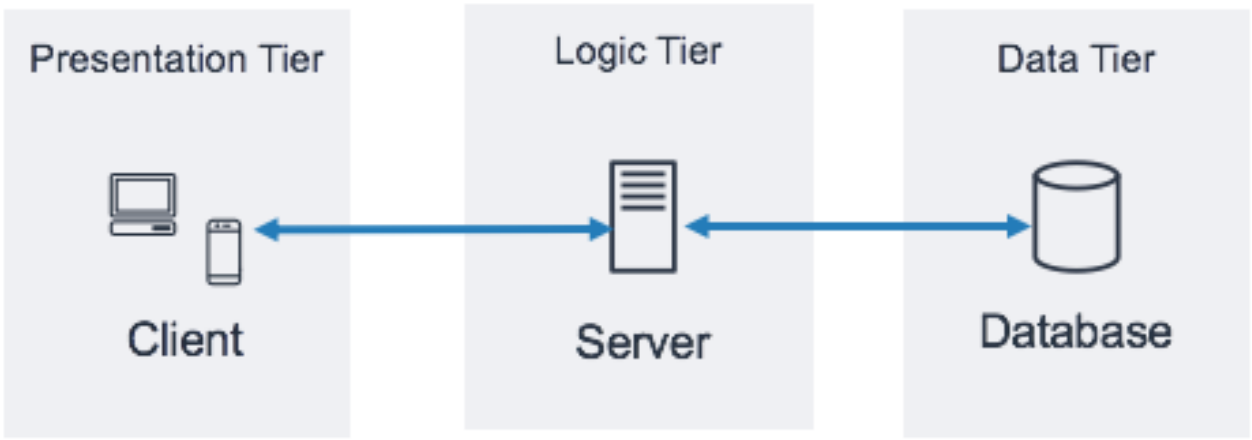
TRINITY - DEV WEB

BOOTSTRAP

TRINITY - DEV WEB

To set up your web platform, a common architecture, called 3-tier architecture, is needed:

- ✓ the **data** tier:
  a back-end database contains the data and its management system ;
- ✓ the **logic** tier:
  a middle server contains the business logic and provides some services via an API ;
- ✓ the **presentation** tier:
  a front-end client serving static content (HTML pages, JS scripts, CSS style sheets,…).



You should search for the benefits and disadvantages of this type of architecture.

{EPITECH}

API

REST principles

Your API must respect the **RE**presentational **S**tate **T**ransfer principles in order to ensure efficient and scalable communication between the backend and the clients (web or mobile).

Here are the key principles you should follow:

✓ Stateless:
Each request contains all the necessary information to be understood by the server.
The server should not maintain state between two requests.

> For instance: each POST request to create a product must contain all the necessary information (name, price, etc.), without the need to store data on the client session on the server side.

✓ Client-Server:
There is a strict separation between the client (user interface) and the server (backend).

> For instance: the client simply queries the API via `/api/products` to obtain the list of products, without having the code embedded in their project. We can often encounter this problem with native PHP projects where we find database management directly in the display rendering files.

✓ Uniform Interface:
The API has a uniform interface allowing resources to be manipulated consistently.
HTTP methods (such as GET, POST, PUT, and DELETE) are used to manipulate resources.

> For instance:
>
> - `GET /api/products` collects all products,
> - `POST /api/products` creates a new product,
> - `PUT /api/products/:id`: updates a specific product,
> - `DELETE /api/products/:id`: deletes a product.

Resource Management with an ORM

Use an ORM like sequelize in order to simplify interactions with the database.
ORM allows you to manipulate data as objects, rather than using direct SQL queries.

The main principles of **O**bject **R**elational **M**apping are:

- ✓ **Database abstraction**:
  ORM allows you to work with JS objects (or other languages) instead of directly manipulating SQL tables and columns.

  > For instance: You define a `Product` model in Sequelize, and every interaction with the database can be done with methods like `Product.create()` or `Product.findAll()`.

- ✓ **Models and relations**:
  Entities (such as products, users, orders, sales, …) are represented as models and the relationships between these models (such as this product belongs to that order) can be managed directly by the ORM.

  > Have a look a the model basics.

- ✓ **Automatic migration**:
  Migrations allow you to modify and maintain the database structure in a secure and versioned manner. The ORM automatically generates the necessary migrations to create or modify the tables.

  > Check how to synchronize your models.

Securing the API with JSON Web Tokens

Some routes are sensitive, like the ones used for adding products, or managing sales.
You MUST secure them! To do so, use use JWT for user authentication and authorization.

The token is generated upon login and sent with each request requiring authentication.
A refresh token is necessary to set up on your API.

API documentation with Swagger

A good API should always been well documented, in order to allow developers to use it correctly.
Swagger is a tool that automatically generates interactive documentation.

Setting up a Swagger has a double challenge:

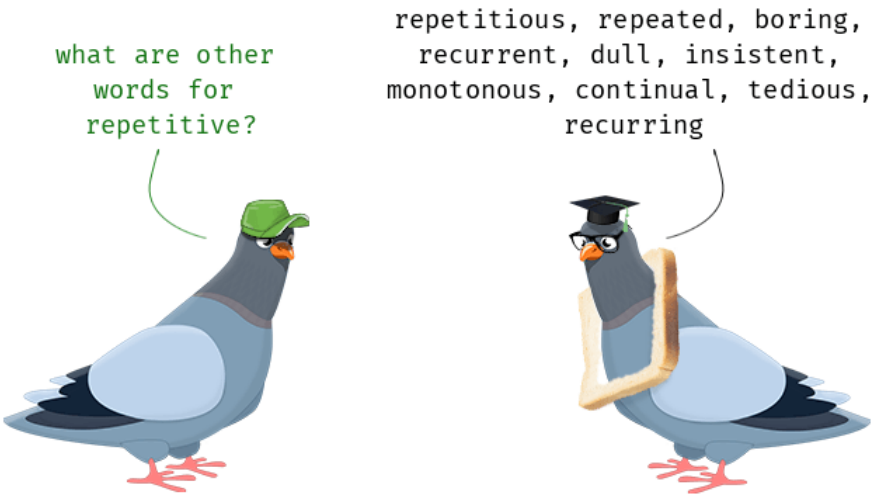
- ✓ **API self-description**:
  Routes, parameters and response types are described in Swagger, and documentation is accessible online via an interactive web interface.

- ✓ **Endpoint testing**:
  Swagger allows developers to directly test endpoints via the interface without using external tools like Postman.

Front

You now need to build a nice and functional frontend, with an impressive interface providing the best possible user experience (without which your application would be unusable).
For this bootstrap, you will use *create-react-app*.

You should always make your work easier by using tools that automate repetitive tasks.



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 editing products.

Read the documentation to understand what a tool can do.
Make sure *create-react-app* is installed globally on your machine.

Feel free to use external libraries, but remember that they should be installed automatically via a package manager, such as `npm`.

The API must be served from the same domain as your frontend. Actually, you don't need to connect to the API to develop the frontend. Just mock the API during the development phase.

In order to connect your reactJS front to your API, you must use a solution allowing you to make requests, the most used library to allow this is the axios library.

v 0.1