{EPITECH}
LEARN DIFFERENT*

EPYTODO

BUILD AN API TO CREATE A TODO APP

EPYTODO

language: Node.js

The project idea is to build a Todo List. Thanks to it, you'll be able to handle all the tasks you need to do easily! We will mainly focus on the "backend" side of the project, but feel free to show us what you can by building a "frontend" to your project as a bonus.

Within this project, you'll have to develop:

1. Your MySQL database.
2. A web server using Node.js

The bootstrap will help you a lot!

MySQL database

Into your database, you'll have to manage various users and their respective tasks (todos).

Pay attention to the instructions below

Create a file named `epytodo.sql` at the root of your repository.

You have to write into it your whole database scheme. You can also use external tools to build your Database and export it as a `.sql` file later.

The database's name **must** be `epytodo`.
It **must** contain 2 tables:

1. user
2. todo

Here are the description of the fields each table must contain:

1. user table
    - ✓ id (mandatory, not null, auto-increments)
    - ✓ email (mandatory, not null, unique)
    - ✓ password (mandatory not null)
    - ✓ name (mandatory not null)
    - ✓ firstname (mandatory not null)
    - ✓ created_at (set to the current datetime by default)
2. todo table
    - ✓ id (mandatory not null, auto-increments)
    - ✓ title (mandatory not null)
    - ✓ description (mandatory not null)
    - ✓ created_at (set to the current datetime by default)
    - ✓ due_time (mandatory, not null, datetime)
    - ✓ status (**not started** by default / **todo** / **in progress** / **done**)
    - ✓ user_id (mandatory, reference to the id of the user that get assigned to the task)

Think about the last one: why do you need this?
Maybe it has to do with relationships, foreign keys…

Choose the type of each field carefully, depending on what it's going to be used for.

Once your scheme is created, import your file into your MySQL server.

```
Terminal
B-WEB-200> cat epytodo.sql | mysql -u root -p
```

Your SQL file has to be placed in the root folder when turned in.
Do not insert any records into this file.

Web server

Unlike other programming languages, Node and express let you build things your own way, meaning that you can do the same thing in tons of different ways.
This also means that your file structure can get quite messy, so try to keep your code and your architecture as clean as possible if you don't want to get lost.

Here's how your repository's structure should look like (you can adapt the structure to your needs) :

```
|-- .env
|-- package.json
`-- src
    |-- config
    |   `-- db.js
    |-- index.js
    |-- middleware
    |   |-- auth.js
    |   `-- notFound.js
    `-- routes
        |-- auth
        |   `-- auth.js
        |-- todos
        |   |-- todos.js
        |   `-- todos.query.js
        `-- user
            |-- user.js
            `-- user.query.js
```

More explanations about what each folder/file is for :

- ✓ `src`: your main folder.
- ✓ `package.json`: your app package file, it must be filled with all the necessary informations, dependencies and have a `start` script.
- ✓ `config`: contains the files that deal with the connection to the database.
- ✓ `index.js`: the main file, the one that starts everything (it calls and runs the `app`).
- ✓ `middleware`: contains all the middlewares created
- ✓ `routes`: contains all the subfolders that contain the routes needed for the project.

You should have a node_modules folder at the root of your project. If you look closely, it is not listed on the tree above, that's because we **DO NOT PUSH** this folder. Check out gitignore

Environment variables

The `.env` file must contain all the configuration variables that will be necessary for the Project.

Here are the required ones :

- ✓ MYSQL_DATABASE
- ✓ MYSQL_HOST
- ✓ MYSQL_USER
- ✓ MYSQL_ROOT_PASSWORD
- ✓ PORT // used for the Express server.
- ✓ SECRET // used for the JSON Web Token (JWT).

API

It's not the most fun part, but take some time to read about API.

Here, we're using a REST API to create our CRUD system.

All data will transit into JSON format.

During your research, you will face a lot of packages and entities that will help you build your app. Some of them are good and some are evil. For this project, you will only be authorized to use the packages listed below. Every other package will be forbidden or needs to be authorized by the pedagogical team.

- ✓ Express
- ✓ mysql2
- ✓ dotenv
- ✓ jsonwebtoken
- ✓ bcryptjs
- ✓ body-parser This one is optional, find out why.

Routes

Here is a listing of all the routes we expect for this project.

route	method	protected	description
/register	POST	no	register a new user
/login	POST	no	connect a user
/user	GET	yes	view all user information
/user/todos	GET	yes	view all user tasks
/users/:id or :email	GET	yes	view user information
/users/:id	PUT	yes	update user information
/users/:id	DELETE	yes	delete user
/todos	GET	yes	view all the todos
/todos/:id	GET	yes	view the todo
/todos	POST	yes	create a todo
/todos/:id	PUT	yes	update a todo
/todos/:id	DELETE	yes	delete a todo

In this project, we expect you to protect your route and only make them accessible to logged-in users.
To do so, the protected routes should receive a valid JWT (JSON Web Token) in the Authorization header (See jsonwebtoken npm package)

Definitions of format

Every route can have multiple methods, parameters or responses, everything you need will be explained in this documentation.

Defaults

By default, each error will be treated as follows.
If there are more details, that will be explained in the related part.
For occurring common errors we suggest you build a custom error handler middleware and simply send the error instead of dealing with it in every route.

1. If the user is not logged in :

Terminal

```
{ "msg": "No token, authorization denied" }
```

1. If the user sends an invalid Token :

Terminal

```
{ "msg": "Token is not valid" }
```

2. If the task or user does not exist :

Terminal

```
{ "msg": "Not found" }
```

3. If user give bad parameters :

Terminal

```
{ "msg": "Bad parameter" }
```

4. If there's another error :

Terminal

```
{ "msg": "Internal server error" }
```

Register

POST `/register`

Request body:

Terminal

```
{
  "email": "nao.marvin@epitech.eu",
  "name": "Marvin",
  "firstname": "Nao",
  "password": "3paulbec"
}
```

Response:

Terminal

```
{ "token": "Token of the newly registered user" }
```

Response, if the account already exists:

Terminal

```
{ "msg": "Account already exists" }
```

The password should not be stored as plain text! You have to hash it before inserting it into your database. Check the packages listed above. Do not hesitate to do some research on your own to find out the best way to do it.

POST `/login`

Request body:

Terminal

```
{
    "email": "username",
    "password": "password"
}
```

Response body:

Terminal

```
{ "token": "Token of the newly logged in user" }
```

Response body, if the credentials are incorrect:

Terminal

```
{ "msg": "Invalid Credentials" }
```

users

GET `/user`

Response body:

Terminal

```
{
    "id": 1,
    "email": "email@test.eu",
    "password": "hashed password",
    "created_at": "2021-03-03 19:24:00",
    "firstname": "test",
    "name": "test"
}
```

GET `/user/todos`

Response body:

Terminal

```
[
    {
        "id": 1,
        "title": "title",
        "description": "desc",
        "created_at": "2021-03-03 19:24:00",
        "due_time": "2021-03-04 19:24:00",
        "user_id": 3,
        "status": "done"
    },
    {
        "id": 2,
        "title": "title",
        "description": "desc",
        "created_at": "2021-03-05 19:24:00",
        "due_time": "2021-03-06 19:24:00",
        "user_id": 3,
        "status": "in progress"
    }
]
```

GET `/users/:id` **and** `/users/:email`

Response body:

Terminal

```
{
    "id": 1,
    "email": "email@test.eu",
    "password": "hashed password",
    "created_at": "2021-03-03 19:24:00",
    "firstname": "test",
    "name": "test"
}
```

PUT `/users/:id`

Request body:

Terminal

```
{
    "email": "updated_email@test.eu",
    "password": "updated_password",
    "firstname": "updated_test",
    "name": "updated_test"
}
```

Response body:

Terminal

```
{
    "id": 1,
    "email": "updatedemail@test.eu",
    "password": "hashed password",
    "created_at": "2021-03-03 19:24:00",
    "firstname": "test",
    "name": "test"
}
```

DELETE `/users/:id`

Response body:

Terminal

```
{ "msg": "Successfully deleted record number: ${id}" }
```

todos

GET `/todos`

Response body:

Terminal

```
[
    {
        "id": 1,
        "title": "title",
        "description": "desc",
        "created_at": "2021-03-03 19:24:00",
        "due_time": "2021-03-04 19:24:00",
        "user_id": 1,
        "status": "done"
    },
    {
        "id": 2,
        "title": "title",
        "description": "desc",
        "created_at": "2021-03-05 19:24:00",
        "due_time": "2021-03-06 19:24:00",
        "user_id": 2,
        "status": "in progress"
    }
]
```

GET `/todos/:id`

Response body:

Terminal

```
{
    "id": 2,
    "title": "title",
    "description": "desc",
    "created_at": "2021-03-05 19:24:00",
    "due_time": "2021-03-06 19:24:00",
    "user_id": 3,
    "status": "in progress"
}
```

POST /todos

Request body:

Terminal

```
{
    "title": "title",
    "description": "desc",
    "due_time": "2021-03-06 19:24:00",
    "user_id": 3,
    "status": "todo"
}
```

Response body: the created todo (just like the GET todo route).

PUT `/todos/:id`

Request body:

Terminal

```
{
    "title": "Updated title",
    "description": "Updated desc",
    "due_time": "2021-03-07 19:24:00",
    "user_id": 1,
    "status": "in progress"
}
```

Response body: the updated todo (just like the GET todo route).

DELETE `/todos/:id`

Terminal

```
{ "msg": "Successfully deleted record number: ${id}" }
```

Each response must use an appropriate HTTP status.

v 3.2