Practical Assignment: Development of a RESTful API with MVC Pattern

Introduction

In this project, you will develop a **RESTful API** using **JavaScript** as the programming language in the backend environment. The API will follow the **MVC (Model-View-Controller)** design pattern, with the particularity that the "View" will be adapted to a backend context, interpreted as the presentation of data to the client through **HTTP responses** (e.g., in JSON format) instead of a graphical interface. This practical assignment aims to introduce you to the fundamentals of backend development, enabling you to design and build a functional API that serves as a bridge between the frontend and a database.

Since APIs are an essential component in backend development, this project will help you understand how to facilitate efficient and secure communication between different parts of an application. As this is your first personal backend project, this assignment includes detailed instructions and considerations to guide you step by step.

Objectives

- Design and implement a RESTful API that supports **CRUD operations** (Create, Read, Update, Delete).
- Apply the **MVC** pattern adapted to a backend environment.
- Write code in **JavaScript** following **programming best practices**.
- Perform **testing** to ensure the functionality and quality of the API.
- Document the API clearly and comprehensively.

Technical Requirements

- **Programming Language:** JavaScript, using **Node.js** as the runtime environment.
- **Framework:** You can use **Express.js** (recommended for its simplicity) or another framework of your choice to manage routes and middleware.
- **Database:** Choose a database you are comfortable with (e.g., **MongoDB**, **MySQL**, **PostgreSQL**, or even an in-memory database like SQLite). Include the necessary configuration to connect the API to the database.
- **Testing:** Implement tests (unit and/or integration) using a framework such as **Jest**, **Mocha**, or similar.

- **Documentation:** Document the API endpoints using tools like **Swagger**, **Postman**, or a **README.md** file.

Project Details

1. Model

- Define at least **one data model** that represents an entity in your application. For example:
  - Users (with fields like `id`, `name`, `email`).
  - Products (with fields like `id`, `name`, `price`).
  - Posts (with fields like `id`, `title`, `content`).
- Implement the logic to interact with the database, supporting CRUD operations:
  - **Create:** Insert a new resource.
  - **Read:** Retrieve one or multiple resources.
  - **Update:** Update an existing resource.
  - **Delete:** Delete a resource.

2. Controller

- Create **controllers** that process HTTP requests and use the models to perform CRUD operations.
- Ensure you:
  - Handle errors (e.g., return a 404 code if a resource does not exist).
  - Send clear and structured HTTP responses (e.g., in JSON format).

3. Routes

- Define **routes** for each CRUD operation using appropriate HTTP verbs:
  - `GET /resource` - Retrieve all resources.
  - `GET /resource/:id` - Retrieve a specific resource.
  - `POST /resource` - Create a new resource.
  - `PUT /resource/:id` or `PATCH /resource/:id` - Update an existing resource.
  - `DELETE /resource/:id` - Delete a resource.
- Use descriptive and consistent names for routes.

4. Testing

- Write **tests** for each endpoint, verifying:
  - Successful cases (e.g., creating a resource and receiving a 201 code).
  - Error cases (e.g., attempting to update a non-existent resource and receiving a 404 code).
- Use a testing framework to automate tests and generate reports.

5. Documentation

- Provide a clear description of each endpoint, including:
  - **URL** (e.g., `/api/products`).
  - **HTTP Method** (GET, POST, etc.).
  - **Required Parameters** (in the URL, body, or query).
  - **Examples** of requests and responses.
- Example:

```
GET /api/products
Description: Retrieves the list of products.
Response: 200 OK
[
  { "id": 1, "name": "Laptop", "price": 1200 },
  { "id": 2, "name": "Mouse", "price": 25 }
]
```

Additional Considerations

- **Best Practices:**
  - Write modular code (separate models, controllers, and routes into different files).
  - Use consistent naming for variables and functions (e.g., camelCase).
  - Define constants for fixed values (such as HTTP status codes or error messages).
- **Basic Security:** Validate user inputs to prevent errors or attacks (e.g., ensure required fields are present).
- **Scalability:** Although this is your first project, consider how you could expand the API in the future (e.g., adding JWT authentication as an optional challenge).
- **Debugging:** Use tools like `console.log` or a debugger to identify issues during development.

Deliverables

1. **Source Code:** All API files, organized in a clear structure.
2. **Documentation:** A file (e.g., `README.md`) with instructions and endpoint details.
3. **Execution Instructions:** Explain how to install dependencies (`npm install`), configure the database, and run the API (`npm start`).
4. **Test Report:** Evidence of the tests performed and their results.

Evaluation Criteria

- **Functionality:** The API must correctly perform CRUD operations.
- **Code Quality:** The code must be clean, organized, and follow best practices.
- **Testing:** Tests must cover the main cases and detect errors.
- **Documentation:** It must be clear and sufficient for another developer to use the API without issues.

Recommended Resources

- Official documentation for **Node.js** and **Express.js**.
- Tutorials on RESTful APIs and the MVC pattern.
- Testing guides for **Jest** or **Mocha**.
- Documentation examples in **Swagger** or **Postman**.

Final Tips

As this is your first backend project, we recommend:

- **Plan:** Break the work into stages (setup, models, routes, testing, documentation).
- **Test Often:** Run your API and perform manual tests during development.
- **Seek Help:** If you encounter difficulties, consult official documentation or online tutorials.