**Node.js:**

**Problem 1: Basic Node.js Server**

**Problem Statement**

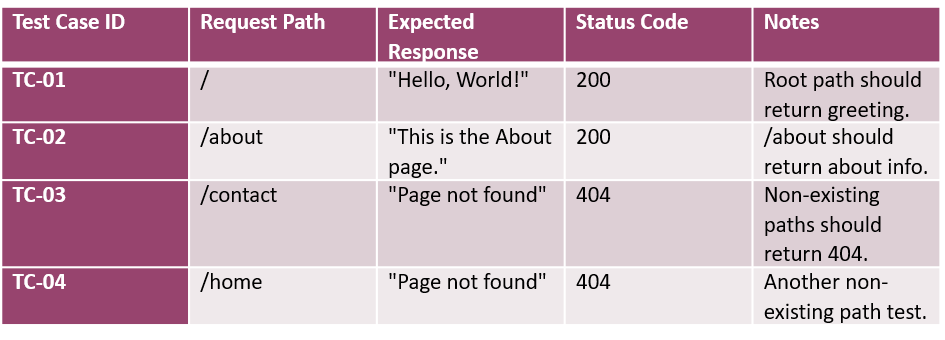
Create a simple Node.js HTTP server that listens on port 3000. The server should respond based on the following conditions:

* If the request is to the root path (/), respond with "Hello, World!".
* If the request is to the /about path, respond with "This is the About page."
* For any other path, respond with a 404 status and the message "Page not found".

**Functional Requirements**

1. The server should use the Node.js http module.
2. Proper HTTP response headers (e.g., Content-Type) should be set for all responses.
3. The server should run indefinitely until manually stopped.

**Test Cases:**



**Problem 2: Serving Static Files**

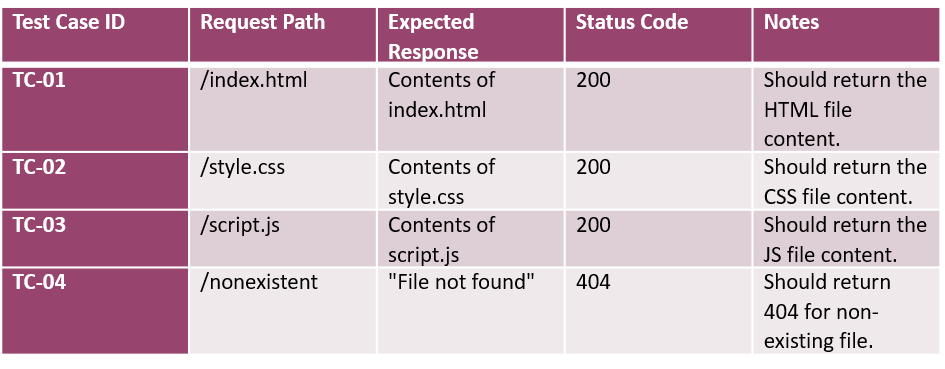
**Problem Statement**

Create a Node.js server that serves static files (HTML, CSS, JS) from the current directory. The server should return a proper 404 status code if the requested file doesn't exist.

**Functional Requirements**

1. The server should serve files based on the URL.
2. It should handle different file types like .html, .css, and .js.
3. If the file doesn't exist, the server should return a 404 error.

**Test Cases:**



**Problem 3: Basic Routing**

**Problem Statement**

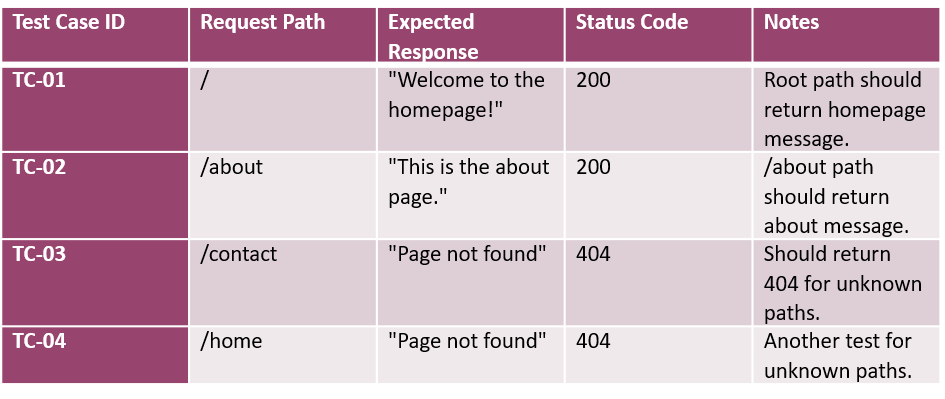
Create a Node.js server that routes requests based on the URL path. The server should handle the following:

* / should return "Welcome to the homepage!"
* /about should return "This is the about page."
* Any other URL should return "Page not found" with a 404 status.

**Functional Requirements**

1. The server should handle **multiple routes** using if/else conditions.
2. The server should return appropriate **status** **codes** and **messages** for each route.

**Test Cases:**



**Problem 4: Creating an API with Express (CRUD Operation using Node.js and Express Framework)**

**Problem Statement**

Create a **REST API** using **Express.js.** The API should support the following CRUD operations for managing items:

1. **GET /items**: Return a list of all items.
2. **POST /items**: Add a new item to the list.
3. **PUT /items/:id**: Update an item by ID.
4. **DELETE /items/:id**: Delete an item by ID.

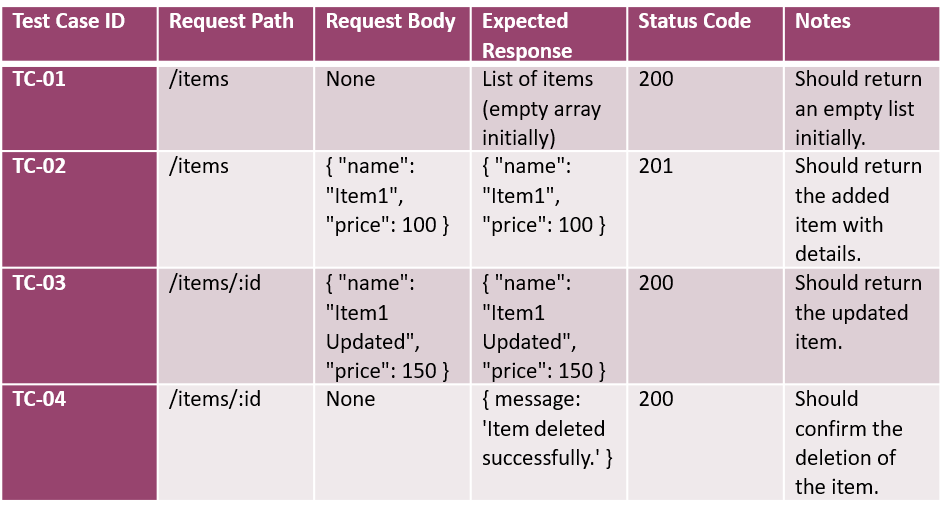
Additionally, create an HTML frontend that interacts with the API using forms and JavaScript. The HTML should allow the user to:

* View a list of items.
* Add a new item.
* Update an existing item.
* Delete an item.

**Functional Requirements**

1. The API should use JSON format for both requests and responses.
2. The POST and PUT methods should validate the request body.
3. Use Express.js for backend routing and handling API requests.
4. The HTML should allow the user to interact with the API for managing the items.

**Test Cases:**



**Problem 5: Interacting with an RDBMS Database (PostgreSQL)**

**Problem Statement:**

Create a simple **REST API** using **Express.js** and **PostgreSQL** that allows users to interact with a database. The API should support the following actions:

1. **Create**: Add a new user to the database.
2. **Read**: Retrieve all users from the database.
3. **Update**: Update a user's information in the database.
4. **Delete**: Delete a user from the database.

The database will store basic user information such as **id**, **name**, and **email**.

Additionally, implement the following:

* Handle database connection using **PostgreSQL**.
* Use **Express.js** to build the API routes.
* Handle errors appropriately and send proper HTTP status codes.

**Functional Requirements:**

1. **Create User**:
   * **POST** /api/users should allow you to add a new user with a **name** and **email**.
2. **Get All Users**:
   * **GET** /api/users should retrieve all users from the database.
3. **Update User**:
   * **PUT** /api/users/:id should allow you to update the **name** or **email** of a user based on their **id**.
4. **Delete User**:
   * **DELETE** /api/users/:id should allow you to delete a user by their **id**.

**Test Cases:**

A screenshot of a computer screen

Description automatically generated

**Problem 6: Connecting to a MongoDB** **Database**

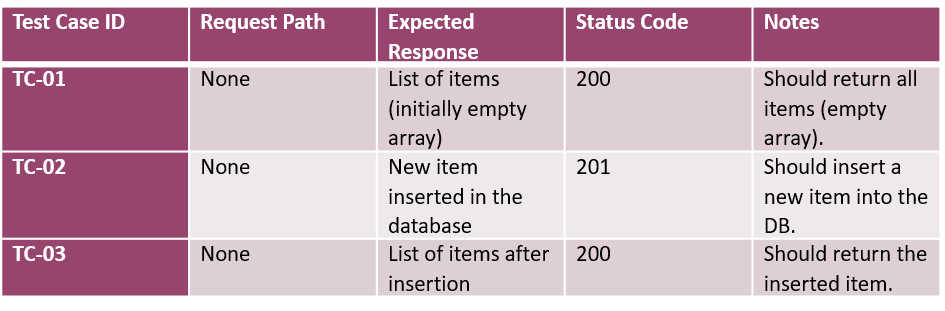
**Problem Statement**

Create a Node.js application that **connects to a MongoDB database** using **Mongoose**. The application should create and retrieve items from the database.

**Functional Requirements**

1. Use Mongoose to connect to a local MongoDB database (**testdb**).
2. Create an **Item model** with name and price fields.
3. **Insert a new item** and **retrieve** all items from the database.

**Test Cases:**



**Problem 7: Creating and Using Middleware**

**Problem Statement**

Create a Node.js application using Express.js with custom **middleware** to log all incoming requests.

**Functional Requirements**

1. Implement a middleware function that logs the HTTP method and URL of each request.
2. The middleware should run before the route handlers.

**Test Cases:**

A close-up of a chart

Description automatically generated

**Problem 8: Authentication with JWT**

**Problem Statement**

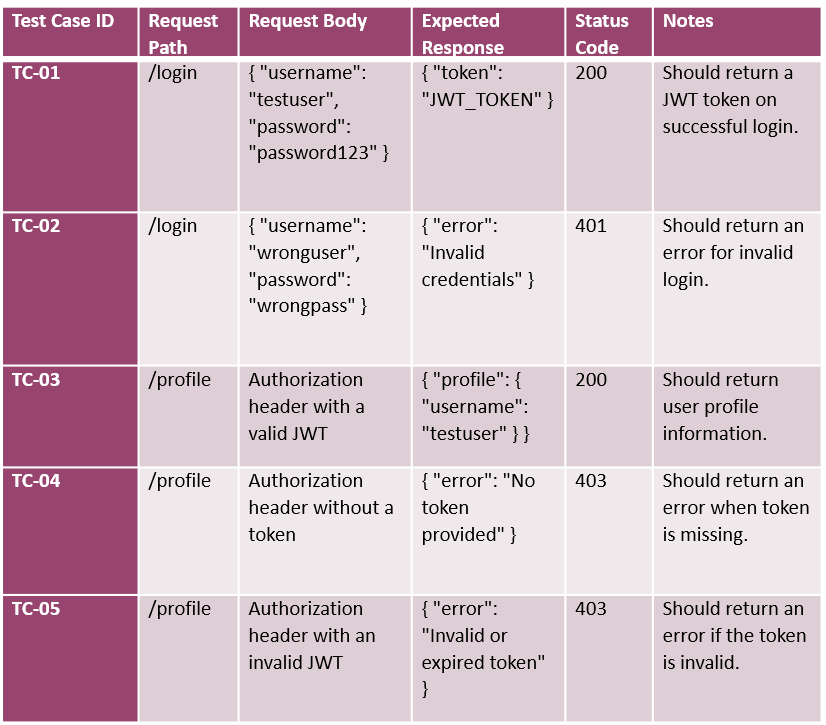
Create a simple authentication system using **JSON Web Tokens** (JWT). You will build the following components:

1. A **login route** to authenticate the user and issue a JWT.
2. A **protected route** that can only be accessed by users with a valid JWT.
3. An **HTML interface** that allows users to log in and access the protected route.

**Functional Requirements**

1. The backend should use **JWT** for user authentication.
2. The **POST /login** route should authenticate the user and **return a JWT** if the **username and password are correct.**
3. The **protected route** (e.g., /profile) should require the user to send the JWT in the request headers, and only authorized users should be able to access it.
4. Create an **HTML page** that:
   * Allows the user to log in with a username and password.
   * Displays a message with the user's profile information after logging in and accessing the protected route.

**Test Cases:**



**Problem 9: File Upload**

**Problem Statement**

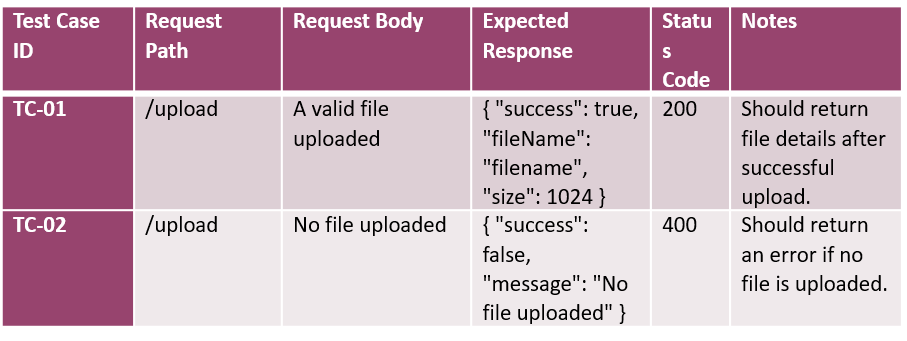
Create a file upload system using **Express**.js, where users can upload files to the server. The system should support the following functionality:

1. A **file upload form** that allows users to select a file and upload it to the server.
2. The backend should **save the file** on the server.
3. The backend should return the file's information (e.g., file name, size, and path) after successful upload.
4. A **UI** to interact with the file upload system.

**Functional Requirements**

1. Use **multer** middleware for handling file uploads in the backend.
2. The **POST /upload** route should handle file uploads and save the file to the server.
3. After successful upload, the server should return the file details as a response.
4. The HTML interface should allow the user to **select a file** and **submit** **the form to upload the file.**

**Test Cases:**



**Problem 10: Storing Files in Database**

**Problem Statement**

Create a system to upload and store files directly in a database (e.g., MongoDB or MySQL). The uploaded file content will be saved in the database, and the file metadata (e.g., file name, size, etc.) will be stored in a separate collection or table.

1. **File Upload API**: Accept file uploads from the client.
2. **File Storage**: Store the file content in the database as a binary object (e.g., BLOB in MySQL or Buffer in MongoDB).
3. **File Retrieval API**: Allow users to retrieve the uploaded file by its ID.
4. **UI for Uploading and Downloading Files**: Provide an HTML interface for users to upload files and download them.

**Functional Requirements**

1. The backend should store the file content in the database.
2. The **POST /upload** route should handle file uploads and save the file in the database.
3. The **GET /file/:id** route should return the file content from the database based on the file's ID.
4. The HTML interface should allow users to upload and download files.

**Test Cases:**

A table with text on it

Description automatically generated