PhotoStockage Backend Documentation

Table of Contents

1. Project Overview

PhotoStockage is a secure backend service for managing photo storage and sharing. It provides a RESTful API that handles user management, photo uploads, social interactions (likes, comments), and content categorization.

Key Features

- User authentication and authorization
- Photo management (upload, edit, delete)
- Social features (likes, comments)
- Download tracking
- Category management
- Admin functionality

Tech Stack

- Node.js
- Express.js
- PostgreSQL
- JWT for authentication
- Jest for testing

2. System Architecture

Core Components

```
Project Structure:
├── controllers/
│   ├── CategoriesController.js
│   ├── CommentsController.js
│   ├── DownloadsController.js
│   ├── LikesController.js
│   ├── PhotoController.js
│   └── UserController.js
├── middleware/
│   └── authMiddleware.js
├── models/
│   ├── CategoriesModel.js
│   ├── CommentsModel.js
│   ├── DownloadsModel.js
│   ├── LikesModel.js
│   ├── PhotoModel.js
│   └── UserModel.js
├── routes/
│   ├── categoriesRoute.js
│   ├── commentsRoute.js
│   ├── downloadsRoute.js
│   ├── likesRoute.js
│   ├── photoRoute.js
│   └── userRoute.js
├── utils/
│   └── db.js
└── index.js
```

3. Getting Started

Prerequisites

- Node.js v12 or higher
- PostgreSQL 12 or higher
- npm or yarn

Installation

```
1. Clone the repository
2. npm install
3. Set up environment variables:

DATABASE_URL=postgresql://username:password@localhost:5432/database
JWT_SECRET=your_jwt_secret
NODE_ENV=development

4. npm start
```

4. API Reference

User Management

Register User

```
POST /user/register
Content-Type: application/json

{
  "username": "string",
  "email": "string",
  "password": "string",
  "user_icon": "string" (optional)
}
```

Login

```
POST /user/login
Content-Type: application/json

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

Photo Management

Upload Photo

```
POST /photos/add_photo
Authorization: Bearer token
Content-Type: application/json

{
  "name": "string",
  "description": "string",
  "path": "string",
  "status": boolean
}
```

5. Database Schema

```
-- Users Table
CREATE TABLE users (
  id UUID PRIMARY KEY,
  username VARCHAR(255) UNIQUE NOT NULL,
  email VARCHAR(255) UNIQUE NOT NULL,
  password VARCHAR(255) NOT NULL,
  user_icon VARCHAR(255),
  access_level BOOLEAN DEFAULT FALSE
);

-- Photos Table
CREATE TABLE photos (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  name VARCHAR(255) NOT NULL,
```

```
  description TEXT,
  path VARCHAR(255),
  status BOOLEAN DEFAULT true,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

6. Authentication & Authorization

> **JWT Implementation:**
>
> - Tokens generated upon login/registration
> - 30-day expiration
> - Stored in HTTP-only cookies
> - Contains user ID, username, email, and access level

7. Security Features

- **Input Sanitization**
  - HTML sanitization using sanitize-html
  - Data validation with validator
- **Rate Limiting**
  - 100 requests per 15 minutes per IP
- **Security Headers**
  - Helmet middleware implementation
  - XSS protection
  - Content Security Policy

8. Testing

```
Test Structure:
__tests__/
├── integration/
│   ├── user.test.js
│   ├── comments.test.js
│   ├── likes.test.js
```

```
│   ├── photos.test.js
│   └── downloads.test.js
└── setup/
    └── testDb.js

Run tests: npm test
```

9. Error Handling

> **Standard Error Response Format**
>
> ```
> {
>   error: string,
>   details?: string,
>   message?: string
> }
> ```

> **HTTP Status Codes:**
>
> - 200: Success
> - 201: Created
> - 400: Bad Request
> - 401: Unauthorized
> - 403: Forbidden
> - 404: Not Found
> - 500: Internal Server Error