Dossier de projet

Name of project

Image

My name

Index

1. Introduction

i. Objective of the document

This document serves as a comprehensive guide for the project "photoStockage," offering an in-depth look at both the frontend and backend technologies employed. It outlines the decisions I made to ensure the successful completion of a fully functional CRUD (Create, Read, Update, Delete) web application.

The document begins by elucidating the project's purpose, providing a clear understanding of its objectives and goals. It then delves into detailed wireframes and prototypes, offering visual representations of the application's structure and flow. Additionally, it includes screenshots of key code segments, accompanied by thorough explanations to ensure clarity and understanding.

User experience diagrams and visual design decision explanations are also incorporated, providing insights into the thought process behind the application's interface and user interactions. Furthermore, the document includes comprehensive database schemas, outlining the structure and relationships within the database.

Testing strategies are discussed, complete with code examples to illustrate the methodologies used to ensure the application's reliability and performance. A detailed schema for deploying and maintaining the project in the future is also provided, ensuring long-term sustainability and scalability.

The document concludes with a reflection on the challenges encountered during the creation and development of the project, along with the solutions implemented to overcome these obstacles. Finally, it includes concise documentation of critical aspects such as installation procedures, backend API endpoints, and other essential components, ensuring a smooth and efficient development process.

ii. Project overview

“photoStockage” is a hybrid website that seamlessly blends the essence of social media with the functionality of an image-sharing platform. At its core, it serves as a space for users to upload, download, and explore images while drawing inspiration from a vast collection of visual content. The platform is designed to be fully responsive, ensuring a smooth and intuitive user experience across all devices, from mobile phones to large desktop screens.

The primary target audience of “photoStockage” consists of professionals who require high-quality, copyright-free images for various creative and commercial projects, provided they adhere to legal and ethical guidelines. A secondary audience includes artists, designers, and other creatives who use the platform as a source of inspiration, a networking hub, or a space for mentorship and collaboration. By fostering an engaging and supportive community, the platform bridges the gap between professional image resources and social interaction.

One of the defining features of “photoStockage” is its emphasis on user interaction and content management. Users can create, edit, and delete accounts while sharing their own images and personal stories behind them. The platform allows members to engage with content through likes, comments, and saved collections. Additionally, users can manage their interactions by editing or deleting comments and controlling the data they share. A dedicated support system is also in place, enabling direct communication with the site administration for any assistance or inquiries.

The purpose of “photoStockage” is to create an inclusive and dynamic environment where creativity and accessibility go hand in hand. By providing a legally sound repository of high-quality images, it empowers professionals and creatives to enhance their work without concerns over copyright restrictions. At the same time, the platform encourages artistic expression, idea exchange, and community-driven growth. Whether users are looking for the perfect visual asset, seeking artistic motivation, or aiming to connect with like-minded individuals, “photoStockage” offers a versatile and enriching experience.

2. Project Scope and Architecture

i. Scope

In “photoStockage”, I have personally developed the essential features that form the backbone of the platform. At its core, the website provides users with the ability to create a personal account, customize their profile, and manage their own content with ease. Each user has full control over their uploaded images, including the ability to add, edit, and delete content as needed. The platform is designed not only for sharing visuals but also for fostering engagement within the community, allowing users to interact with images through likes and comments, save their favorite photos in personalized collections, and explore a diverse range of creative content.

To ensure smooth and secure operations, I developed a backend API that enforces request limits, preventing abuse and maintaining system stability. Security has been a primary focus throughout development, leading me to integrate various protective measures to safeguard user data and interactions. In addition to authentication and data validation mechanisms, I have implemented security protocols to filter and sanitize user input, ensuring a safe environment for all members of the platform.

On the frontend, I have structured the interface to provide a seamless and intuitive user experience. By leveraging Next.js, I have taken advantage of its built-in optimizations, such as dynamic image handling and efficient page navigation, to enhance performance across different devices. The use of Next.js' Image and Link components has allowed me to optimize media delivery and ensure a smooth browsing experience, even when handling high-resolution images. By adhering to modern development best practices, I have aimed to create a platform that balances performance, usability, and scalability, making “photoStockage” accessible to a broad audience, from casual users to industry professionals.

When defining the target audience for “photoStockage”, I drew directly from my own experience as both a professional web developer and a creative. I have personally encountered the challenges faced by individuals in both of these roles, which helped shape the platform's core functionalities to address their specific needs. As a developer, I often required high-quality, royalty-free images for projects without legal restrictions, while as a creative, I sought inspiration, interaction, and opportunities to connect with others in the field. These experiences allowed me to structure “photoStockage” in a way that serves both groups effectively.

The primary audience consists of professionals, including web developers, designers, marketers, and content creators, who need access to high-resolution, copyright-free images for commercial and personal projects. To enhance their experience, I implemented a search functionality that allows users to find images by name or category, streamlining their workflow and making it easy to locate the perfect visual asset. By offering a free and accessible image repository, “photoStockage” eliminates the need for expensive stock image subscriptions, empowering professionals to focus on their projects without worrying about licensing restrictions.

The secondary audience comprises creatives such as photographers, artists, and designers who use the platform to showcase their work, seek inspiration, and engage with like-minded individuals. For them, “photoStockage” is more than just an image repository—it's a space for interaction and community-building. I developed features that encourage engagement, including a like button that allows users to appreciate and support others' work, as well as a commenting system where they can share feedback and start discussions. Additionally, I designed user profiles to include email contact information, enabling direct communication between members. This feature fosters potential collaborations, mentorship opportunities, and professional networking.

To further facilitate communication, I personally designed a Contact Us form that allows users to reach out directly to the platform's administration. Whether they have inquiries, need assistance, or want to report an issue, this feature ensures that users can receive support efficiently. By integrating these interactive elements, “photoStockage” serves not only as a practical tool for professionals but also as a dynamic creative hub where users can connect, share, and grow.

For the development of “photoStockage”, I chose technologies that align with both efficiency and ease of use. On the backend, I implemented Node.js with the Express framework, as it provides a lightweight and straightforward environment that enhances the development experience. For the frontend, I utilized Next.js, leveraging its App Router to optimize navigation and performance while maintaining a structured and scalable codebase. Given my prior experience with React, Next.js was a natural choice that allowed for seamless integration of best practices.

For data management, I opted for PostgreSQL, an open-source relational database with strong community support and excellent scalability. This choice ensures that the platform can efficiently handle data storage while remaining adaptable to future expansion if needed. While the platform is not yet hosted, it has been designed to be deployable on a VPS or traditional hosting services, with certain limitations on the number of concurrent users depending on the hosting environment.

To ensure security and performance, I adhered to the best practices recommended by the chosen frameworks. Additionally, I implemented security measures using various libraries to protect user data and maintain platform integrity. By combining these technologies, I have created a robust, scalable, and developer-friendly foundation for “photoStockage”, ensuring a smooth experience for all users.

As an independent project without external funding, the primary limitation of “photoStockage” is financial. Since there are no sponsors or revenue sources supporting its development, the scalability and long-term sustainability of the platform are directly tied to available resources. Hosting costs, in particular, play a crucial role in determining user capacity. While the project is designed to be deployable on both VPS and traditional hosting services, the number of users it can support would ultimately depend on the hosting infrastructure available at the time of deployment.

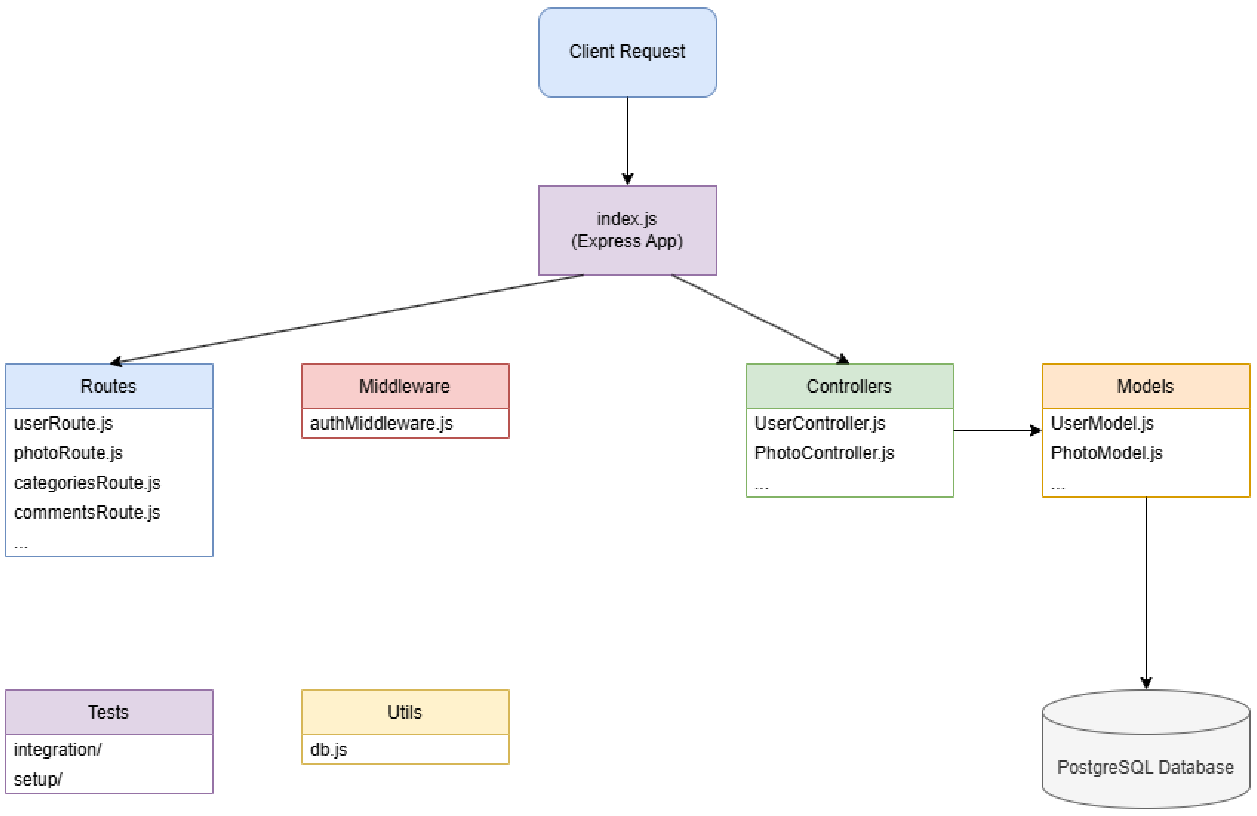
From a legal perspective, “photoStockage” includes a disclaimer in the "About" section that outlines user responsibilities and ensures compliance with copyright and content regulations. This disclaimer explicitly states that users must abide by the law when uploading and sharing content, and it relieves the administration of liability for any misuse of the platform.

To maintain a safe and appropriate environment, a moderation system is in place to address potential violations. If inappropriate or illegal content is detected by the administration, it is promptly removed. Additionally, users have the ability to report content, and depending on the severity of the violation, necessary actions are taken, including notifying the relevant authorities when required.

In terms of data privacy, “photoStockage” follows a minimal data retention policy. The platform does not collect or store unnecessary user information, ensuring a privacy-first approach. The only user data retained includes the email, username, and last login timestamp, which are stored securely in the database. These are essential for authentication purposes, particularly for securely managing JWT-based authentication using HTTPS cookies. By limiting data storage to only what is necessary, the platform enhances privacy while maintaining core functionality.

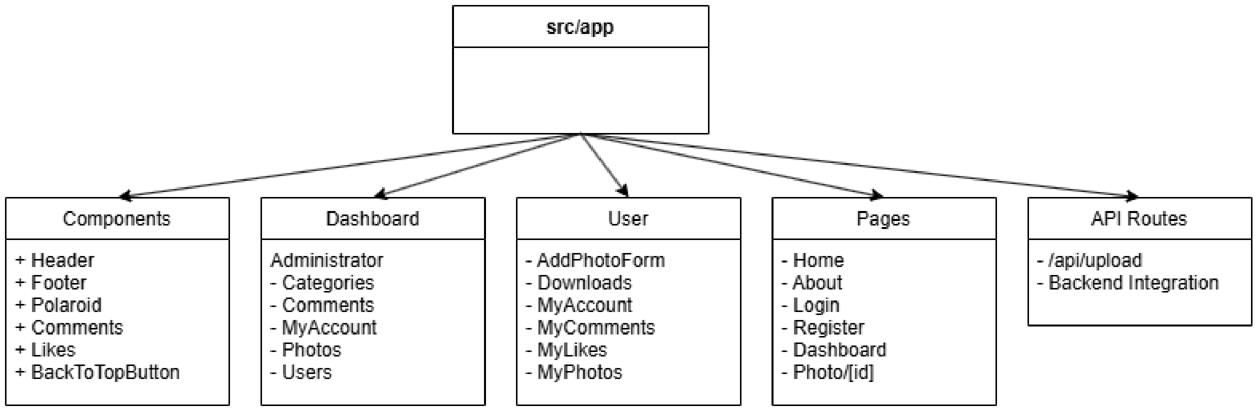
ii. Architecture overview

Backend flow diagram:



The entry point of the backend is the index file. It is written using the express framework, leveraging an MVC (without the view part as the frontend is separate) structure model. Upon a frontend request, the backend through index is making a call to the appropriate route which in turn is going to call a specific function of the controller that will finally call the appropriate model in order to retrieve or send data to the database. The operations that send data to the database are protected through a middleware that check if the user is authenticated and if necessary if the user is the admin for certain protected routes.

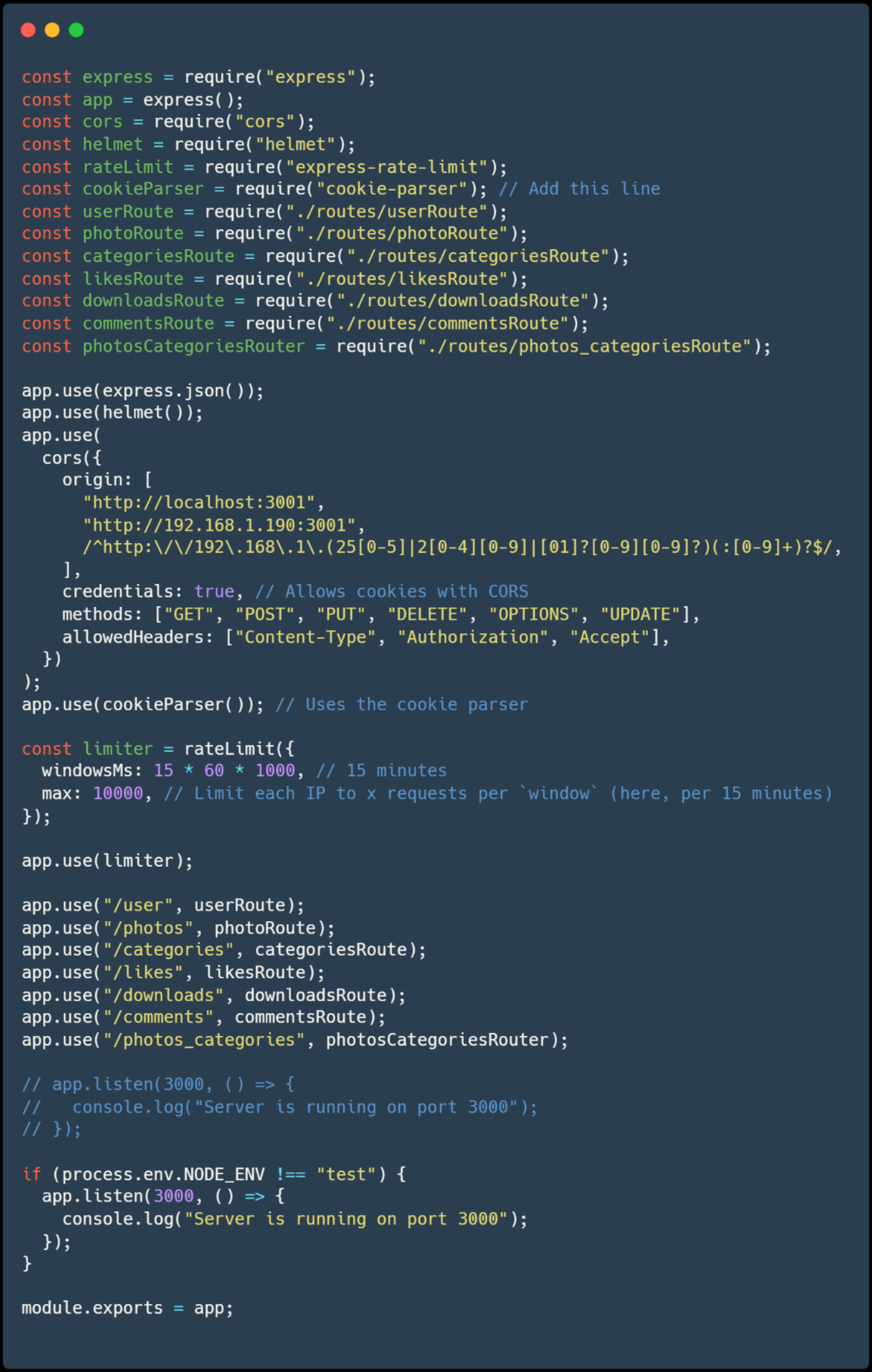
Frontend flow diagram:



The frontend is separated in sections. The main pages of the frontend consist of the homepage, an about page, a login page for users that are not logged in, a register page, a dashboard page for logged in users and the individual photo pages. It includes the header and footer components that are present throughout the whole experience. The polaroid component which is a component I created to imitate the well-known polaroid style of photos with some extra features like enlargement when hovered or touched for mobile screens and interesting shadows to give a depth of field. There are also the comments and likes components as well as a basic back to the top component that appears on every page when the user scrolled past a certain point of the page. I split the dashboard into two parts, the administration part and the user part, with separate, distinct, responsibilities. The administrator has the ability to view all photos, users, comments and delete them if that is deemed necessary. The user on the other hand, has the ability to upload images, edit them, delete them, same with comments, view their likes and downloads and of course manage their account. Every action that requires a database function involves an API call to the backend.

3. Code implementation

i. Backend



```
const express = require("express");
const app = express();
const cors = require("cors");
const helmet = require("helmet");
const rateLimit = require("express-rate-limit");
const cookieParser = require("cookie-parser"); // Add this line
const userRoute = require("./routes/userRoute");
const photoRoute = require("./routes/photoRoute");
const categoriesRoute = require("./routes/categoriesRoute");
const likesRoute = require("./routes/likesRoute");
const downloadsRoute = require("./routes/downloadsRoute");
const commentsRoute = require("./routes/commentsRoute");
const photosCategoriesRouter = require("./routes/photos_categoriesRoute");

app.use(express.json());
app.use(helmet());
app.use(
  cors({
    origin: [
      "http://localhost:3001",
      "http://192.168.1.190:3001",
      /^http:\/\/192\.168\.1\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)(:[0-9]+)?$/,
    ],
    credentials: true, // Allows cookies with CORS
    methods: ["GET", "POST", "PUT", "DELETE", "OPTIONS", "UPDATE"],
    allowedHeaders: ["Content-Type", "Authorization", "Accept"],
  })
);
app.use(cookieParser()); // Uses the cookie parser

const limiter = rateLimit({
  windowsMs: 15 * 60 * 1000, // 15 minutes
  max: 10000, // Limit each IP to x requests per `window` (here, per 15 minutes)
});

app.use(limiter);

app.use("/user", userRoute);
app.use("/photos", photoRoute);
app.use("/categories", categoriesRoute);
app.use("/likes", likesRoute);
app.use("/downloads", downloadsRoute);
app.use("/comments", commentsRoute);
app.use("/photos_categories", photosCategoriesRouter);

// app.listen(3000, () => {
//   console.log("Server is running on port 3000");
// });

if (process.env.NODE_ENV !== "test") {
  app.listen(3000, () => {
    console.log("Server is running on port 3000");
  });
}

module.exports = app;
```

As mentioned earlier, the entry point for the backend is the index file. At the top of the file all the required libraries are called. The app is using express, helmet for security, cors also to restricted access from unknown sources, a cookie parser for the decoding of the jwt that is stored in an https only cookie, a limiter for limiting requests in order to avoid DOS attacks, followed by the routes and finally the definition of the port that the backend is listening to for active connections.

The cors section accepts localhost connections that are only coming from the port 3001 which is used by the frontend, the local address 192.168.1.190 which at the time of coding was the address assigned to my mobile phone for testing purposes as well as any local address in the range of 192.168.1.x/16 (0-255) which are local addresses for further testing. Credentials are required, meaning for certain routes a jwt is required. The methods that are allowed are GET methods in order to fetch data from the database and the rest of them that are in order to make a manipulation to the database which are all protected with the use of the middleware.

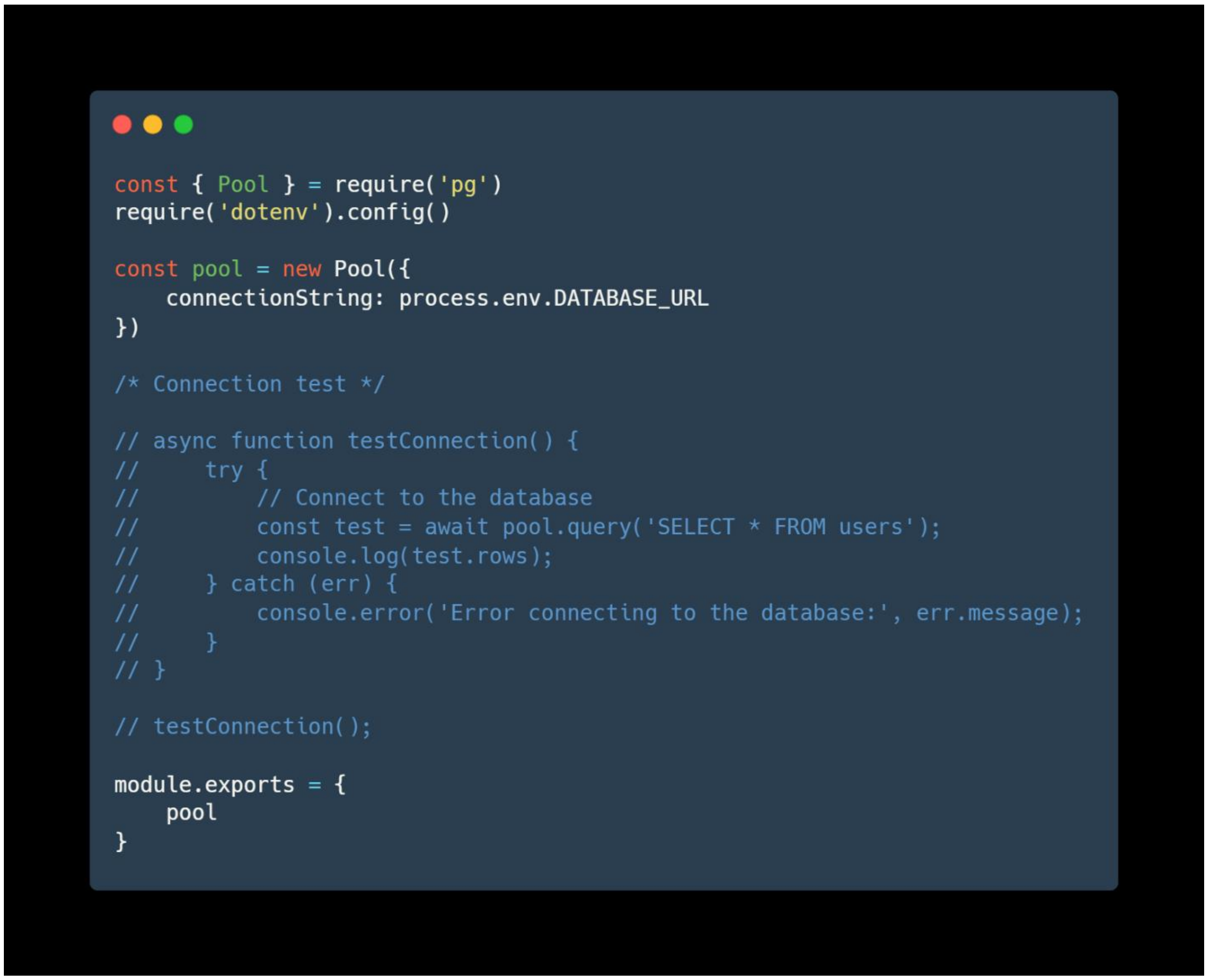
The app limiter restricts to 10.000 requests every 15 minutes in order to protect from malicious usage of the platform.

The app router calls all the required routes in order for the platform to function.

Finally we listen to a specified port for connections as long as the environment is set to be anything else except for test.

I chose the MVC structure as it helps me keep my files and folders more organized in a cleaner way and more important, it separates concerns of files which is a practice that keeps everything neat.

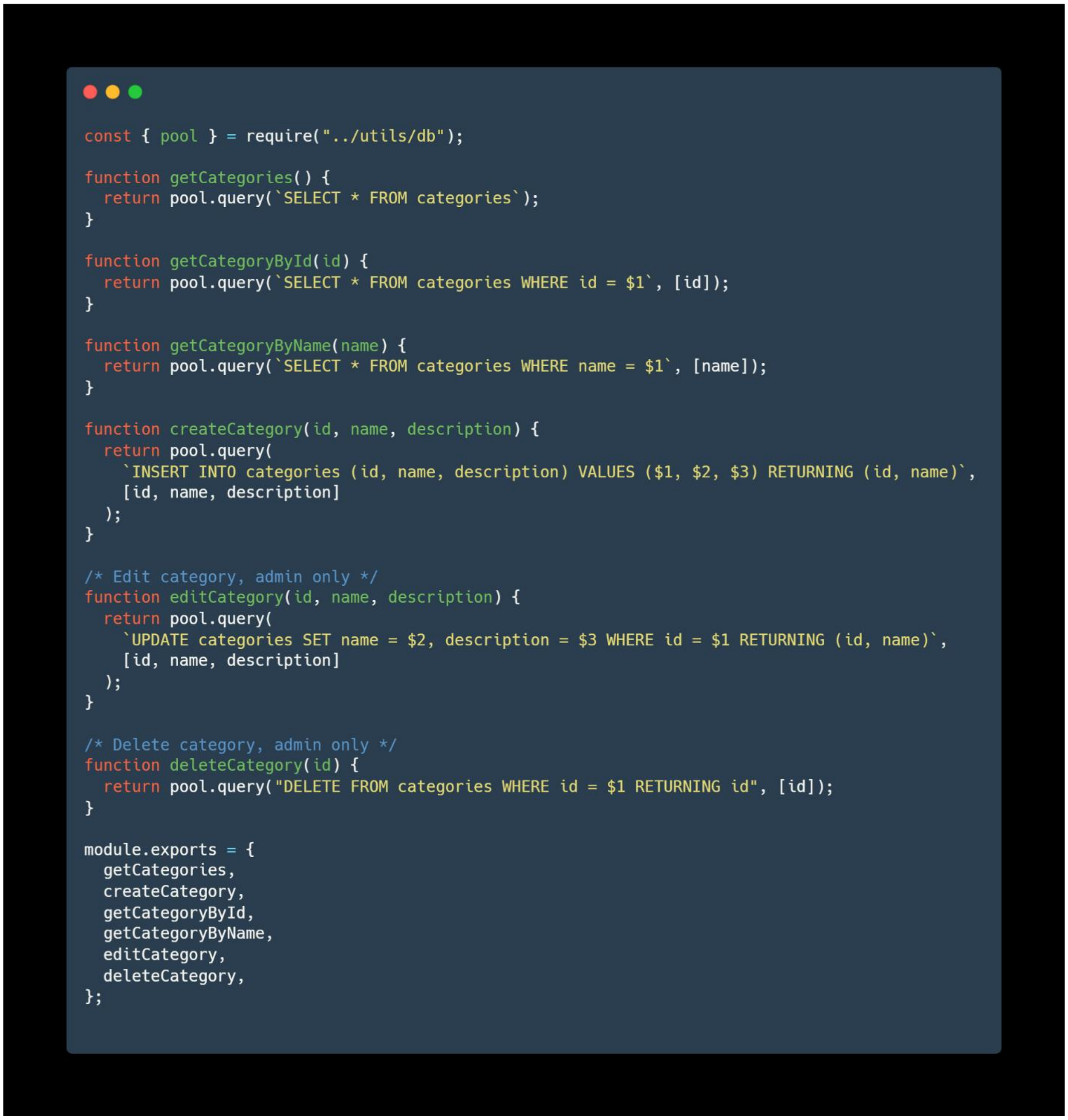
Database connection:



```
const { Pool } = require('pg')
require('dotenv').config()

const pool = new Pool({
    connectionString: process.env.DATABASE_URL
})

/* Connection test */

// async function testConnection() {
//     try {
//         // Connect to the database
//         const test = await pool.query('SELECT * FROM users');
//         console.log(test.rows);
//     } catch (err) {
//         console.error('Error connecting to the database:', err.message);
//     }
// }

// testConnection();

module.exports = {
    pool
}
```

For the database connection I am using the “pg” library which is a library used to make connections to a PostgreSQL database. From it I call an object called Pool in order to handle, open and close, the connections to the database using the URL coming from the .env file (for security reasons it will not be shared here). The file also includes a connection test function that is used to determine whether the interaction with the database is successful or not. Finally, pool is exported in order to be used by other files.

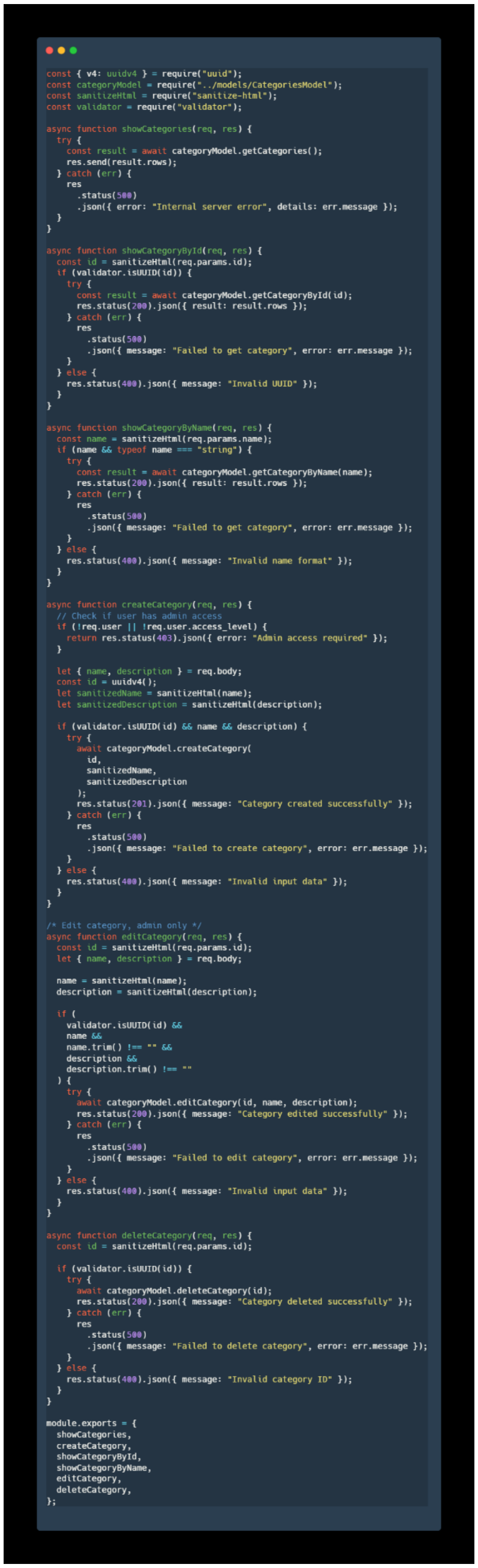
Models:



```javascript
const { pool } = require("../utils/db");

function getCategories() {
  return pool.query(`SELECT * FROM categories`);
}

function getCategoryById(id) {
  return pool.query(`SELECT * FROM categories WHERE id = $1`, [id]);
}

function getCategoryByName(name) {
  return pool.query(`SELECT * FROM categories WHERE name = $1`, [name]);
}

function createCategory(id, name, description) {
  return pool.query(
    `INSERT INTO categories (id, name, description) VALUES ($1, $2, $3) RETURNING (id, name)`,
    [id, name, description]
  );
}

/* Edit category, admin only */
function editCategory(id, name, description) {
  return pool.query(
    `UPDATE categories SET name = $2, description = $3 WHERE id = $1 RETURNING (id, name)`,
    [id, name, description]
  );
}

/* Delete category, admin only */
function deleteCategory(id) {
  return pool.query("DELETE FROM categories WHERE id = $1 RETURNING id", [id]);
}

module.exports = {
  getCategories,
  createCategory,
  getCategoryById,
  getCategoryByName,
  editCategory,
  deleteCategory,
};
```

This is a model example from the categories model. At the very top of the file we call pool which is coming from the database file as described above in order to establish connections to the database. As an example, the first function is a performing a SELECT statement to the database in order to retrieve all the fields and the values from the table "categories" of the database by performing the query `SELECT * FROM categories`. After all the desired functions are defined, they are exported in order to be imported and used by the controller.

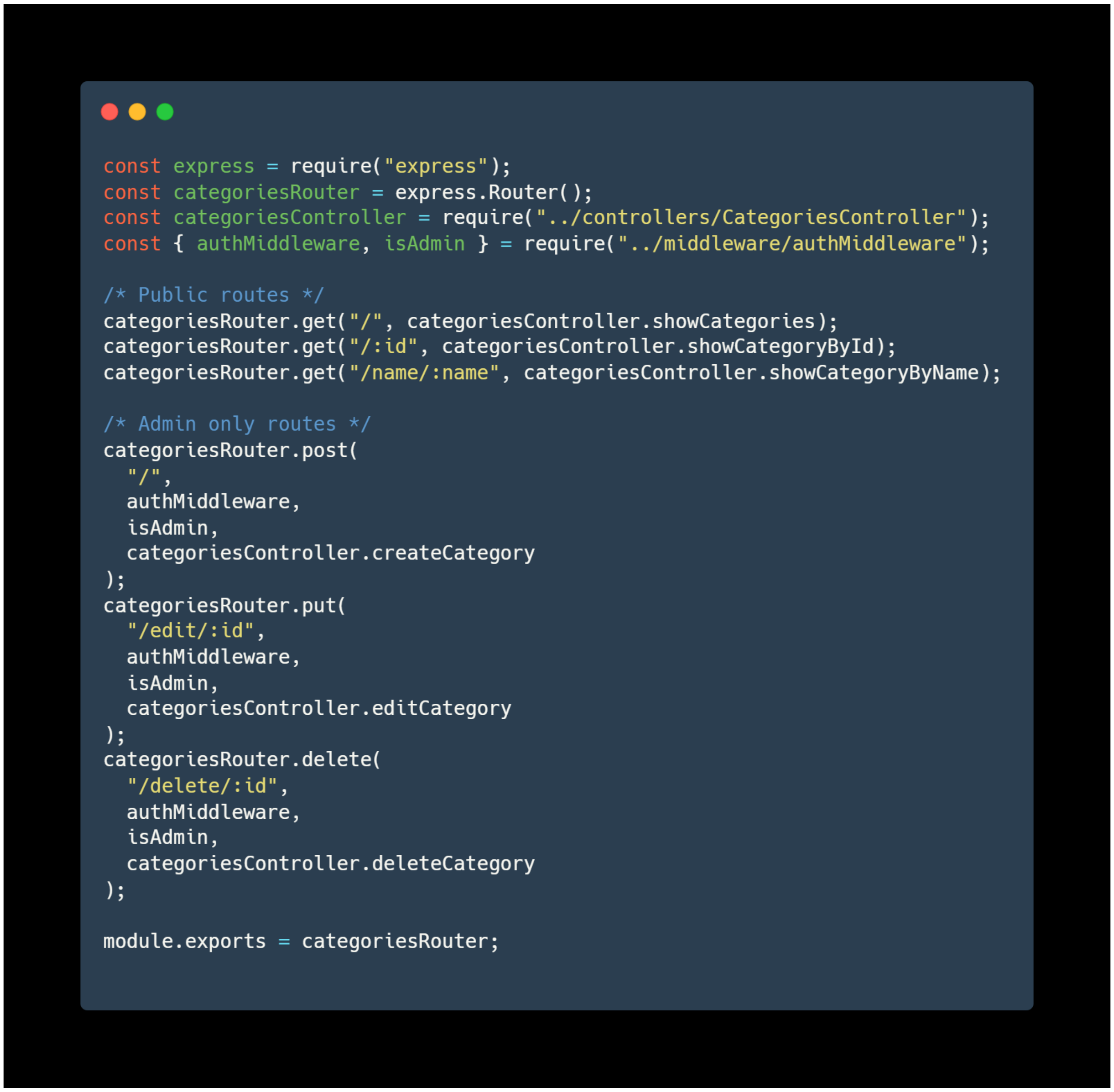
Controller:



```javascript
const { v4: uuidv4 } = require("uuid");
const categoryModel = require("../models/CategoriesModel");
const sanitizeHtml = require("sanitize-html");
const validator = require("validator");

async function showCategories(req, res) {
  try {
    const result = await categoryModel.getCategories();
    res.send(result.rows);
  } catch (err) {
    res
      .status(500)
      .json({ error: "Internal server error", details: err.message });
  }
}

async function showCategoryById(req, res) {
  const id = sanitizeHtml(req.params.id);
  if (validator.isUUID(id)) {
    try {
      const result = await categoryModel.getCategoryById(id);
      res.status(200).json({ result: result.rows });
    } catch (err) {
      res
        .status(500)
        .json({ message: "Failed to get category", error: err.message });
    }
  } else {
    res.status(400).json({ message: "Invalid UUID" });
  }
}

async function showCategoryByName(req, res) {
  const name = sanitizeHtml(req.params.name);
  if (name && typeof name === "string") {
    try {
      const result = await categoryModel.getCategoryByName(name);
      res.status(200).json({ result: result.rows });
    } catch (err) {
      res
        .status(500)
        .json({ message: "Failed to get category", error: err.message });
    }
  } else {
    res.status(400).json({ message: "Invalid name format" });
  }
}

async function createCategory(req, res) {
  // Check if user has admin access
  if (!req.user || !req.user.access_level) {
    return res.status(403).json({ error: "Admin access required" });
  }

  let { name, description } = req.body;
  const id = uuidv4();
  let sanitizedName = sanitizeHtml(name);
  let sanitizedDescription = sanitizeHtml(description);

  if (validator.isUUID(id) && name && description) {
    try {
      await categoryModel.createCategory(
        id,
        sanitizedName,
        sanitizedDescription
      );
      res.status(201).json({ message: "Category created successfully" });
    } catch (err) {
      res
        .status(500)
        .json({ message: "Failed to create category", error: err.message });
    }
  } else {
    res.status(400).json({ message: "Invalid input data" });
  }
}

/* Edit category, admin only */
async function editCategory(req, res) {
  const id = sanitizeHtml(req.params.id);
  let { name, description } = req.body;

  name = sanitizeHtml(name);
  description = sanitizeHtml(description);

  if (
    validator.isUUID(id) &&
    name &&
    name.trim() !== "" &&
    description &&
    description.trim() !== ""
  ) {
    try {
      await categoryModel.editCategory(id, name, description);
      res.status(200).json({ message: "Category edited successfully" });
    } catch (err) {
      res
        .status(500)
        .json({ message: "Failed to edit category", error: err.message });
    }
  } else {
    res.status(400).json({ message: "Invalid input data" });
  }
}

async function deleteCategory(req, res) {
  const id = sanitizeHtml(req.params.id);

  if (validator.isUUID(id)) {
    try {
      await categoryModel.deleteCategory(id);
      res.status(200).json({ message: "Category deleted successfully" });
    } catch (err) {
      res
        .status(500)
        .json({ message: "Failed to delete category", error: err.message });
    }
  } else {
    res.status(400).json({ message: "Invalid category ID" });
  }
}

module.exports = {
  showCategories,
  createCategory,
  showCategoryById,
  showCategoryByName,
  editCategory,
  deleteCategory,
};
```

Following the categories model, we have the categories controller. At the top of the file I perform the necessary imports, uuid for generating ids for the categories, the model itself so I can use its functions, sanitize-html and validator for security purposes. Taking as an example the creation of a category: first we check if the user performing the request is an administrator or not, if not, a status error 403 is returned meaning an unauthorized attempt of a request. Following we have the definition of variables. The reason it comes after the validation of the user is to avoid assigning addresses in memory when the request is unauthorized. I define the name of the category and the description of it, coming from the body of the request. Then I define an id using the “uuid” library that randomizes a text to assign and use an id. Then I sanitize my name and description variables and check if they are of the correct types using validator, if not I get an error 400, meaning there was a problem with the data that were sent. After, I try to send the request using the model’s “createCategory” function, passing the data required. If everything succeeds, I get a status code 201 with the message "Category created successfully" which means that everything went well. If there is a problem an error 500 is returned meaning there was an internal server error. Finally export my functions in order to be used by the routes.

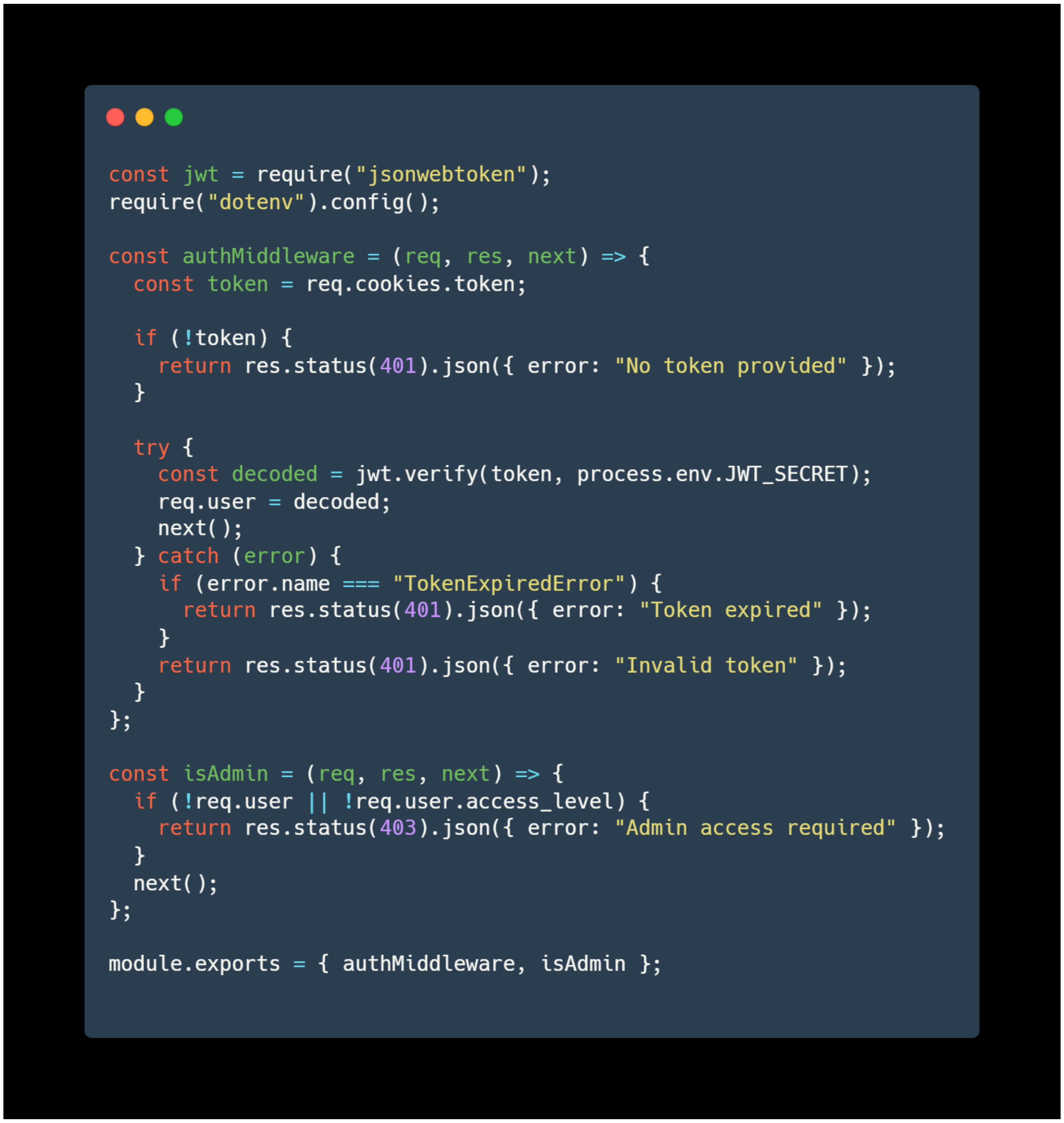
Routes:



```
const express = require("express");
const categoriesRouter = express.Router();
const categoriesController = require("../controllers/CategoriesController");
const { authMiddleware, isAdmin } = require("../middleware/authMiddleware");

/* Public routes */
categoriesRouter.get("/", categoriesController.showCategories);
categoriesRouter.get("/:id", categoriesController.showCategoryById);
categoriesRouter.get("/name/:name", categoriesController.showCategoryByName);

/* Admin only routes */
categoriesRouter.post(
  "/",
  authMiddleware,
  isAdmin,
  categoriesController.createCategory
);
categoriesRouter.put(
  "/edit/:id",
  authMiddleware,
  isAdmin,
  categoriesController.editCategory
);
categoriesRouter.delete(
  "/delete/:id",
  authMiddleware,
  isAdmin,
  categoriesController.deleteCategory
);

module.exports = categoriesRouter;
```

Using the categories route as an example, I import express and from express I use its Router method. I import the controller and the middleware's function that authenticate the user and check if the user is an administrator. A typical route will use a method followed by the URL (eg. "/") followed by the middleware functions if the route is protected or directly the controller function that needs to be executed. Finally the router is exported to be used from the index file as mentioned in the beginning of the chapter.

Middleware:



```
const jwt = require("jsonwebtoken");
require("dotenv").config();

const authMiddleware = (req, res, next) => {
  const token = req.cookies.token;

  if (!token) {
    return res.status(401).json({ error: "No token provided" });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    if (error.name === "TokenExpiredError") {
      return res.status(401).json({ error: "Token expired" });
    }
    return res.status(401).json({ error: "Invalid token" });
  }
};

const isAdmin = (req, res, next) => {
  if (!req.user || !req.user.access_level) {
    return res.status(403).json({ error: "Admin access required" });
  }
  next();
};

module.exports = { authMiddleware, isAdmin };
```

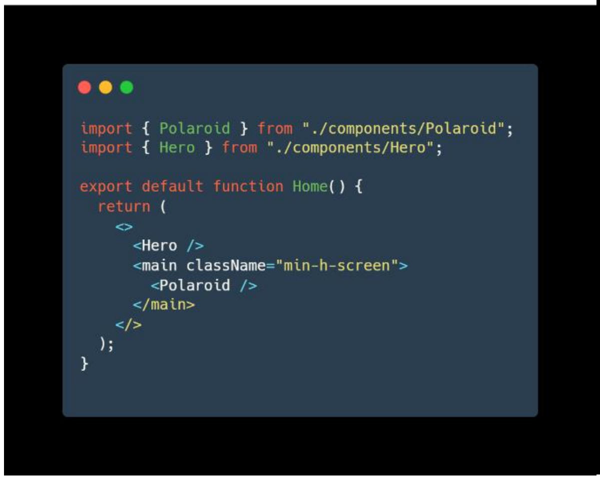
The middleware requires the use of jwt and the .env file variables. For the "authMiddleware" function I define the token coming from the requests cookies. If it does not exist, an error message is returned notifying the user that they did not provide a token. If the token exists, the function then tries to decode the token using the secret I defined in the .env file. The use of next(); is important, if it is not present the application will finish the action here, not moving to the next function. If there is an error, we get an appropriate message (eg: Token expired, or the token is invalid). The isAdmin function checks the if the user coming for the request has an access level of administrator and moves to the next function. The functions are also exported so they can be used by other files.

ii. Frontend

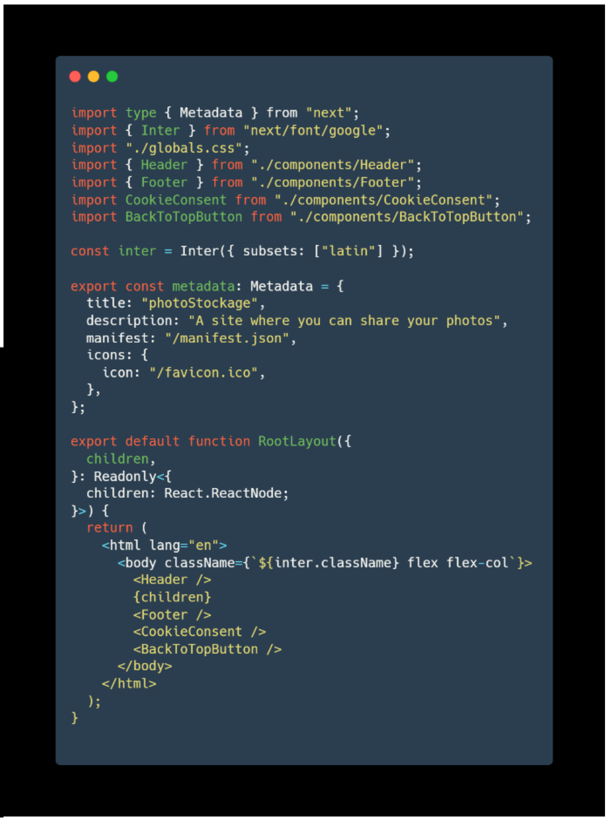
For the frontend I used Nextjs because I am more experienced with React than with Angular and Nextjs' app router is a better developer experience than React's router. Nextjs also offers features like the Link and Image components, server side rendering and data caching by default which makes content management a lot easier and intuitive.

The libraries, also called packages, that are used in the project can be found in the package.json and package-lock.json files. I also chose to use typescript as it enables the usage of types for javascript which makes the code easier to read and type safe. Also, I am using tailwind throughout the project as it allows more flexibility that vanilla css and avoids the pitfalls of using bootstrap.

The frontend part of the project is split into pages, components and utilities. The entry point of the frontend is the homepage.



```
import { Polaroid } from "./components/Polaroid";
import { Hero } from "./components/Hero";

export default function Home() {
  return (
    <>
      <Hero />
      <main className="min-h-screen">
        <Polaroid />
      </main>
    </>
  );
}
```

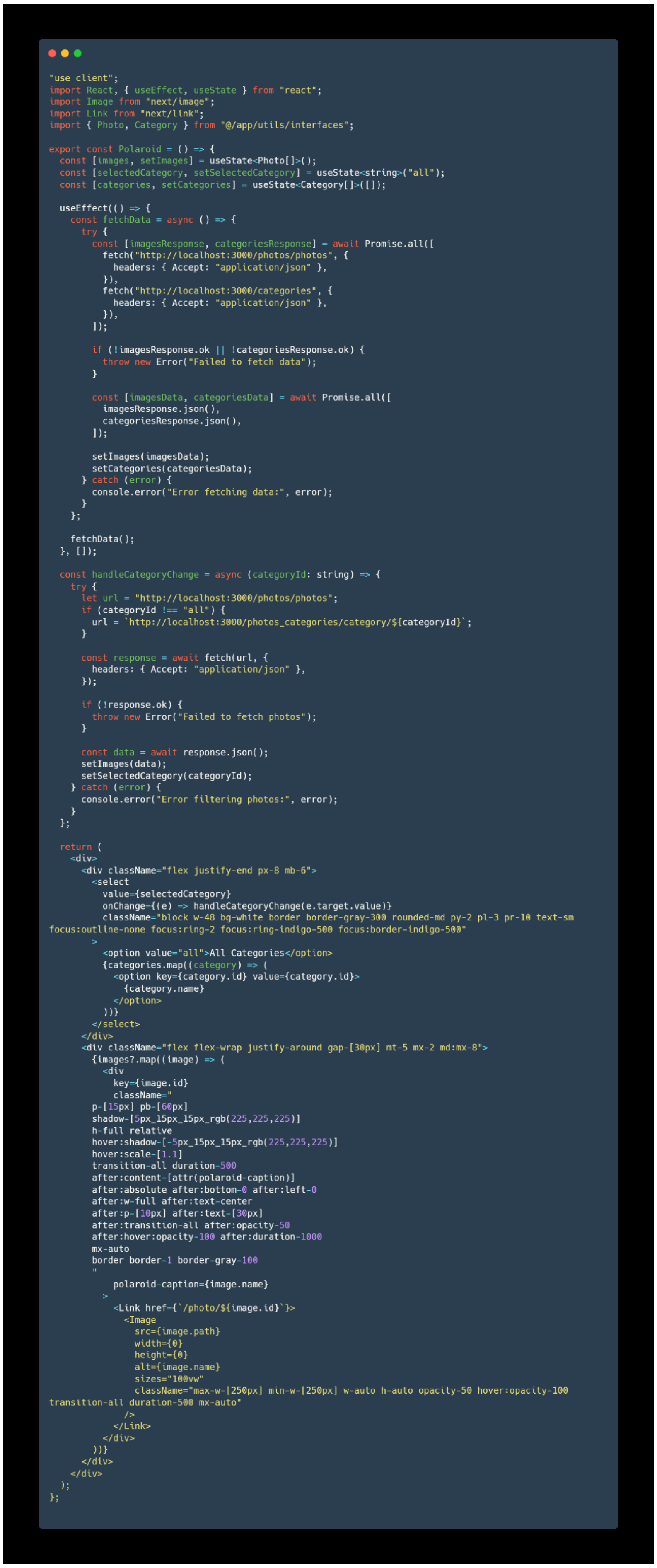


```
import type { Metadata } from "next";
import { Inter } from "next/font/google";
import "./globals.css";
import { Header } from "./components/Header";
import { Footer } from "./components/Footer";
import CookieConsent from "./components/CookieConsent";
import BackToTopButton from "./components/BackToTopButton";

const inter = Inter({ subsets: ["latin"] });

export const metadata: Metadata = {
  title: "photoStockage",
  description: "A site where you can share your photos",
  manifest: "/manifest.json",
  icons: {
    icon: "/favicon.ico",
  },
};

export default function RootLayout({
  children,
}: Readonly<{
  children: React.ReactNode;
}>) {
  return (
    <html lang="en">
      <body className={`${inter.className} flex flex-col`}>
        <Header />
        {children}
        <Footer />
        <CookieConsent />
        <BackToTopButton />
      </body>
    </html>
  );
}
```

In the homepage, I am importing the Polaroid component and the Hero component which are components I made for this project. Then the page is returned following the structure: Hero component -> a main html element with a minimum height set to cover the entire viewport -> the Polaroid component as its child. They are all wrapped in an empty element also known as React Fragment.

The layout file sets the structure of the pages (it is maintain throughout the entire project) in the following way: at the top I import the necessary methods, functions, libraries like the Metadata type from next, the Inter font family from google, the global css file that contains only the imports for tailwind, the Header and Footer components, the CookieConsent and BackToTheTop components. I then define the subset of my font and my metadata for the project like the title for example. The I define the structure to be followed when rendering a page: the html tag with the lang property set to English. The body element that will be using the inter font, a display flex and the direction of column for flex. The Header component renders at the top followed by the pages children, then the Footer component and finally the CookieConsent and BackToTheTop components.

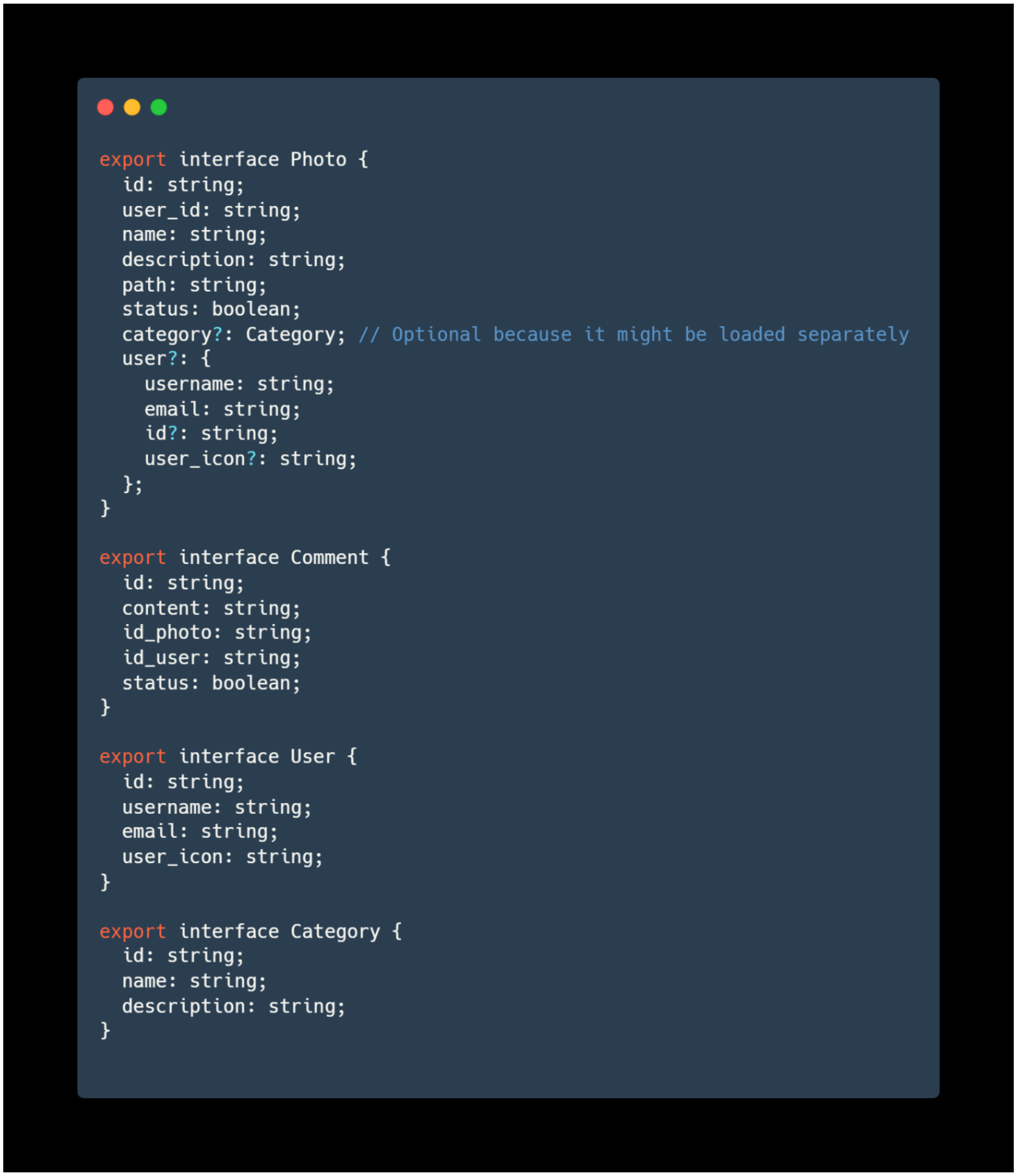
Polaroid component:



```
"use client";
import React, { useEffect, useState } from "react";
import Image from "next/image";
import Link from "next/link";
import { Photo, Category } from "@/app/utils/interfaces";

export const Polaroid = () => {
  const [images, setImages] = useState<Photo[]>();
  const [selectedCategory, setSelectedCategory] = useState<string>("all");
  const [categories, setCategories] = useState<Category[]>([]);

  useEffect(() => {
    const fetchData = async () => {
      try {
        const [imagesResponse, categoriesResponse] = await Promise.all([
          fetch("http://localhost:3000/photos/photos", {
            headers: { Accept: "application/json" },
          }),
          fetch("http://localhost:3000/categories", {
            headers: { Accept: "application/json" },
          }),
        ]);

        if (!imagesResponse.ok || !categoriesResponse.ok) {
          throw new Error("Failed to fetch data");
        }

        const [imagesData, categoriesData] = await Promise.all([
          imagesResponse.json(),
          categoriesResponse.json(),
        ]);

        setImages(imagesData);
        setCategories(categoriesData);
      } catch (error) {
        console.error("Error fetching data:", error);
      }
    };

    fetchData();
  }, []);

  const handleCategoryChange = async (categoryId: string) => {
    try {
      let url = "http://localhost:3000/photos/photos";
      if (categoryId !== "all") {
        url = `http://localhost:3000/photos_categories/category/${categoryId}`;
      }

      const response = await fetch(url, {
        headers: { Accept: "application/json" },
      });

      if (!response.ok) {
        throw new Error("Failed to fetch photos");
      }

      const data = await response.json();
      setImages(data);
      setSelectedCategory(categoryId);
    } catch (error) {
      console.error("Error filtering photos:", error);
    }
  };

  return (
    <div>
      <div className="flex justify-end px-8 mb-6">
        <select
          value={selectedCategory}
          onChange={(e) => handleCategoryChange(e.target.value)}
          className="block w-48 bg-white border border-gray-300 rounded-md py-2 pl-3 pr-10 text-sm
focus:outline-none focus:ring-2 focus:ring-indigo-500 focus:border-indigo-500"
        >
          <option value="all">All Categories</option>
          {categories.map((category) => (
            <option key={category.id} value={category.id}>
              {category.name}
            </option>
          ))}
        </select>
      </div>
      <div className="flex flex-wrap justify-around gap-[30px] mt-5 mx-2 md:mx-8">
        {images?.map((image) => (
          <div
            key={image.id}
            className="
        p-[15px] pb-[60px]
        shadow-[5px_15px_15px_rgb(225,225,225)]
        h-full relative
        hover:shadow-[-5px_15px_15px_rgb(225,225,225)]
        hover:scale-[1.1]
        transition-all duration-500
        after:content-[attr(polaroid-caption)]
        after:absolute after:bottom-0 after:left-0
        after:w-full after:text-center
        after:p-[10px] after:text-[30px]
        after:transition-all after:opacity-50
        after:hover:opacity-100 after:duration-1000
        mx-auto
        border border-1 border-gray-100
        "
            polaroid-caption={image.name}
          >
            <Link href={`/photo/${image.id}`}>
              <Image
                src={image.path}
                width={0}
                height={0}
                alt={image.name}
                sizes="100vw"
                className="max-w-[250px] min-w-[250px] w-auto h-auto opacity-50 hover:opacity-100
transition-all duration-500 mx-auto"
              />
            </Link>
          </div>
        ))}
      </div>
    </div>
  );
};
```

The polaroid component uses client rendering as it involves hooks and hook actions can only happen on the client side. React and the hooks useEffect and useState are imported directly from React as well as Image and Link from next. Finally the interfaces for Photo and Category, as defined by me, are also imported. In the Polaroid component itself we define the images of type array of Photo, selectedCategory of type string, categories of type array of Category. All of them are using the hook useState in order for the component to "remember" any changes performed during the navigation of the user. A useEffect is called because of the need of retrieving data from the database. In particular, the active photos and categories are returned followed by error checking. Then, the images value is set to the result of the image fetch request and the categories value is set to the result of the category fetch request. I call the function in order to execute it and I define an empty dependency array so the request is only executed once per page refresh. I also set a function that changes the category id sent to the API request accordingly. I then request a render of my component. I wrap the entire component in a div instead of React Fragment as I want to keep it as distinct element. I then set a div to have a display flex with justifying the content to the right (end), a padding on the x axis of 8rems and margin of 6rems for the bottom. I then set the my list for categories. When a different option is selected the function to change the categories is called and a new call to the API is made in order to fetch the corresponding data. After the list element I create my images gallery. A key is required so that Nextjs can distinguish the elements, making them unique so it knows which ones would probably require re-rendering if a change happened. The Image is wrapped in a Link. The image is self is having its full quality by using the sizes property but its width is constrained to 250px. There is also an alt tag which serves for the visually impaired as well as connection problems as it is rendered as text which can be read out loud by readers for those with eyesight issues.

Interfaces and Types:

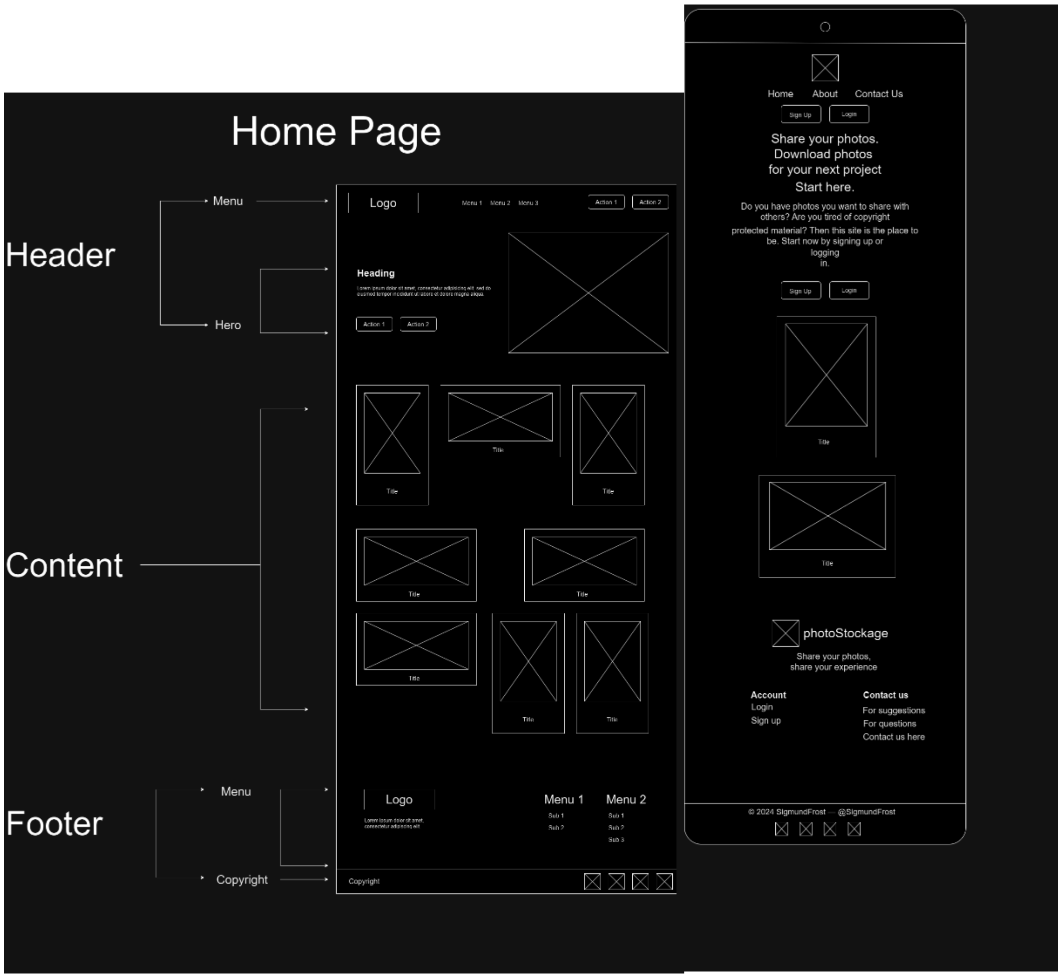


```typescript
export interface Photo {
  id: string;
  user_id: string;
  name: string;
  description: string;
  path: string;
  status: boolean;
  category?: Category; // Optional because it might be loaded separately
  user?: {
    username: string;
    email: string;
    id?: string;
    user_icon?: string;
  };
}

export interface Comment {
  id: string;
  content: string;
  id_photo: string;
  id_user: string;
  status: boolean;
}

export interface User {
  id: string;
  username: string;
  email: string;
  user_icon: string;
}

export interface Category {
  id: string;
  name: string;
  description: string;
}
```

A Photo, Comment, User and Category interfaces are defined as they necessary for components that involve them. For example the Photo interface is defined by and id of type string that is required, a user_id of type string, a name of type string, a description of type string, a path of type string, a status of type Boolean, an optional category and an optional object user that contains a username of type string, an email of type string, an optional id of type string and an optional user_icon of type string. This describes that data that is expected to be retrieved by the database and, more importantly, the structure of the data to be sent to the database in order to avoid violations and errors.

4. Design and User Experience

i. Wireframes

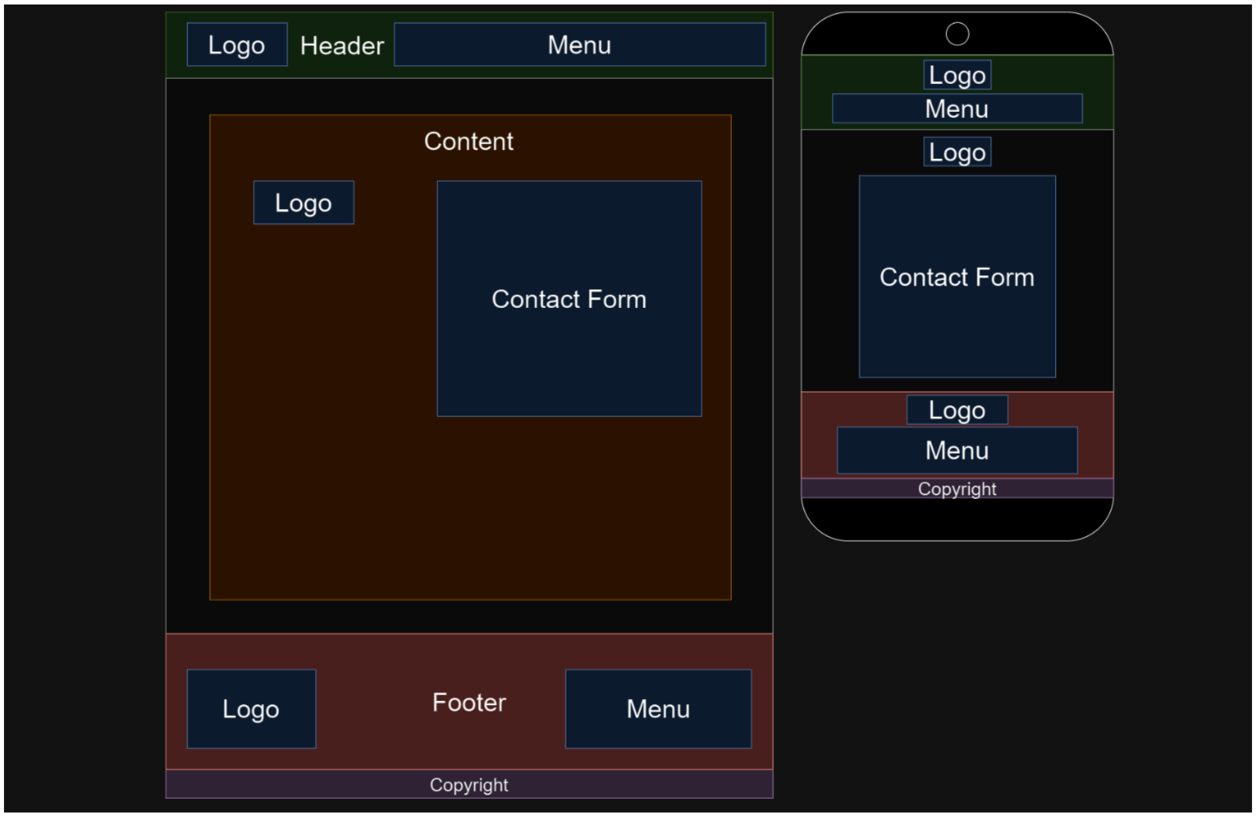


I created the wireframes using an application called draw.io. I divided the design in two parts: the desktop or large screen design and the mobile or small screen design.

In the example above We can see the wireframe of the home page. For the desktop version, I put the header menu on the very top, that includes the logo, the main menu and the login, register, logout, user icon buttons. Below, I created the Hero section that includes a short description of the site and its purpose and a call to action button. At the right side of the screen I put a large image that represents the quality of the images in the rest of the platform. Below that we can find the gallery of images with characteristic polaroid look that I created which includes a frame, the image at center-top and its title at center-bottom. At the bottom of the page we can find the footer with the logo of the site and two headings with submenus and at the very bottom the copyright and links to social media.

For the mobile version, I put the logo first, at the center-top of the page, followed by the main menu and the menu buttons, all centered. The hero section is missing its image as it would not fit the screen and it would, in my opinion, ruin the design of the page. The gallery is also set to feature one image on each row. The footer sees the logo with the motto at the top of it and the submenus bellow in the same row. The copyright section is also divided into two rows.

Zoning:



In the zoning we can clearly distinguish the main points of interest for both desktops and mobiles. The pages are split into 3 sections, top, main and bottom. With emphasis on the main section as it covers most of the screen, followed by the footer which contains important submenus and lastly, the smallest of them all, the top section as it is the first section that the user sees once they access the platform.

ii. User Flow

The user flow is described in details in the "cahier des charges" document. Here is a summary:

The homepage of the platform is designed for simplicity and ease of navigation. At the top, a discreet main menu features the platform's logo, primary navigation links, and options to register or log in. Below this, a prominent hero section (in desktop view) includes a brief description of the platform, call-to-action buttons for registration and login, and a representative image on the right edge.

The main content area features a filter in the top-right corner for sorting photos by category. Photos are displayed in a Polaroid-like layout, enlarging upon user interaction (touch on mobile or hover on desktop). Clicking or tapping a photo leads to its dedicated page.

The footer, organized into two rows and three columns, includes the platform's logo with its motto, an account section, a contact section, copyright information, and social media icons. A "scroll to top" button is conveniently placed in the bottom-right corner. Both the main menu and footer are consistent across all pages.

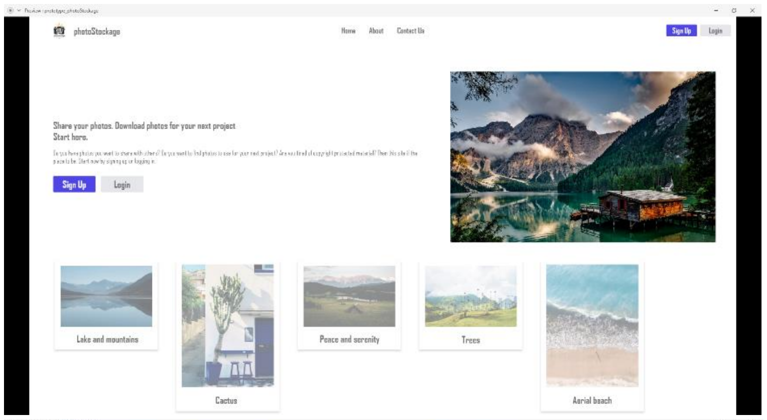
Selecting the "About" link in the main menu directs users to the About page, providing general information, cookie settings, and a legal notice. The "Contact Us" section features a visually engaging form styled as an envelope for sending messages to the site's administration.

The registration and login forms include all necessary fields for creating or accessing an account. Once logged in, the call-to-action button on the main page changes to a "Logout" option. Users can access their account dashboard by clicking on their profile icon, where they can manage their account, photos, comments, likes, downloads, and log out.

Administrators have additional management tools in their dashboard to oversee all photos, categories, comments, and user accounts, with the option to log out. Overall, the navigation and user journey are designed to be intuitive, unobtrusive, and visually appealing, ensuring a seamless and pleasant user experience.

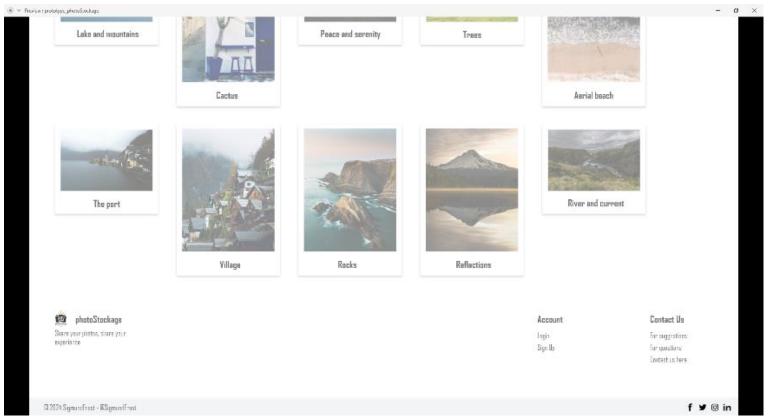
iii. Visual Design

Prototypes:



Share your photos. Download photos for your next project
Start here.

Do you have photos you want to share with others? Do you want to find photos to use for your next project? Are you tired of copyright protected material? Then this site is the place to be. Start now by signing up or logging in.

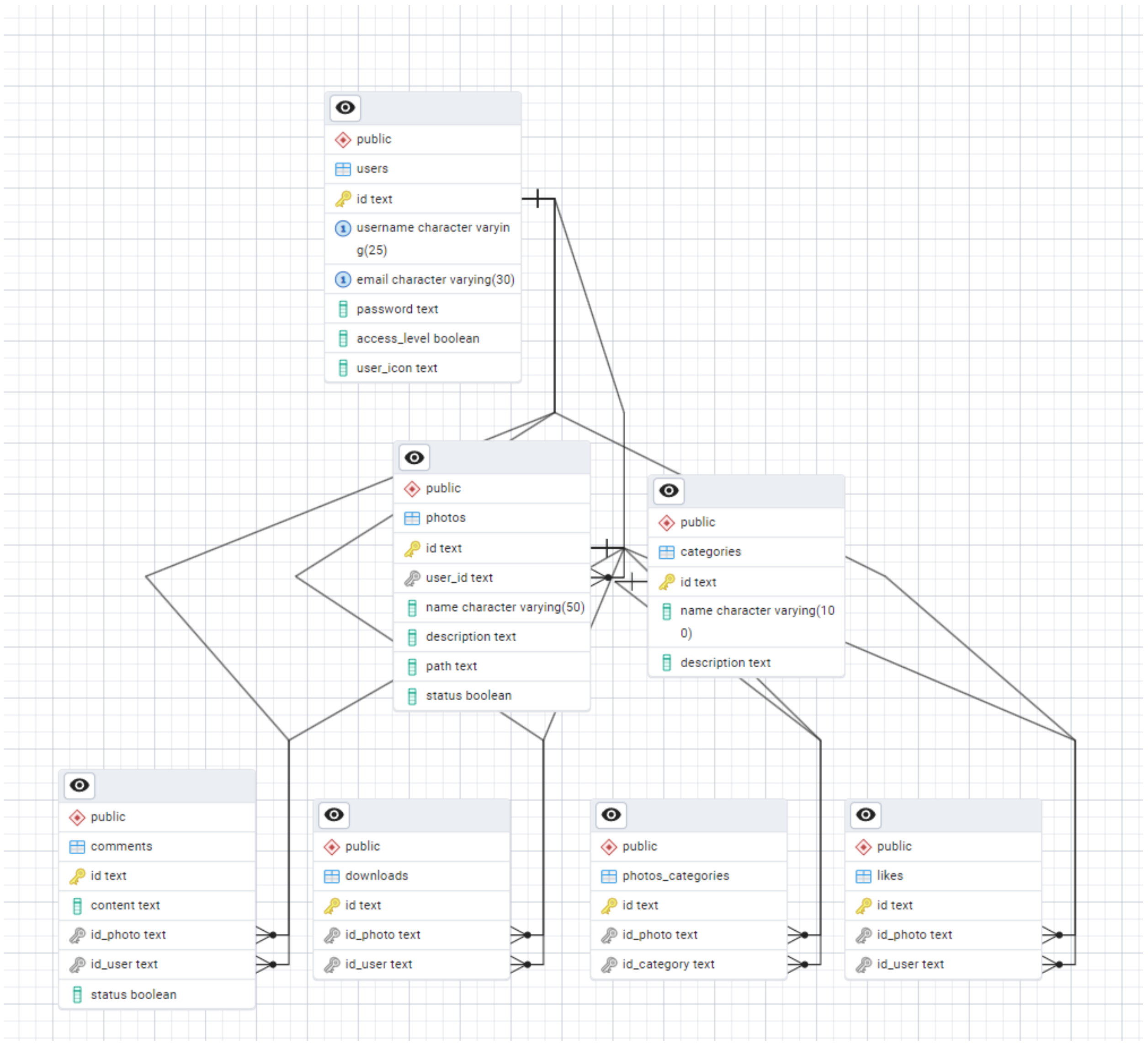


I created the prototypes using Adobe's Adobe XD, as I was specifically instructed to do, which is now obsolete. A superior solution would have been figma. Adobe XD offers a plethora of features to make the prototyping experience as close to the real development as possible. In the prototype of the home page for the desktop, displayed above, we can see a glimpse of the platform's design. The style I chose for the website is a minimalistic one, as I find it intriguing and it is a popular design choice in the recent years from my experience. The stylistic choices are described in the "cahier des charges", here is a summary: The visual direction of the website is guided by a natural color palette (#FBB328, #A88C66, #FFF8F0, #DFE0DF) generated using "mycolor.space" to complement the logo. The typography uses the Inter font family for a clean, modern look, with the logo in Star Avenue font. The design emphasizes simplicity and minimalism, reflecting the core values of "photoStockage" and ensuring a user-friendly experience.

5. Data Management

i. Database Schema

The database schema exists in the "cahier des projets" and it can not be summarized as it contains crucial information. I make the direct reference here:



The database consists of seven tables. The users table consists of six columns: an id which serves as the primary key and must be unique and non-null, of type text. The username field is a required character varying field with a maximum length of 25 characters. An email field is included as a character varying with maximum length of 30 characters, and a password field is included as text. An access_level column is included as a boolean type. Additionally, there is a user_icon column of type text.

The photos table comprises six columns: id, user_id, name, description, path, and status. The id field serves as the primary key and is of type text. User_id is a foreign key referencing the users table and is also of type text. Name is a character varying field with a maximum length of 50 characters. Description is a text field used to describe the photo. Path is a text field that defines where the image is stored. Status is a boolean field indicating the photo's current state.

The likes table (which handles user reactions to photos) consists of three columns: id, id_photo, and id_user. The id serves as the primary key and is of type text. The id_photo field is a foreign key referencing the photos table, and the id_user is a foreign key referencing the users table, both of type text.

The categories table consists of three columns: id, name, and description. The id field is the primary key and is of type text. The name field is a character varying with a maximum length of 100 characters, and description is of type text. These fields store information about the different categories that can be assigned to photos.

The photos_categories table serves as a junction table linking photos with their categories. It contains three columns: id as the primary key of type text, id_photo as a foreign key referencing the photos table, and id_category as a foreign key referencing the categories table, both also of type text.

The comments table has five columns: id as the primary key of type text, content of type text to store the comment's content, id_photo as a foreign key referencing the photos table, id_user as a foreign key referencing the users table, and a status field of type boolean to indicate the comment's state.

The downloads table represents the final table in the schema, containing three columns: id as the primary key of type text, id_photo as a foreign key referencing the photos table, and id_user as a foreign key referencing the users table, tracking which users have downloaded which photos.

ii. Data Security

The data is secured with the use of various libraries that sanitize and scrutinize the data going in and out of the database in the level of the backend as well as type safety in the frontend. If the project is ever hosted it is in the plan to have regular backups of the database, as frequent as once daily, depending on the service provider and the space availability in the hosted environment.

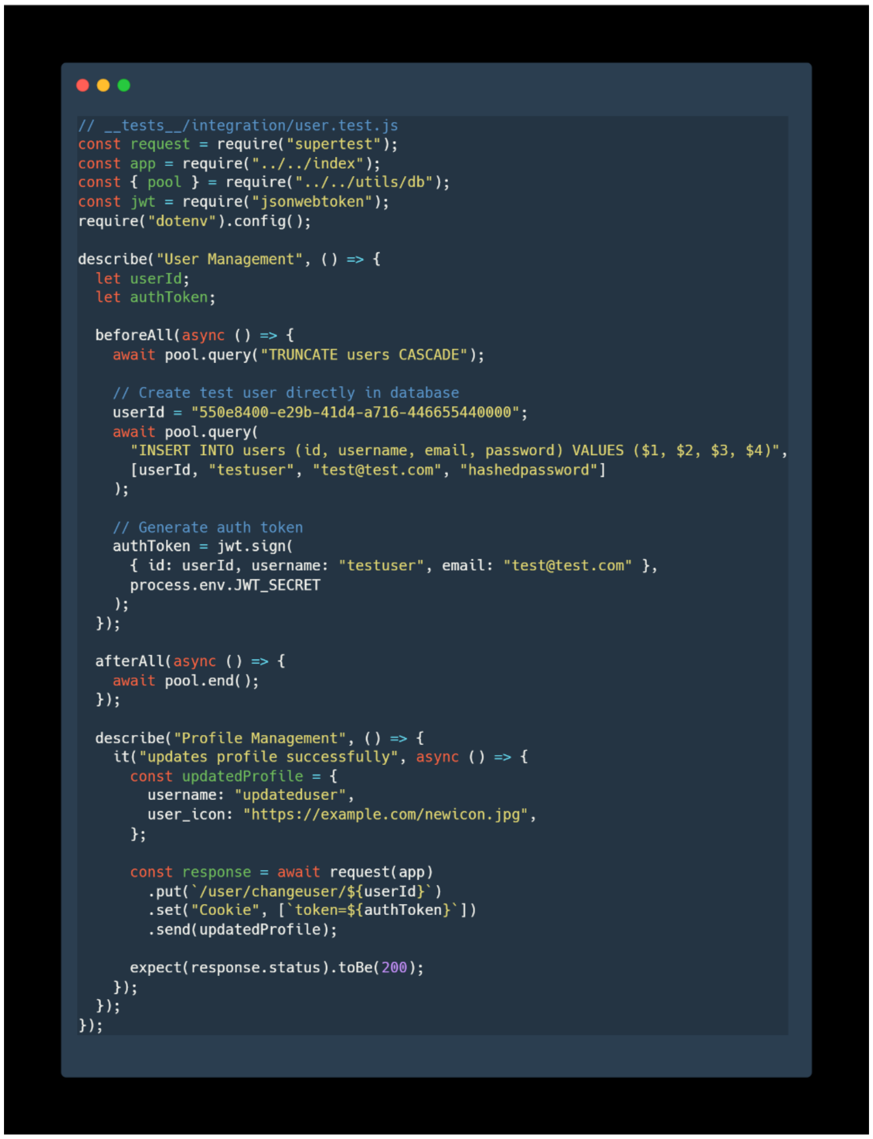
6. Testing

i. Testing Strategy

Backend testing strategy:

In the backend I have implemented unit tests with the use of Jest a library for performing various tests. In the integration folder of the tests I have created a test for each of the functionality of the backend's API. That includes tests for the categories, comments, downloads, likes, photos and users. Also a testDb file is included to perform the tests in a separate environment that would not affect the project's database.

Unit test example:

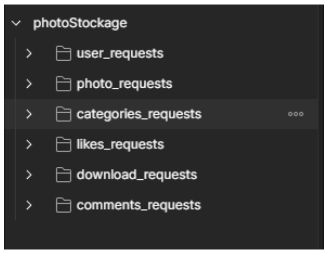


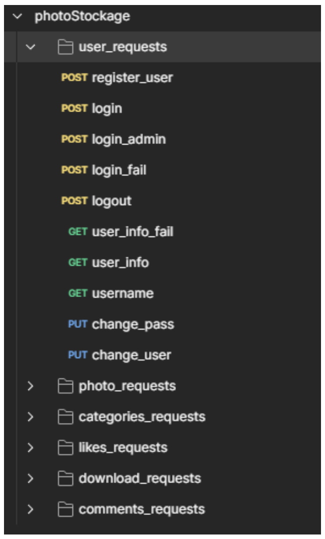
```
// __tests__/integration/user.test.js
const request = require("supertest");
const app = require("../../index");
const { pool } = require("../../utils/db");
const jwt = require("jsonwebtoken");
require("dotenv").config();

describe("User Management", () => {
  let userId;
  let authToken;

  beforeAll(async () => {
    await pool.query("TRUNCATE users CASCADE");

    // Create test user directly in database
    userId = "550e8400-e29b-41d4-a716-446655440000";
    await pool.query(
      "INSERT INTO users (id, username, email, password) VALUES ($1, $2, $3, $4)",
      [userId, "testuser", "test@test.com", "hashedpassword"]
    );

    // Generate auth token
    authToken = jwt.sign(
      { id: userId, username: "testuser", email: "test@test.com" },
      process.env.JWT_SECRET
    );
  });

  afterAll(async () => {
    await pool.end();
  });

  describe("Profile Management", () => {
    it("updates profile successfully", async () => {
      const updatedProfile = {
        username: "updateduser",
        user_icon: "https://example.com/newicon.jpg",
      };

      const response = await request(app)
        .put(`/user/changeuser/${userId}`)
        .set("Cookie", [`token=${authToken}`])
        .send(updatedProfile);

      expect(response.status).toBe(200);
    });
  });
});
```

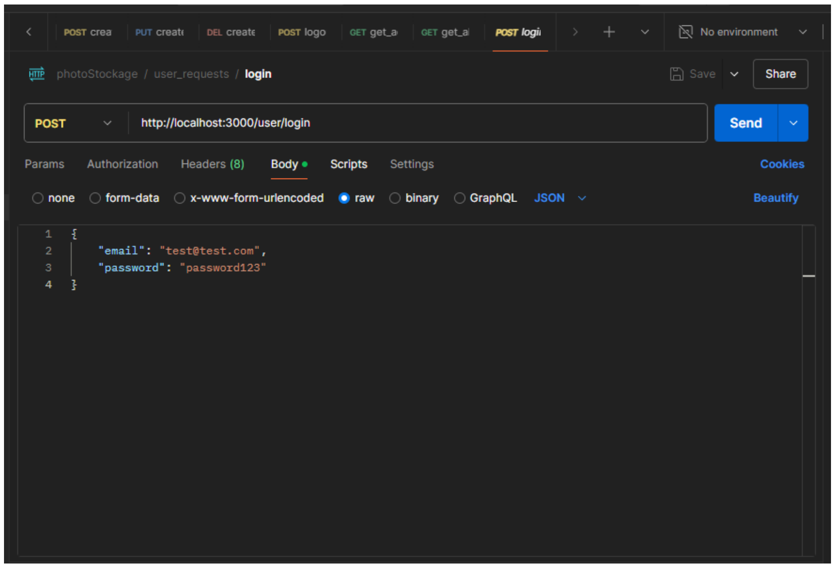
At the very top I import the necessary libraries like supertest which is a dependency package of Jest, the app from the index file, the pool from the utilities folder, the jwt library and the .env variables. The test begins by defining the functionality of it, in this case it is the user management. It creates two variables, the userId and the authToken, they are declared but not defined. The test begins by deleting the users table and cascading the effect to other tables if required. Then a new user is created and inserted into the database using the query: "INSERT INTO users (id, username, email, password) VALUES ($1, $2, $3, $4)", [userId, "testuser", "test@test.com", "hashedpassword"]. Also a jwt is stored in the authToken variable containing the necessary fields of userId, username and email and the jwt secret. After it executed the mentioned functions it tries to perform the profile management which should be successful if the username and user_icon can be updated using a PUT method and the cookie from the authToken. In the end, we expect a code status 200 which means that the actions were successfully executed.

API endpoints tests:

I performed tests meticulously to the API endpoints using the application named “Postman”. Here are some examples of my methods.







POST http://localhost:3000/user/login

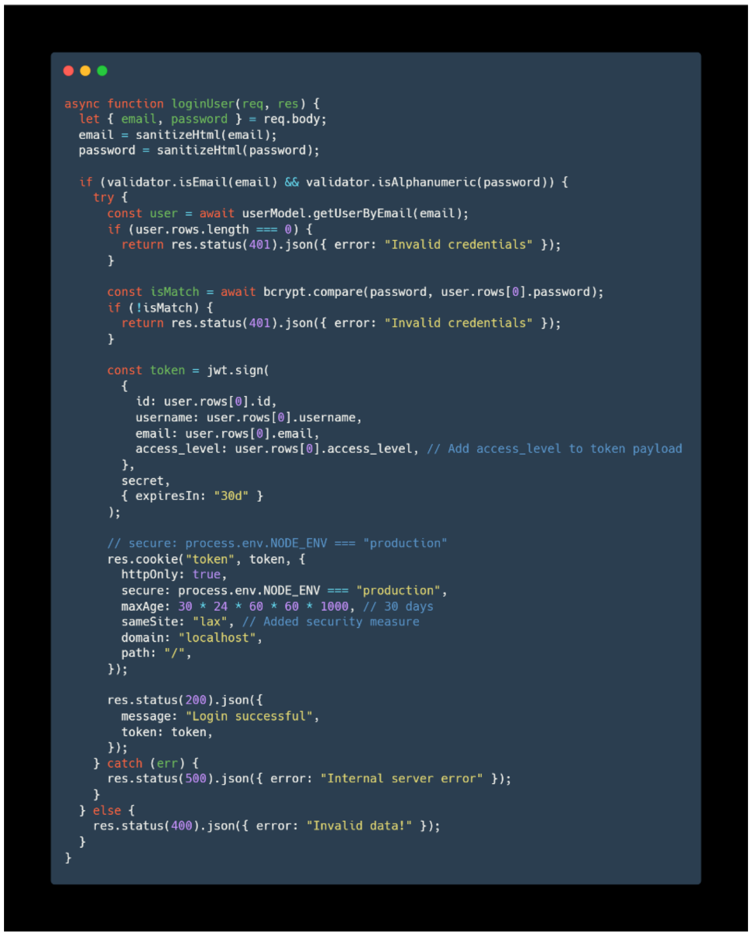
Body: [✓ raw] JSON

```
{
    "email": "test@test.com",
    "password": "password123"
}
```

I have divided in separate file the requests depending on which function I am trying to test each time. I created the folders of user, photo, category, likes, downloads and comments requests. In them I defined all the methods that can be found in the models and controllers of the backend and requests that are guaranteed to fail due to various reasons. In the example presented I attempt a login using a user and a password that do actually exist, expecting a status code of 200 containing the corresponding data.

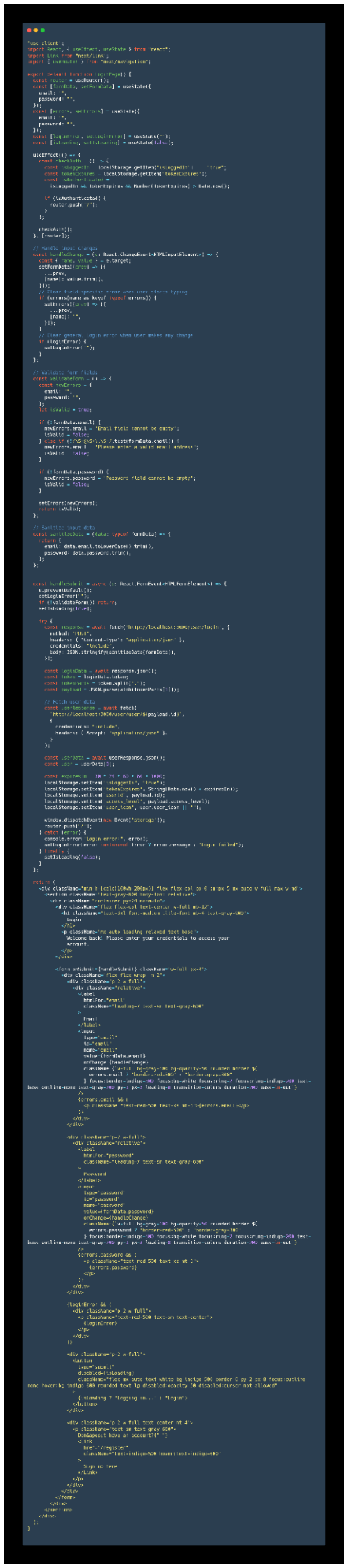
ii. Bug / Error Management

I have incorporated error handling in both the backend and the frontend as evidenced by the examples provided in previous chapter. A typical example:



```javascript
async function loginUser(req, res) {
  let { email, password } = req.body;
  email = sanitizeHtml(email);
  password = sanitizeHtml(password);

  if (validator.isEmail(email) && validator.isAlphanumeric(password)) {
    try {
      const user = await userModel.getUserByEmail(email);
      if (user.rows.length === 0) {
        return res.status(401).json({ error: "Invalid credentials" });
      }

      const isMatch = await bcrypt.compare(password, user.rows[0].password);
      if (!isMatch) {
        return res.status(401).json({ error: "Invalid credentials" });
      }

      const token = jwt.sign(
        {
          id: user.rows[0].id,
          username: user.rows[0].username,
          email: user.rows[0].email,
          access_level: user.rows[0].access_level, // Add access_level to token payload
        },
        secret,
        { expiresIn: "30d" }
      );

      // secure: process.env.NODE_ENV === "production"
      res.cookie("token", token, {
        httpOnly: true,
        secure: process.env.NODE_ENV === "production",
        maxAge: 30 * 24 * 60 * 60 * 1000, // 30 days
        sameSite: "lax", // Added security measure
        domain: "localhost",
        path: "/",
      });

      res.status(200).json({
        message: "Login successful",
        token: token,
      });
    } catch (err) {
      res.status(500).json({ error: "Internal server error" });
    }
  } else {
    res.status(400).json({ error: "Invalid data!" });
  }
}
```

When a user attempts to login, in case of wrong email address or wrong password a status 401 containing the message Invalid credentials is returned. If the jwt can not be created, a status code 500 with the error message Internal server error is returned. Finally if the user has not submitted all of the necessary data or there is a type mismatch, a status code of 400 with the error message of Invalid data is returned.



```
"use client";
import React, { useEffect, useState } from "react";
import Link from "next/link";
import { useRouter } from "next/navigation";

export default function LoginPage() {
  const router = useRouter();
  const [formData, setFormData] = useState({
    email: "",
    password: "",
  });
  const [errors, setErrors] = useState({
    email: "",
    password: "",
  });
  const [loginError, setLoginError] = useState("");
  const [isLoading, setIsLoading] = useState(false);

  useEffect(() => {
    const checkAuth = () => {
      const isLoggedIn = localStorage.getItem("isLoggedIn") === "true";
      const tokenExpires = localStorage.getItem("tokenExpires");
      const isAuthenticated =
        isLoggedIn && tokenExpires && Number(tokenExpires) > Date.now();

      if (isAuthenticated) {
        router.push("/");
      }
    };

    checkAuth();
  }, [router]);

  // Handle input changes
  const handleChange = (e: React.ChangeEvent<HTMLInputElement>) => {
    const { name, value } = e.target;
    setFormData((prev) => ({
      ...prev,
      [name]: value.trim(),
    }));
    // Clear field-specific error when user starts typing
    if (errors[name as keyof typeof errors]) {
      setErrors((prev) => ({
        ...prev,
        [name]: "",
      }));
    }
    // Clear general login error when user makes any change
    if (loginError) {
      setLoginError("");
    }
  };

  // Validate form fields
  const validateForm = () => {
    const newErrors = {
      email: "",
      password: "",
    };
    let isValid = true;

    if (!formData.email) {
      newErrors.email = "Email field cannot be empty";
      isValid = false;
    } else if (!/\S+@\S+\.\S+/.test(formData.email)) {
      newErrors.email = "Please enter a valid email address";
      isValid = false;
    }

    if (!formData.password) {
      newErrors.password = "Password field cannot be empty";
      isValid = false;
    }

    setErrors(newErrors);
    return isValid;
  };

  // Sanitize input data
  const sanitizeData = (data: typeof formData) => {
    return {
      email: data.email.toLowerCase().trim(),
      password: data.password.trim(),
    };
  };


  const handleSubmit = async (e: React.FormEvent<HTMLFormElement>) => {
    e.preventDefault();
    setLoginError("");
    if (!validateForm()) return;
    setIsLoading(true);

    try {
      const response = await fetch("http://localhost:3000/user/login", {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        credentials: "include",
        body: JSON.stringify(sanitizeData(formData)),
      });

      const loginData = await response.json();
      const token = loginData.token;
      const tokenParts = token.split(".");
      const payload = JSON.parse(atob(tokenParts[1]));

      // Fetch user data
      const userResponse = await fetch(
        `http://localhost:3000/user/user/${payload.id}`,
        {
          credentials: "include",
          headers: { Accept: "application/json" },
        }
      );

      const userData = await userResponse.json();
      const user = userData[0];

      const expiresIn = 30 * 24 * 60 * 60 * 1000;
      localStorage.setItem("isLoggedIn", "true");
      localStorage.setItem("tokenExpires", String(Date.now() + expiresIn));
      localStorage.setItem("userId", payload.id);
      localStorage.setItem("access_level", payload.access_level);
      localStorage.setItem("user_icon", user.user_icon || "");

      window.dispatchEvent(new Event("storage"));
      router.push("/");
    } catch (error) {
      console.error("Login error:", error);
      setLoginError(error instanceof Error ? error.message : "Login failed");
    } finally {
      setIsLoading(false);
    }
  };

  return (
    <div className="min-h-[calc(100vh-200px)] flex flex-col px-6 sm:px-5 mx-auto w-full max-w-md">
      <section className="text-gray-600 body-font relative">
        <div className="container py-24 mx-auto">
          <div className="flex flex-col text-center w-full mb-12">
            <h1 className="text-3xl font-medium title-font mb-4 text-gray-900">
              Login
            </h1>
            <p className="mx-auto leading-relaxed text-base">
              Welcome back! Please enter your credentials to access your
              account.
            </p>
          </div>

          <form onSubmit={handleSubmit} className="w-full px-4">
            <div className="flex flex-wrap -m-2">
              <div className="p-2 w-full">
                <div className="relative">
                  <label
                    htmlFor="email"
                    className="leading-7 text-sm text-gray-600"
                  >
                    Email
                  </label>
                  <input
                    type="email"
                    id="email"
                    name="email"
                    value={formData.email}
                    onChange={handleChange}
                    className={`w-full bg-gray-100 bg-opacity-50 rounded border ${
                      errors.email ? "border-red-500" : "border-gray-300"
                    } focus:border-indigo-500 focus:bg-white focus:ring-2 focus:ring-indigo-200 text-
base outline-none text-gray-700 py-1 px-3 leading-8 transition-colors duration-200 ease-in-out`}
                  />
                  {errors.email && (
                    <p className="text-red-500 text-xs mt-1">{errors.email}</p>
                  )}
                </div>
              </div>

              <div className="p-2 w-full">
                <div className="relative">
                  <label
                    htmlFor="password"
                    className="leading-7 text-sm text-gray-600"
                  >
                    Password
                  </label>
                  <input
                    type="password"
                    id="password"
                    name="password"
                    value={formData.password}
                    onChange={handleChange}
                    className={`w-full bg-gray-100 bg-opacity-50 rounded border ${
                      errors.password ? "border-red-500" : "border-gray-300"
                    } focus:border-indigo-500 focus:bg-white focus:ring-2 focus:ring-indigo-200 text-
base outline-none text-gray-700 py-1 px-3 leading-8 transition-colors duration-200 ease-in-out`}
                  />
                  {errors.password && (
                    <p className="text-red-500 text-xs mt-1">
                      {errors.password}
                    </p>
                  )}
                </div>
              </div>

              {loginError && (
                <div className="p-2 w-full">
                  <p className="text-red-500 text-sm text-center">
                    {loginError}
                  </p>
                </div>
              )}

              <div className="p-2 w-full">
                <button
                  type="submit"
                  disabled={isLoading}
                  className="flex mx-auto text-white bg-indigo-500 border-0 py-2 px-8 focus:outline-
none hover:bg-indigo-600 rounded text-lg disabled:opacity-50 disabled:cursor-not-allowed"
                >
                  {isLoading ? "Logging in..." : "Login"}
                </button>
              </div>

              <div className="p-2 w-full text-center mt-4">
                <p className="text-sm text-gray-600">
                  Don&apos;t have an account?{" "}
                  <Link
                    href="/register"
                    className="text-indigo-500 hover:text-indigo-600"
                  >
                    Sign up here
                  </Link>
                </p>
              </div>
            </div>
          </form>
        </div>
      </section>
    </div>
  );
}
```

In the frontend, if there is an error message or a warning message returned from the request to the backend it is displayed with the use of variables saved in the components state for the login page (following the login example). Also, errors in the frontend, like missing data or type mismatch are handled similarly and the user gets instant feedback by displaying said error on their screen.

7. Deployment and Maintenance

i. Deployment Process

If deployment is required there are a few paths that I could follow to perform it. The options are a mix of personal choice and budget restrictions. If the budget allows it, platforms for hosting the frontend like vercel should be chosen. For the backend a VPS that would run it in a docker container would be ideal. As for the database Supabase could be chosen as it offers a robust solution. In my case, with the budget limitations I am facing the best option would be to use one VPS that will host all three, frontend, backend and the database.

In more details, the process, would be as follows: I would first have to export my database if I want to keep my data, if not the backend contains files with the description of the schema and some "demo / dummy data". Then I would need to create a docker container for the backend. It is very important to exclude the node_modules folders and files from the docker container as it would be a waste of space, same for the frontend. For the docker container, it is important that the project has a standardized structure, it includes the package.json, package-lock.json, the .dockerignore and the Dockerfile as well as all of the projects files inside an src folder. The dockerignore file should include the node_modules folder git files the dockerignore file itself, the Dockerfile and error / bug log files. The Dockerfile should include the node version, a working directory, a copy of the package.json and the lock files, a command to run "npm install", a copy command for copying the files from the project, an expose command for our desired port and finally a command to start our project. Similarly, for the frontend, the same files would be required with the addition of extra commands and a file that specifies a build command, the environment type and whether we want to restart on changes or not. The last file is optional.

There is a less popular option of shipping the project to a traditional hosting service with major limitations to how many concurrent connections it could support due to less resources available.

ii. Maintenance

For the part of the maintenance, it would have to be split in multiple sections. A daily section which would involve checks for the system's health, possible performance issues, error logs and security logs.

On a weekly basis, usage analysis to determine if the resource are allocated properly and database health checks. On a less frequent basis, dependency updates as it could pose a security risk if they are not properly and regularly updated, code reviews and refactoring to improve performance and analysis of users' feedback and addition of features.

8. Challenges and Solutions

During the development of the project, I faced numerous challenges, mostly due to lack of prior experience, especially concerning the backend. Using various resources, I managed to overcome the obstacles and produce a product that I am proud of.

One of the early challenges I faced was the phase of prototyping. As I do not have a background in graphic design or web design, the usage of Adobe XD was cumbersome. The solution to this problem came from watching tutorial videos on platforms like youtube and searching for tips and guidance in forums dedicated to either Adobe XD designing or web designing in general.

Another challenge I faced in regard to the frontend was the reconversion of the site's style. As it is the first project in which I used tailwind, I was unaware that the framework is mobile first. I developed numerous pages and components on the desktop monitor first, only to find out later that I had to re-write my styles by starting from a mobile screen and moving "up" to larger screens. The solution to the problem was re-reading the styles and trying to re-imagine it as mobile first.

The biggest challenge was the management of the jwt token. As this was the first project I used jwt and http only cookies, I did not know how to handle the jwt in the frontend, the cookie in the backend and how to implement the authentication functions. During the training, we developed a number of projects that used the jwt which helped a lot in its implementation.

9. Documentation and Conclusion

i. Documentation

I have included documentation and implementation instructions for the project. The variable and function names are intuitive so they can help other developers use the project if they want. There is documentation for both the frontend and the backend. Here is an example of the documentation of the API.

PhotoStockage Backend Documentation

Table of Contents

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

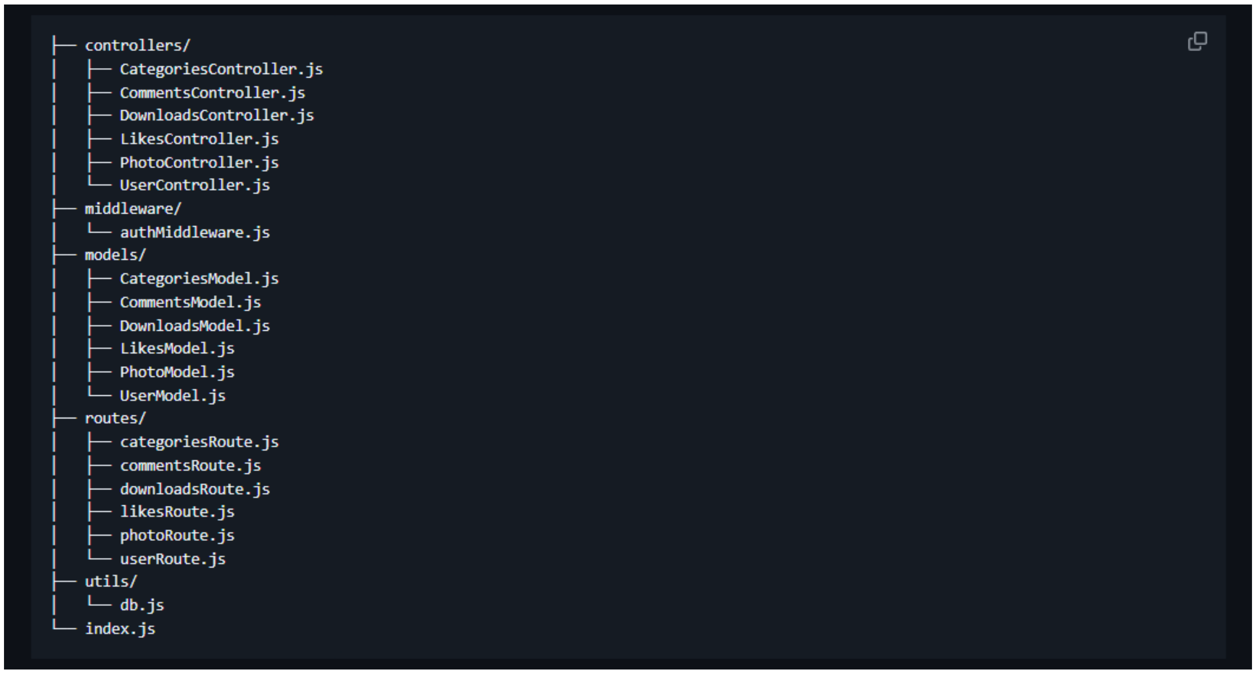
- Node.js
- Express.js
- PostgreSQL
- JWT for authentication
- Jest for testing

System Architecture

Core Components

1. **Server Configuration (index.js)**
    - Express application setup
    - Middleware configuration
    - Route registration
    - CORS and security settings
2. **Middleware Layer (middleware/)**
    - Authentication validation
    - Admin access control
    - Rate limiting
    - Security headers (Helmet)
3. **Controllers Layer (controllers/)**
    - Business logic implementation
    - Request handling
    - Response formatting
    - Error management
4. **Models Layer (models/)**
    - Database interactions
    - Data validation
    - Query execution
5. **Routes Layer (routes/)**
    - API endpoint definitions
    - Route protection
    - Request routing

Directory 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
```

Getting Started

Prerequisites

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

Installation

1. Clone the repository
2. Install dependencies:

npm install

3. Set up environment variables:

DATABASE_URL=postgresql://username:password@localhost:5432/database

JWT_SECRET=your_jwt_secret

NODE_ENV=development

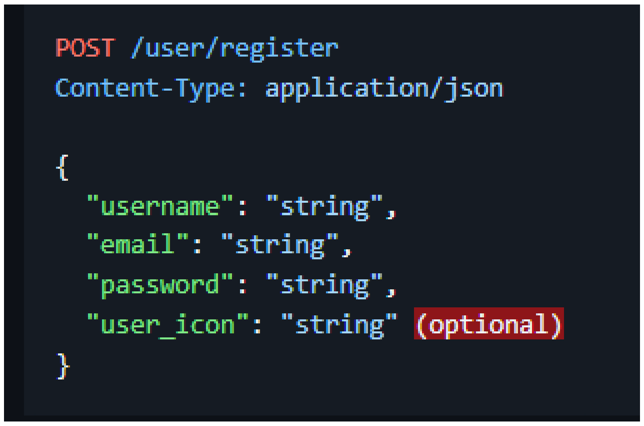
4. Start the server:

npm start

API Reference

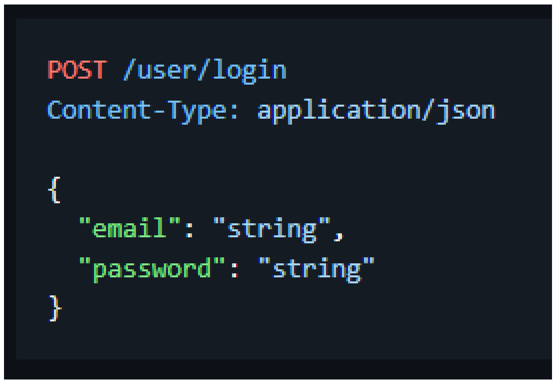
User Management

Register User



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

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

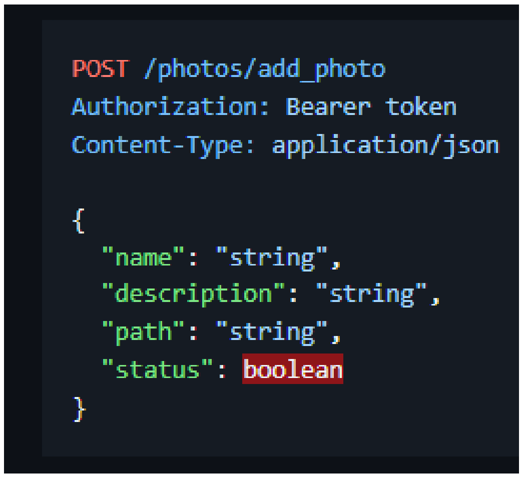
Login



```
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
}
```

Get Photos

GET /photos/photos

Categories

Create Category (Admin only)

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

{
  "name": "string",
  "description": "string"
}
```

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

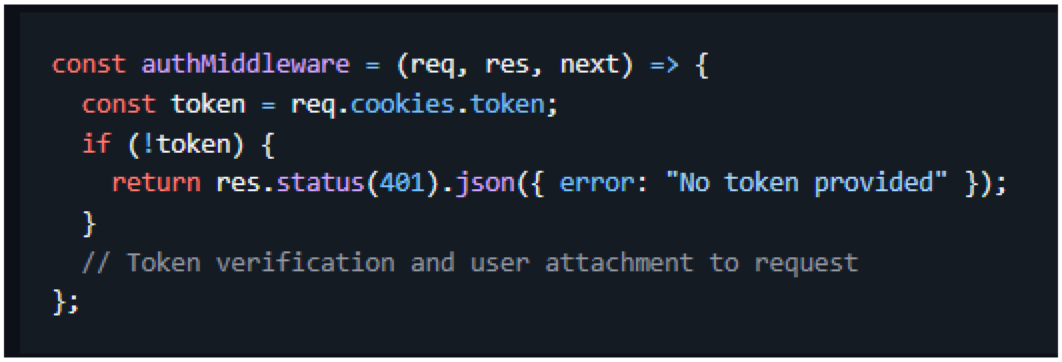
```
CREATE TABLE photos (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(id),
  name VARCHAR(255) NOT NULL,
  description TEXT,
  path VARCHAR(255),
  status BOOLEAN DEFAULT true,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

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

Middleware Protection



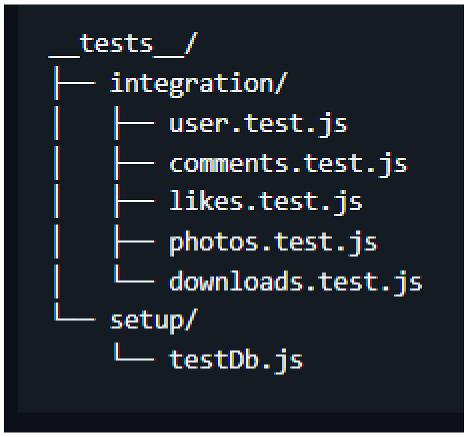
```javascript
const authMiddleware = (req, res, next) => {
  const token = req.cookies.token;
  if (!token) {
    return res.status(401).json({ error: "No token provided" });
  }
  // Token verification and user attachment to request
};
```

Security Features

1. **Input Sanitization**
   - HTML sanitization using sanitize-html
   - Data validation with validator
2. **Rate Limiting**
   - 100 requests per 15 minutes per IP
3. **Security Headers**
   - Helmet middleware implementation
   - XSS protection
   - Content Security Policy
   - Frame protection
4. **Password Security**
   - Bcrypt hashing
   - Salt rounds: 10

Testing

Test Structure



```
__tests__/
├── integration/
│   ├── user.test.js
│   ├── comments.test.js
│   ├── likes.test.js
│   ├── photos.test.js
│   └── downloads.test.js
└── setup/
    └── testDb.js
```

Running Tests

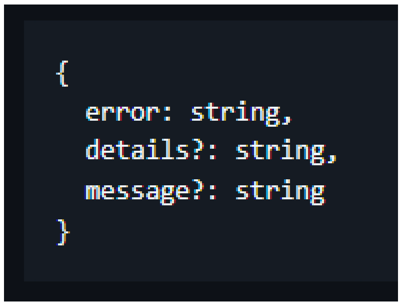
npm test

Coverage Areas

- User authentication
- CRUD operations
- Permission validation
- Error handling
- Input validation

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

ii. Conclusion

During the development of this project, I improved a lot as a developer. It was a great opportunity to learn new technologies like express, nextjs and tailwind which have helped me create projects faster and more efficiently. I had the chance to learn more about the security of projects, work on my backend skills and apply my knowledge on the technologies I was already familiar with. I was also free to utilize my imagination and create a site that I envisioned, with my personal style and to express myself. In the future I will revisit the project to improve on its code, make it more performant, apply new guidelines for security and make stylistic updates.

10. Annexes

Desktop interface:

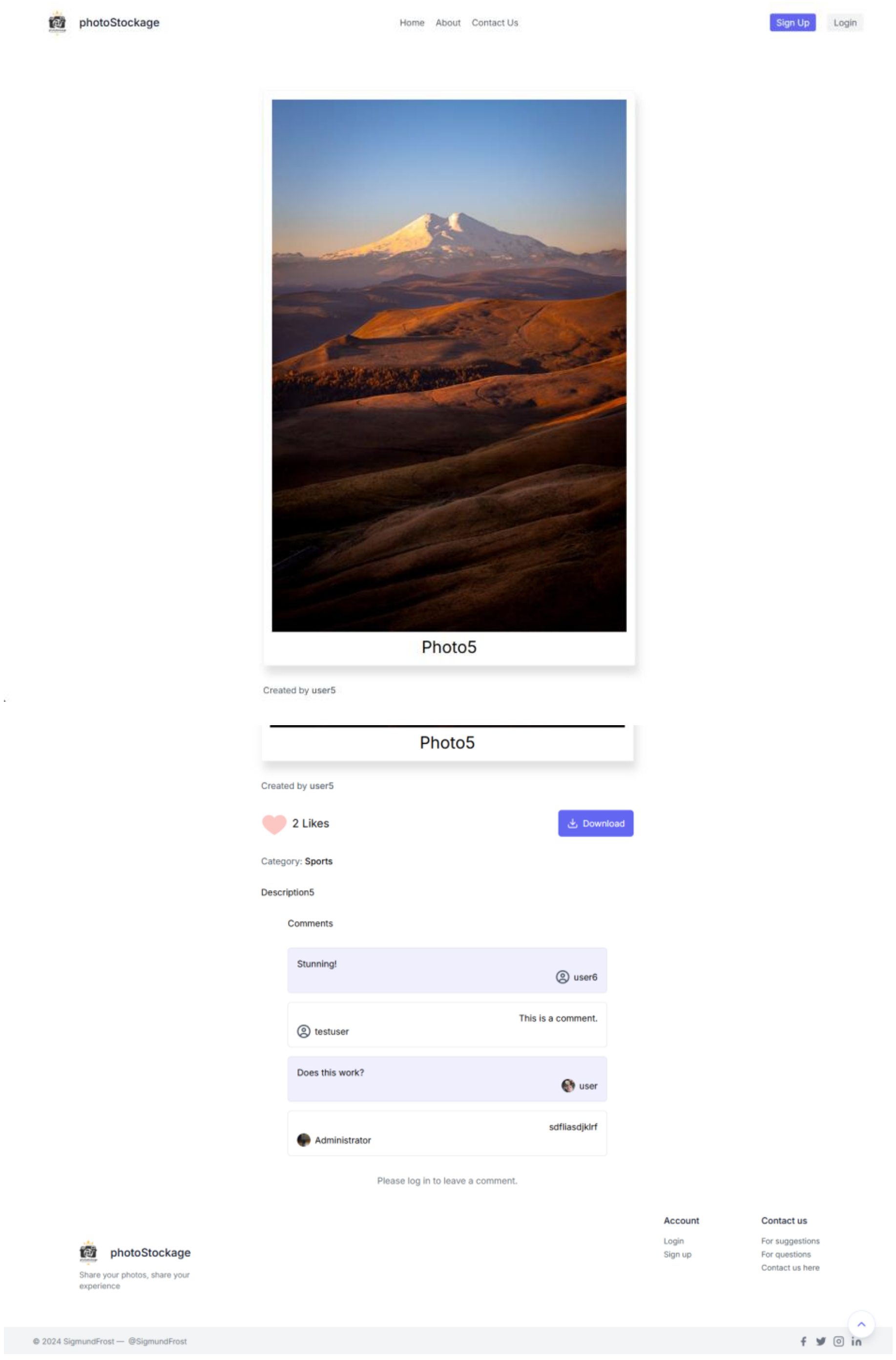


Photo5

Created by user5

Photo5

Created by user5

2 Likes

Category: **Sports**

Description5

Comments

Stunning!
user6

This is a comment.
testuser

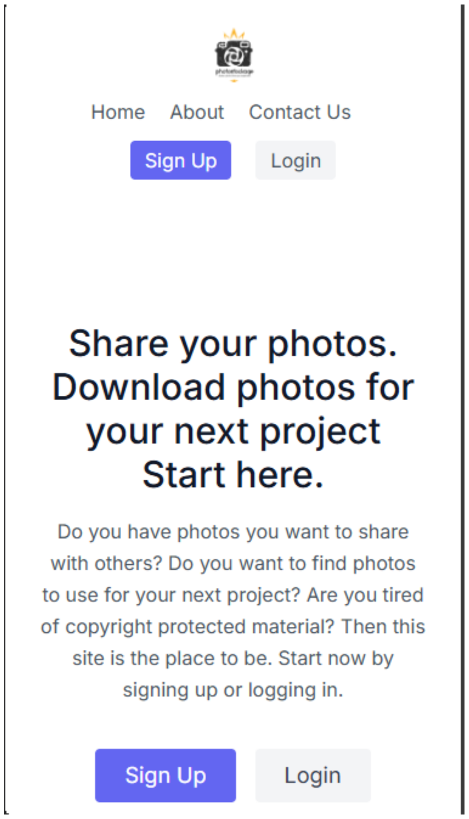
Does this work?
user

sdfliasdjklrf
Administrator

Please log in to leave a comment.

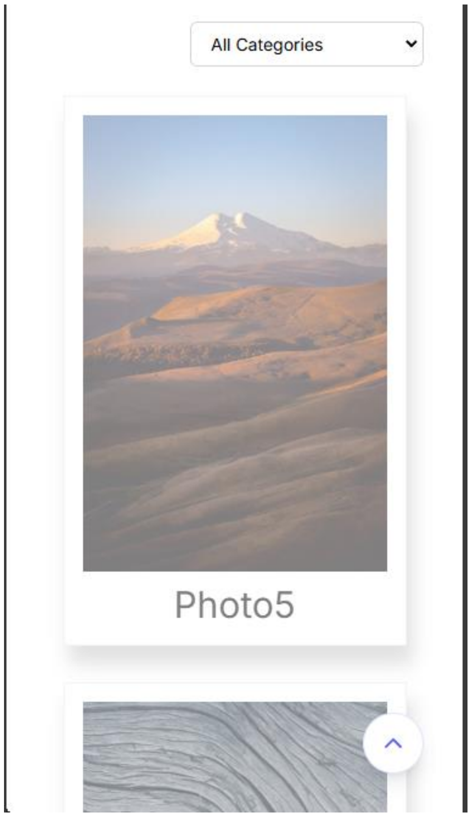
© 2024 SigmundFrost — @SigmundFrost

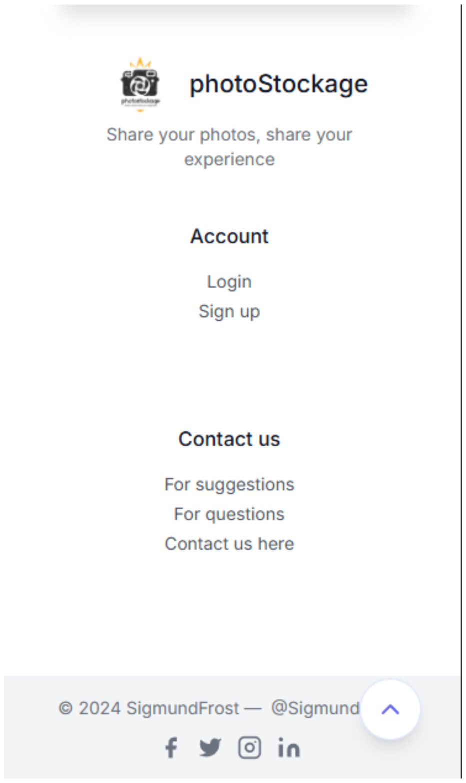
Mobile interface:



Share your photos. Download photos for your next project Start here.

Do you have photos you want to share with others? Do you want to find photos to use for your next project? Are you tired of copyright protected material? Then this site is the place to be. Start now by signing up or logging in.





photoStockage

Share your photos, share your experience

Account

Login
Sign up

Contact us

For suggestions
For questions
Contact us here

© 2024 SigmundFrost — @Sigmund

Github repository:

This is the link to the project's github repository that includes the entire project, images, screenshot and other sources used in the making of it: https://github.com/EsFrost/m2i_projet . The project itself is hosted as a demo at: https://photostockage.vercel.app .