Dossier de projet

“photoStockage”

Projet réalisé dans le cadre de la présentation au

Titre Professionnel Développeur Web et Web Mobile

présenté par

Sotirios NATSIOS

M2i Scribtel

Index

1. Introduction 1
   i. Objectif du document 1
   ii. Aperçu du projet 1
2. Portée et architecture du projet 2
   i. Portée 2
   ii. Aperçu de l'architecture 5
3. Implémentation du code 6
   i. Backend 6
   ii. Frontend 13
4. Design et expérience utilisateur 17
   i. Wireframes 17
   ii. Flux utilisateur 18
   iii. Design visuel 19
5. Gestion des données 20
   i. Schéma de la base de données 20
   ii. Sécurité des données 22
6. Tests 22
   i. Stratégie de test 22
   ii. Gestion des bugs / erreurs 25
7. Déploiement et maintenance 28
   i. Processus de déploiement 28
   ii. Maintenance 28
8. Défis et solutions 29
9. Documentation et conclusion 29
   i. Documentation 29
   ii. Conclusion 37
10. Annexes 37

1. Introduction

i. Objectif du document

Ce document sert de guide complet pour le projet "photoStockage", offrant un aperçu détaillé des technologies frontend et backend utilisées. Il décrit les décisions que j'ai prises pour garantir la réussite de la création d'une application web CRUD (Create, Read, Update, Delete) entièrement fonctionnelle.

Le document commence par expliquer le but du projet, fournissant une compréhension claire de ses objectifs et buts. Il explore ensuite des wireframes et des prototypes détaillés, offrant des représentations visuelles de la structure et du flux de l'application. Des captures d'écran de segments de code clés sont également incluses, accompagnées d'explications approfondies pour assurer clarté et compréhension.

Des diagrammes d'expérience utilisateur et des explications sur les décisions de design visuel sont intégrés, offrant un aperçu du processus de réflexion derrière l'interface et les interactions utilisateur de l'application. De plus, le document inclut des schémas de base de données complets, décrivant la structure et les relations au sein de la base de données.

Les stratégies de test sont discutées, avec des exemples de code pour illustrer les méthodologies utilisées pour garantir la fiabilité et les performances de l'application. Un schéma détaillé pour le déploiement et la maintenance future du projet est également fourni, assurant une durabilité et une évolutivité à long terme.

Le document se conclut par une réflexion sur les défis rencontrés lors de la création et du développement du projet, ainsi que les solutions mises en œuvre pour les surmonter. Enfin, il inclut une documentation concise des aspects critiques tels que les procédures d'installation, les endpoints de l'API backend, et d'autres composants essentiels, assurant un processus de développement fluide et efficace.

ii. Aperçu du projet

"photoStockage" est un site web hybride qui combine l'essence des réseaux sociaux avec la fonctionnalité d'une plateforme de partage d'images. Au cœur du projet, il sert d'espace pour les utilisateurs afin de télécharger, télécharger et explorer des images tout en puisant dans une vaste collection de contenu visuel. La plateforme est conçue pour être entièrement réactive, garantissant une expérience utilisateur fluide et intuitive sur tous les appareils, des téléphones mobiles aux grands écrans de bureau.

L'audience cible principale de "photoStockage" se compose de professionnels qui nécessitent des images de haute qualité, sans droits d'auteur, pour divers projets créatifs et commerciaux, à condition qu'ils respectent les directives légales et éthiques. Une audience secondaire inclut les artistes, designers, et autres créatifs qui utilisent la plateforme comme source d'inspiration, un hub de réseautage, ou un espace de mentorat et de collaboration. En favorisant une communauté engageante et supportive, la plateforme comble le fossé entre les ressources d'images professionnelles et l'interaction sociale.

L'une des caractéristiques distinctives de "photoStockage" est son accent sur l'interaction utilisateur et la gestion de contenu. Les utilisateurs peuvent créer, éditer et supprimer des comptes tout en partageant leurs propres images et les histoires personnelles derrière elles. La plateforme permet aux membres d'interagir avec le contenu via des likes, des commentaires, et des collections sauvegardées. De plus, les utilisateurs peuvent gérer leurs interactions en éditant ou en supprimant des commentaires et en contrôlant les données qu'ils partagent. Un système de support dédié est également en place, permettant une communication directe avec l'administration du site pour toute assistance ou demande.

Le but de "photoStockage" est de créer un environnement inclusif et dynamique où créativité et accessibilité vont de pair. En fournissant un dépôt légalement solide d'images de haute qualité, il permet aux professionnels et aux créatifs d'améliorer leur travail sans se soucier des restrictions de droits d'auteur. En même temps, la plateforme encourage l'expression artistique, l'échange d'idées et la croissance impulsée par la communauté. Que les utilisateurs recherchent l'actif visuel parfait, cherchent une motivation artistique, ou visent à se connecter avec des individus partageant les mêmes idées, "photoStockage" offre une expérience polyvalente et enrichissante.

2. Portée et architecture du projet

i. Portée

Dans "photoStockage", j'ai personnellement développé les fonctionnalités essentielles qui constituent la colonne vertébrale de la plateforme. Au cœur du site, les utilisateurs disposent de la capacité de créer un compte personnel, de personnaliser leur profil et de gérer leur propre contenu avec facilité. Chaque utilisateur a un contrôle total sur ses images téléchargées, y compris la possibilité d'ajouter, de modifier et de supprimer du contenu selon les besoins. La plateforme est conçue non seulement pour partager des visuels, mais aussi pour favoriser l'engagement au sein de la communauté, permettant aux utilisateurs d'interagir avec les images via des likes et des commentaires, de sauvegarder leurs photos préférées dans des collections personnalisées, et d'explorer une gamme diversifiée de contenus créatifs.

Pour garantir des opérations fluides et sécurisées, j'ai développé une API backend qui impose des limites de requêtes, prévenant les abus et maintenant la stabilité du système. La sécurité a été une priorité tout au long du développement, ce qui m'a conduit à intégrer diverses mesures de protection pour protéger les données et interactions des utilisateurs. En plus des mécanismes d'authentification et de validation des données, j'ai mis en

œuvre des protocoles de sécurité pour filtrer et assainir les entrées des utilisateurs, assurant ainsi un environnement sûr pour tous les membres de la plateforme.

Sur le frontend, j'ai structuré l'interface pour offrir une expérience utilisateur fluide et intuitive. En utilisant Next.js, j'ai tiré parti de ses optimisations intégrées, telles que la gestion dynamique des images et la navigation efficace des pages, pour améliorer les performances sur différents appareils. L'utilisation des composants Image et Link de Next.js m'a permis d'optimiser la livraison des médias et d'assurer une expérience de navigation fluide, même avec des images haute résolution. En adhérant aux meilleures pratiques de développement modernes, j'ai visé à créer une plateforme qui équilibre performance, utilisabilité et évolutivité, rendant "photoStockage" accessible à un large public, des utilisateurs occasionnels aux professionnels de l'industrie.

En définissant l'audience cible de "photoStockage", je me suis directement inspiré de ma propre expérience en tant que développeur web et créatif. J'ai personnellement rencontré les défis auxquels sont confrontés les individus dans ces deux rôles, ce qui m'a aidé à façonner les fonctionnalités principales de la plateforme pour répondre à leurs besoins spécifiques. En tant que développeur, j'avais souvent besoin d'images de haute qualité, sans droits d'auteur, pour des projets sans restrictions légales, tandis qu'en tant que créatif, je cherchais de l'inspiration, de l'interaction et des opportunités de connexion avec d'autres dans le domaine. Ces expériences m'ont permis de structurer "photoStockage" de manière à servir efficacement les deux groupes.

L'audience principale se compose de professionnels, y compris les développeurs web, les designers, les marketeurs et les créateurs de contenu, qui ont besoin d'accéder à des images haute résolution, sans droits d'auteur, pour des projets commerciaux et personnels. Pour améliorer leur expérience, j'ai implémenté une fonctionnalité de recherche qui permet aux utilisateurs de trouver des images par nom ou par catégorie, simplifiant ainsi leur flux de travail et facilitant la localisation de l'actif visuel parfait. En offrant un dépôt d'images gratuit et accessible, "photoStockage" élimine le besoin d'abonnements coûteux à des banques d'images, permettant aux professionnels de se concentrer sur leurs projets sans se soucier des restrictions de licence.

L'audience secondaire comprend des créatifs tels que des photographes, des artistes et des designers qui utilisent la plateforme pour présenter leur travail, chercher de l'inspiration et interagir avec des individus partageant les mêmes idées. Pour eux, "photoStockage" est plus qu'un simple dépôt d'images - c'est un espace d'interaction et de construction communautaire. J'ai développé des fonctionnalités qui encouragent l'engagement, y compris un bouton de like permettant aux utilisateurs d'apprécier et de soutenir le travail des autres, ainsi qu'un système de commentaires où ils peuvent partager des retours et initier des discussions. De plus, j'ai conçu les profils utilisateurs pour inclure des informations de contact par email, permettant une communication directe entre les membres. Cette fonctionnalité favorise les collaborations potentielles, les opportunités de mentorat et le réseautage professionnel.

Pour faciliter davantage la communication, j'ai personnellement conçu un formulaire de contact qui permet aux utilisateurs de contacter directement l'administration de la plateforme. Qu'ils aient des questions, besoin d'assistance ou souhaitent signaler un problème, cette fonctionnalité assure que les utilisateurs peuvent recevoir un soutien efficace. En intégrant ces éléments interactifs, "photoStockage" sert non seulement d'outil pratique

pour les professionnels, mais aussi de hub créatif dynamique où les utilisateurs peuvent se connecter, partager et grandir.

Pour le développement de "photoStockage", j'ai choisi des technologies qui allient efficacité et facilité d'utilisation. Pour le backend, j'ai implémenté Node.js avec le framework Express, car il offre un environnement léger et simple qui améliore l'expérience de développement. Pour le frontend, j'ai utilisé Next.js, exploitant son App Router pour optimiser la navigation et les performances tout en maintenant une base de code structurée et évolutive. Étant donné mon expérience préalable avec React, Next.js était un choix naturel qui permettait une intégration fluide des meilleures pratiques.

Pour la gestion des données, j'ai opté pour PostgreSQL, une base de données relationnelle open-source avec un excellent support communautaire et une grande évolutivité. Ce choix garantit que la plateforme peut gérer efficacement le stockage des données tout en restant adaptable à une future expansion si nécessaire. Bien que la plateforme ne soit pas encore hébergée, elle a été conçue pour être déployable sur un VPS ou des services d'hébergement traditionnels, avec certaines limitations sur le nombre d'utilisateurs simultanés en fonction de l'environnement d'hébergement.

Pour garantir la sécurité et les performances, j'ai suivi les meilleures pratiques recommandées par les frameworks choisis. De plus, j'ai mis en œuvre des mesures de sécurité à l'aide de diverses bibliothèques pour protéger les données des utilisateurs et maintenir l'intégrité de la plateforme. En combinant ces technologies, j'ai créé une base solide, évolutive et conviviale pour les développeurs pour "photoStockage", assurant une expérience fluide pour tous les utilisateurs.

En tant que projet indépendant sans financement externe, la principale limitation de "photoStockage" est financière. Puisqu'il n'y a pas de sponsors ou de sources de revenus soutenant son développement, l'évolutivité et la durabilité à long terme de la plateforme sont directement liées aux ressources disponibles. Les coûts d'hébergement, en particulier, jouent un rôle crucial dans la détermination de la capacité des utilisateurs. Bien que le projet soit conçu pour être déployable à la fois sur des VPS et des services d'hébergement traditionnels, le nombre d'utilisateurs qu'il peut supporter dépendra en fin de compte de l'infrastructure d'hébergement disponible au moment du déploiement.

D'un point de vue juridique, "photoStockage" inclut une clause de non-responsabilité dans la section "À propos" qui décrit les responsabilités des utilisateurs et garantit le respect des réglementations sur le droit d'auteur et le contenu. Cette clause de non-responsabilité stipule explicitement que les utilisateurs doivent se conformer à la loi lors du téléchargement et du partage de contenu, et elle dégage l'administration de toute responsabilité en cas de mauvaise utilisation de la plateforme.

Pour maintenir un environnement sûr et approprié, un système de modération est en place pour traiter les éventuelles violations. Si un contenu inapproprié ou illégal est détecté par l'administration, il est immédiatement supprimé. De plus, les utilisateurs ont la possibilité de signaler du contenu, et en fonction de la gravité de la

violation, des mesures nécessaires sont prises, y compris la notification des autorités compétentes lorsque cela est requis.

En termes de confidentialité des données, "photoStockage" suit une politique de rétention minimale des données. La plateforme ne collecte ni ne stocke de données utilisateur inutiles, garantissant une approche axée sur la confidentialité. Les seules données utilisateur conservées incluent l'email, le nom d'utilisateur et le dernier horodatage de connexion, qui sont stockés en toute sécurité dans la base de données. Ces informations sont essentielles à des fins d'authentification, en particulier pour la gestion sécurisée de l'authentification basée sur JWT à l'aide de cookies HTTPS. En limitant le stockage des données à ce qui est strictement nécessaire, la plateforme améliore la confidentialité tout en maintenant les fonctionnalités principales.

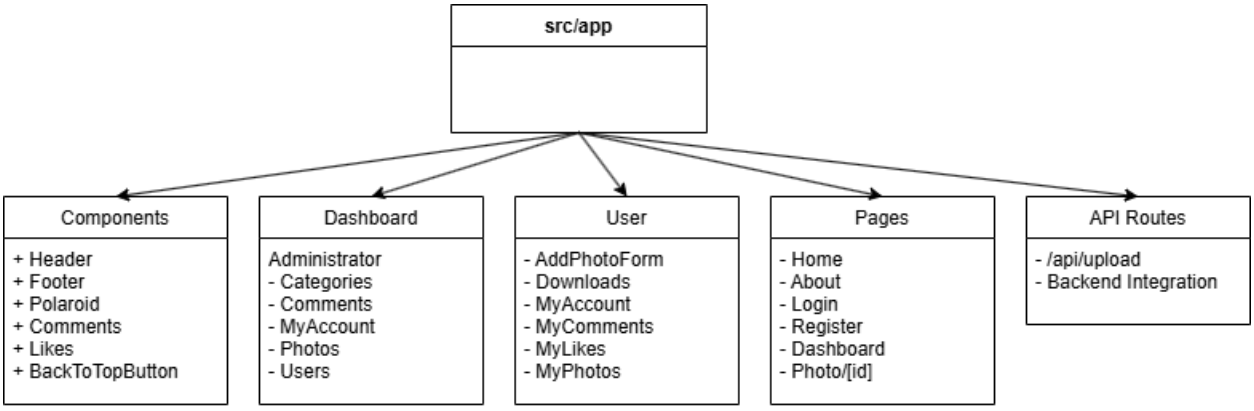
ii. Aperçu de l'architecture

Diagramme de flux backend :

Le point d'entrée du backend est le fichier index. Il est écrit en utilisant le framework Express, en exploitant une structure de modèle MVC (sans la partie vue car le frontend est séparé). Lors d'une requête frontend, le backend via l'index appelle la route appropriée, qui à son tour appelle une fonction spécifique du contrôleur

qui appellera finalement le modèle approprié pour récupérer ou envoyer des données à la base de données. Les opérations qui envoient des données à la base de données sont protégées par un middleware qui vérifie si l'utilisateur est authentifié et, si nécessaire, si l'utilisateur est l'administrateur pour certaines routes protégées.

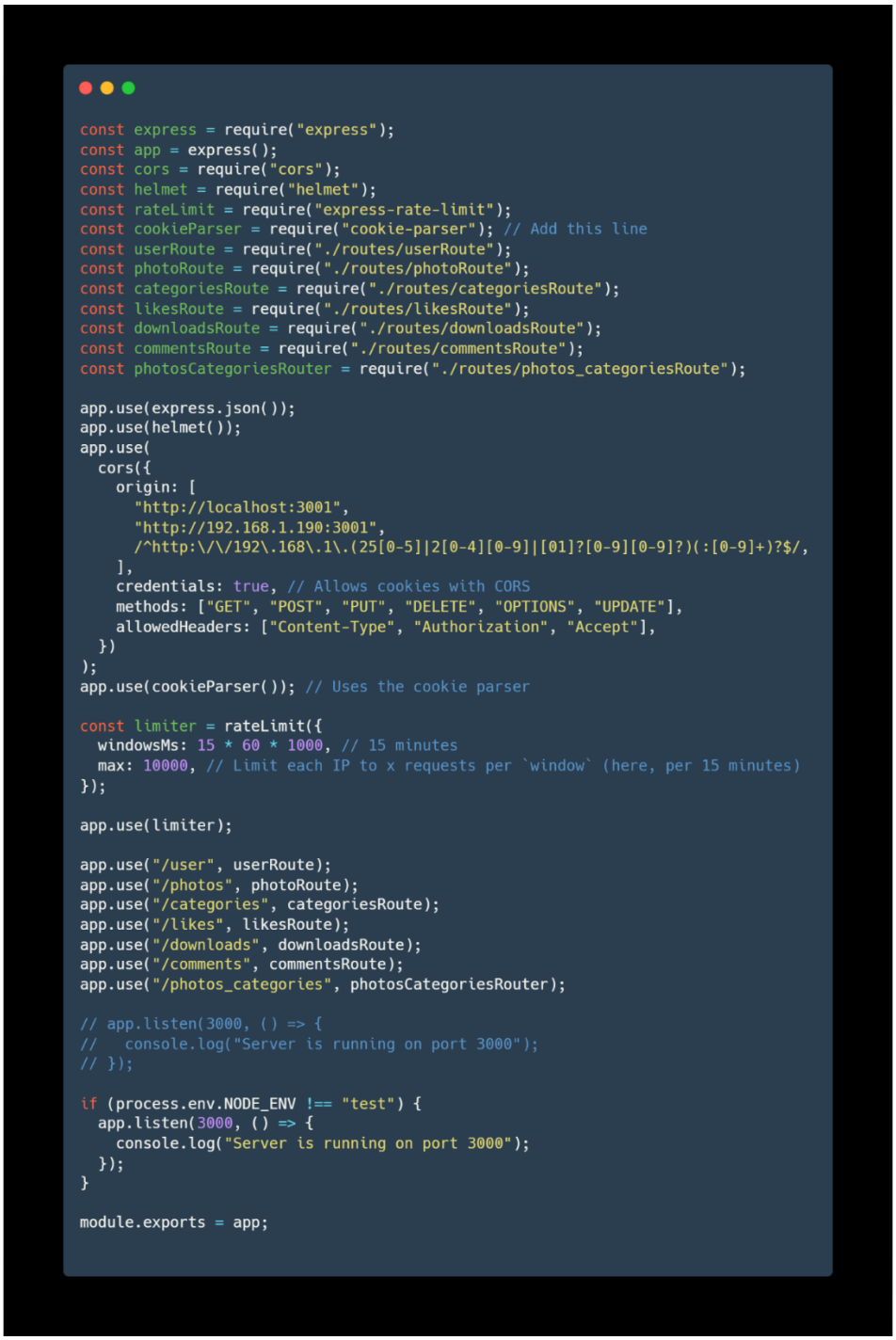
Diagramme de flux frontend :



Le frontend est divisé en sections. Les pages principales du frontend comprennent la page d'accueil, une page à propos, une page de connexion pour les utilisateurs non connectés, une page d'inscription, une page de tableau de bord pour les utilisateurs connectés et les pages individuelles des photos. Il inclut les composants d'en-tête et de pied de page présents tout au long de l'expérience. Le composant polaroid, que j'ai créé pour imiter le style polaroid bien connu des photos avec des fonctionnalités supplémentaires comme l'agrandissement lors du survol ou du toucher pour les écrans mobiles et des ombres intéressantes pour donner une profondeur de champ. Il y a également les composants de commentaires et de likes, ainsi qu'un composant de retour en haut de base qui apparaît sur chaque page lorsque l'utilisateur a fait défiler une certaine partie de la page. J'ai divisé le tableau de bord en deux parties, la partie administration et la partie utilisateur, avec des responsabilités distinctes et séparées. L'administrateur a la possibilité de voir toutes les photos, les utilisateurs, les commentaires et de les supprimer si nécessaire. L'utilisateur, quant à lui, a la possibilité de télécharger des images, de les éditer, de les supprimer, de même pour les commentaires, de voir ses likes et téléchargements et, bien sûr, de gérer son compte. Chaque action nécessitant une fonction de base de données implique un appel API au backend.

3. Implémentation du code

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;
```

Comme mentionné précédemment, le point d'entrée du backend est le fichier index. En haut du fichier, toutes les bibliothèques nécessaires sont appelées. L'application utilise Express, Helmet pour la sécurité, CORS pour restreindre l'accès depuis des sources inconnues, un parseur de cookies pour le décodage du JWT stocké dans un cookie HTTPS uniquement, un limiteur pour limiter les requêtes afin d'éviter les attaques DOS, suivi des routes et enfin la définition du port sur lequel le backend écoute les connexions actives.

La section CORS accepte les connexions locales provenant uniquement du port 3001 utilisé par le frontend, l'adresse locale 192.168.1.190 qui, au moment du codage, était l'adresse attribuée à mon téléphone mobile à des fins de test, ainsi que toute adresse locale dans la plage 192.168.1.x/16 (0-255) pour des tests supplémentaires. Les identifiants sont requis, ce qui signifie que pour certaines routes, un JWT est nécessaire. Les méthodes autorisées sont les méthodes GET pour récupérer des données de la base de données et les autres méthodes pour effectuer des manipulations sur la base de données, toutes protégées par l'utilisation du middleware.

Le limiteur d'application restreint à 10 000 requêtes toutes les 15 minutes pour protéger contre une utilisation malveillante de la plateforme.

Le routeur de l'application appelle toutes les routes nécessaires pour que la plateforme fonctionne.

Enfin, nous écoutons sur un port spécifié pour les connexions tant que l'environnement est défini sur autre chose que "test".

J'ai choisi la structure MVC car elle m'aide à garder mes fichiers et dossiers plus organisés de manière plus propre et, plus important encore, elle sépare les préoccupations des fichiers, ce qui est une pratique qui maintient tout en ordre.

Connexion à la base de données :

```
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
}
```

Pour la connexion à la base de données, j'utilise la bibliothèque "pg" qui est une bibliothèque utilisée pour établir des connexions à une base de données PostgreSQL. À partir de celle-ci, j'appelle un objet appelé Pool pour gérer, ouvrir et fermer les connexions à la base de données en utilisant l'URL provenant du fichier .env (pour des raisons de sécurité, il ne sera pas partagé ici). Le fichier inclut également une fonction de test de connexion utilisée pour déterminer si l'interaction avec la base de données est réussie ou non. Enfin, le pool est exporté pour être utilisé par d'autres fichiers.

Modèles :

```
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,
};
```

Les modèles sont des exemples de fonctions qui interagissent directement avec la base de données. Par exemple, un modèle de catégorie pourrait inclure des fonctions pour récupérer, créer, mettre à jour et supprimer des catégories. Ces fonctions sont ensuite exportées pour être importées et utilisées par les contrôleurs.

Contrôleurs :

```
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,
};
```

Les contrôleurs sont responsables de la logique métier et de la gestion des requêtes. Ils utilisent les fonctions des modèles pour interagir avec la base de données. Par exemple, un contrôleur de catégorie pourrait inclure des fonctions pour vérifier si l'utilisateur est un administrateur, valider les données d'entrée, et appeler les fonctions du modèle pour effectuer les opérations nécessaires.

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;
```

Les routes définissent les points d'entrée pour les requêtes API. Elles utilisent les contrôleurs pour gérer les requêtes et peuvent inclure des middlewares pour l'authentification et l'autorisation. Par exemple, une route de catégorie pourrait inclure des définitions pour les méthodes GET, POST, PUT et DELETE, avec des middlewares pour vérifier si l'utilisateur est authentifié et s'il est un administrateur.

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 };
```

Les middlewares sont utilisés pour des tâches spécifiques comme l'authentification et l'autorisation. Par exemple, un middleware d'authentification pourrait vérifier si un token JWT est présent dans les cookies de la requête, le décoder et attacher les informations de l'utilisateur à la requête. Un middleware d'autorisation pourrait vérifier si l'utilisateur a le niveau d'accès requis pour effectuer une action spécifique.

ii. Frontend

Pour le frontend, j'ai utilisé Next.js car je suis plus expérimenté avec React qu'avec Angular, et le routeur d'application de Next.js offre une meilleure expérience de développement que le routeur de React. Next.js offre également des fonctionnalités comme les composants Link et Image, le rendu côté serveur et la mise en cache des données par défaut, ce qui facilite et rend plus intuitif la gestion du contenu.

Les bibliothèques, également appelées packages, utilisées dans le projet peuvent être trouvées dans les fichiers package.json et package-lock.json. J'ai également choisi d'utiliser TypeScript car il permet l'utilisation de types pour JavaScript, ce qui rend le code plus facile à lire et sûr en termes de types. De plus, j'utilise Tailwind CSS tout au long du projet car il offre plus de flexibilité que le CSS vanilla et évite les pièges de l'utilisation de Bootstrap.

La partie frontend du projet est divisée en pages, composants et utilitaires. Le point d'entrée du frontend est la page d'accueil.

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

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

```tsx
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>
  );
}
```

Dans la page d'accueil, j'importe le composant Polaroid et le composant Hero, qui sont des composants que j'ai créés pour ce projet. Ensuite, la page est rendue en suivant la structure : composant Hero -> un élément HTML principal avec une hauteur minimale définie pour couvrir toute la vue -> le composant Polaroid en tant qu'enfant. Ils sont tous enveloppés dans un élément vide également connu sous le nom de Fragment React.

Le fichier de mise en page définit la structure des pages (il est maintenu tout au long du projet) de la manière suivante : en haut, j'importe les méthodes, fonctions, bibliothèques nécessaires comme le type Metadata de Next, la famille de polices Inter de Google, le fichier CSS global qui contient uniquement les imports pour Tailwind, les composants Header et Footer, les composants CookieConsent et BackToTheTop. Ensuite, je définis le sous-ensemble de ma police et mes métadonnées pour le projet, comme le titre par exemple. Puis, je définis la structure à suivre lors du rendu d'une page : l'élément HTML avec la propriété lang définie sur anglais. L'élément body qui utilisera la police Inter, un affichage flex et la direction de colonne pour flex. Le composant Header est rendu en haut, suivi des enfants de la page, puis le composant Footer et enfin les composants CookieConsent et BackToTheTop.

```
"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>
  );
};
```

Composant Polaroid :

Le composant Polaroid utilise le rendu côté client car il implique des hooks, et les actions de hooks ne peuvent se produire que côté client. React et les hooks useEffect et useState sont importés directement depuis React, ainsi que les composants Image et Link de Next.js. Enfin, les interfaces pour Photo et Category, telles que définies par moi, sont également importées. Dans le composant Polaroid lui-même, nous définissons les images de type tableau de Photo, la catégorie sélectionnée de type chaîne, les catégories de type tableau de Category. Tous utilisent le hook useState pour que le composant "se souvienne" de tout changement effectué pendant la navigation de l'utilisateur. Un useEffect est appelé en raison de la nécessité de récupérer des données depuis la base de données. En particulier, les photos actives et les catégories sont retournées, suivies d'une vérification des erreurs. Ensuite, la valeur des images est définie sur le résultat de la requête de récupération des images, et la valeur des catégories est définie sur le résultat de la requête de récupération des catégories. J'appelle la fonction pour l'exécuter et je définis un tableau de dépendances vide pour que la requête soit exécutée uniquement une fois par actualisation de la page. Je définis également une fonction qui change l'ID de la catégorie

envoyé à la requête API en conséquence. Ensuite, je demande un rendu de mon composant. J'enveloppe tout le composant dans une div au lieu d'un Fragment React car je souhaite le conserver en tant qu'élément distinct. Je définis ensuite une div pour avoir un affichage flex avec justification du contenu à droite (fin), un remplissage sur l'axe des x de 8 rems et une marge de 6 rems pour le bas. Ensuite, je définis ma liste pour les catégories. Lorsque une autre option est sélectionnée, la fonction pour changer les catégories est appelée et une nouvelle requête API est effectuée pour récupérer les données correspondantes. Après l'élément de liste, je crée ma galerie d'images. Une clé est requise pour que Next.js puisse distinguer les éléments, les rendant uniques afin qu'il sache lesquels pourraient nécessiter un nouveau rendu en cas de changement. L'image est enveloppée dans un lien. L'image elle-même a sa qualité complète en utilisant la propriété sizes, mais sa largeur est limitée à 250px. Il y a également un attribut alt qui sert aux personnes malvoyantes ainsi qu'aux problèmes de connexion, car il est rendu sous forme de texte qui peut être lu à haute voix par les lecteurs d'écran pour les personnes ayant des problèmes de vue.

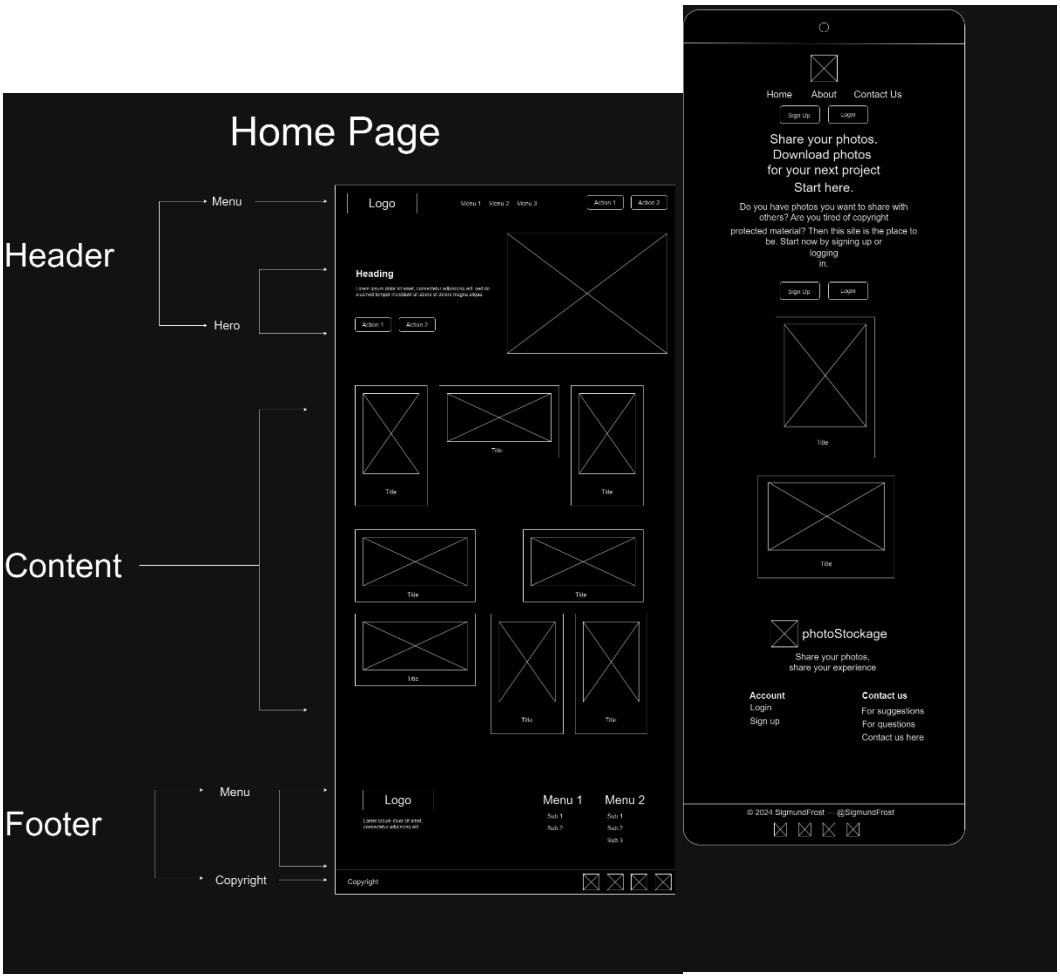
Interfaces et Types :

```
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;
}
```

Des interfaces Photo, Comment, User et Category sont définies car elles sont nécessaires pour les composants qui les impliquent. Par exemple, l'interface Photo est définie par un ID de type chaîne requis, un user_id de type chaîne, un nom de type chaîne, une description de type chaîne, un chemin de type chaîne, un statut de type booléen, une catégorie optionnelle et un objet utilisateur optionnel contenant un nom d'utilisateur de type chaîne, un email de type chaîne, un ID optionnel de type chaîne et une icône utilisateur optionnelle de type chaîne. Cela décrit les données attendues à récupérer depuis la base de données et, plus important encore, la structure des données à envoyer à la base de données pour éviter les violations et les erreurs.

4. Design et expérience utilisateur

i. Wireframes



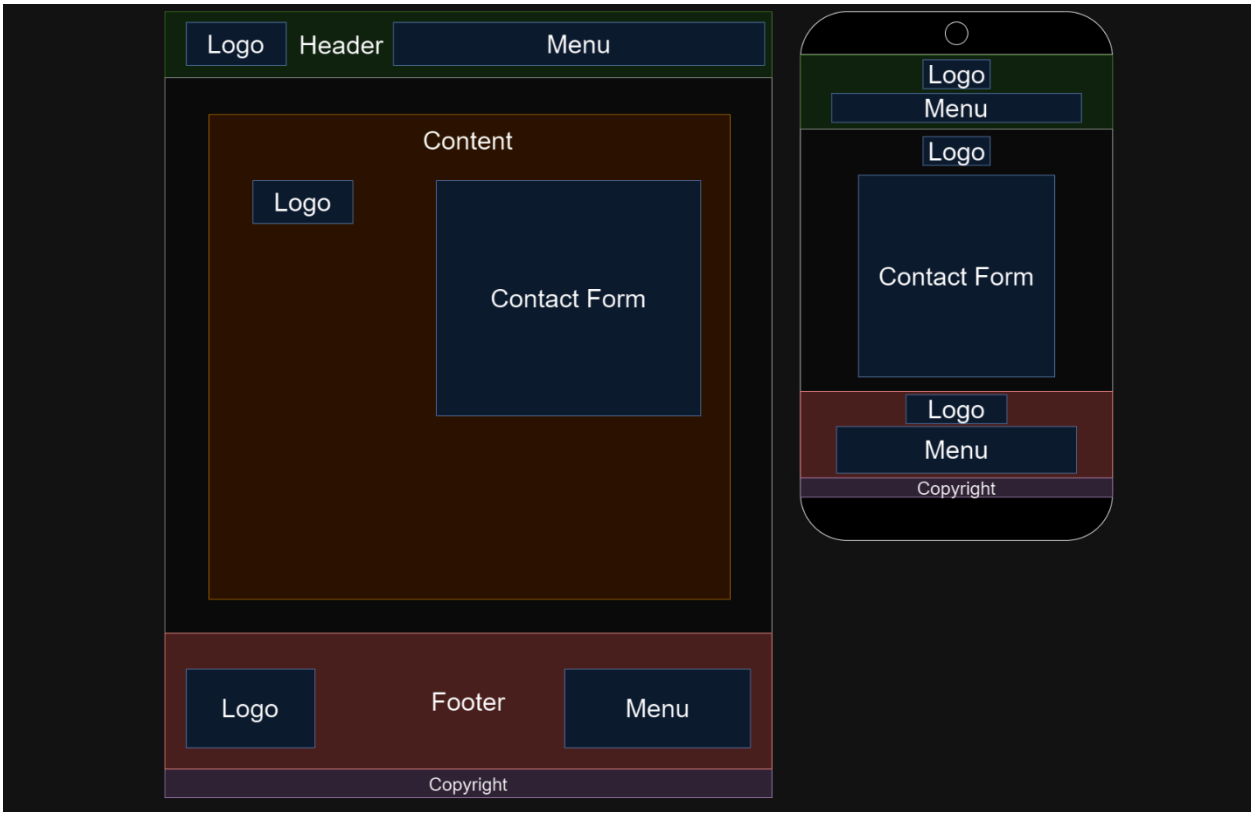
J'ai créé les wireframes en utilisant une application appelée draw.io. J'ai divisé le design en deux parties : le design pour les écrans de bureau ou grands écrans et le design pour les écrans mobiles ou petits écrans.

Dans l'exemple ci-dessus, nous pouvons voir le wireframe de la page d'accueil. Pour la version desktop, j'ai placé le menu principal tout en haut, qui inclut le logo, le menu principal et les boutons de connexion, d'inscription, de déconnexion et l'icône utilisateur. En dessous, j'ai créé la section Hero qui inclut une brève description du site et de son objectif ainsi qu'un bouton d'appel à l'action. Sur le côté droit de l'écran, j'ai placé une grande image représentant la qualité des images sur le reste de la plateforme. En dessous, nous trouvons la galerie d'images avec un look caractéristique de polaroid que j'ai créé, qui inclut un cadre, l'image au centre-haut et son titre au centre-bas. En bas de la page, nous trouvons le pied de page avec le logo du site et deux titres avec des sous-menus, et tout en bas, les mentions légales et les liens vers les réseaux sociaux.

Pour la version mobile, j'ai placé le logo en premier, au centre-haut de la page, suivi du menu principal et des boutons de menu, tous centrés. La section Hero est dépourvue de son image car elle ne s'adapterait pas à l'écran

et, à mon avis, cela ruinerait le design de la page. La galerie est également configurée pour afficher une image par ligne. Le pied de page voit le logo avec le slogan en haut, et les sous-menus en dessous dans la même ligne. La section des mentions légales est également divisée en deux lignes.

Zoning :



Dans le zoning, nous pouvons clairement distinguer les points principaux d'intérêt pour les bureaux et les mobiles. Les pages sont divisées en 3 sections : haut, principal et bas. Avec un accent sur la section principale car elle couvre la majeure partie de l'écran, suivie du pied de page qui contient des sous-menus importants et enfin, la plus petite de toutes, la section haut car c'est la première section que l'utilisateur voit lorsqu'il accède à la plateforme.

ii. Flux utilisateur

Le flux utilisateur est décrit en détail dans le document "cahier des charges". Voici un résumé :

La page d'accueil de la plateforme est conçue pour la simplicité et la facilité de navigation. En haut, un menu principal discret présente le logo de la plateforme, les liens de navigation principaux et les options pour s'inscrire ou se connecter. En dessous, une section Hero (en vue desktop) inclut une brève description de la plateforme, des boutons d'appel à l'action pour l'inscription et la connexion, et une image représentative sur le bord droit.

La zone de contenu principal présente un filtre en haut à droite pour trier les photos par catégorie. Les photos sont affichées dans une mise en page de type Polaroid, s'agrandissant lors de l'interaction de l'utilisateur (toucher sur mobile ou survol sur desktop). Le clic ou le toucher sur une photo mène à sa page dédiée.

Le pied de page, organisé en deux lignes et trois colonnes, inclut le logo de la plateforme avec son slogan, une section compte, une section contact, des informations de copyright et des icônes de réseaux sociaux. Un bouton "retour en haut" est pratiquement placé dans le coin inférieur droit. Le menu principal et le pied de page sont cohérents sur toutes les pages.

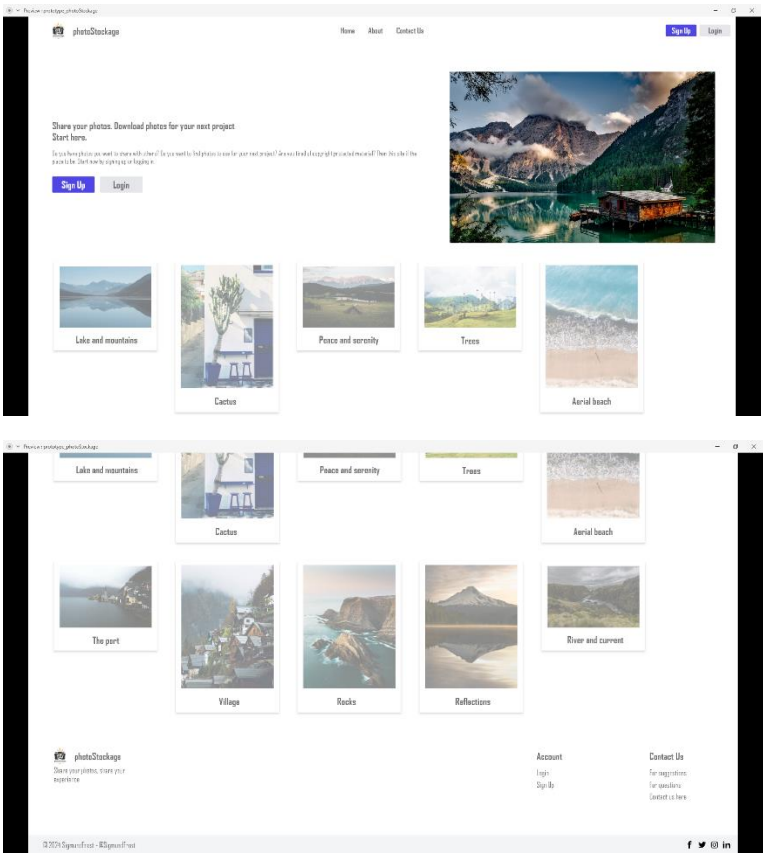
La sélection du lien "À propos" dans le menu principal dirige les utilisateurs vers la page À propos, fournissant des informations générales, des paramètres de cookies et un avis juridique. La section "Contactez-nous" présente un formulaire visuellement engageant stylisé comme une enveloppe pour envoyer des messages à l'administration du site.

Les formulaires d'inscription et de connexion incluent tous les champs nécessaires pour créer ou accéder à un compte. Une fois connecté, le bouton d'appel à l'action sur la page principale change en une option "Déconnexion". Les utilisateurs peuvent accéder à leur tableau de bord de compte en cliquant sur leur icône de profil, où ils peuvent gérer leur compte, leurs photos, commentaires, likes, téléchargements et se déconnecter.

Les administrateurs disposent d'outils de gestion supplémentaires dans leur tableau de bord pour superviser toutes les photos, catégories, commentaires et comptes utilisateurs, avec l'option de se déconnecter. Dans l'ensemble, la navigation et le parcours utilisateur sont conçus pour être intuitifs, peu intrusifs et visuellement attrayants, garantissant une expérience utilisateur fluide et agréable.

iii. Design visuel

Prototypes :

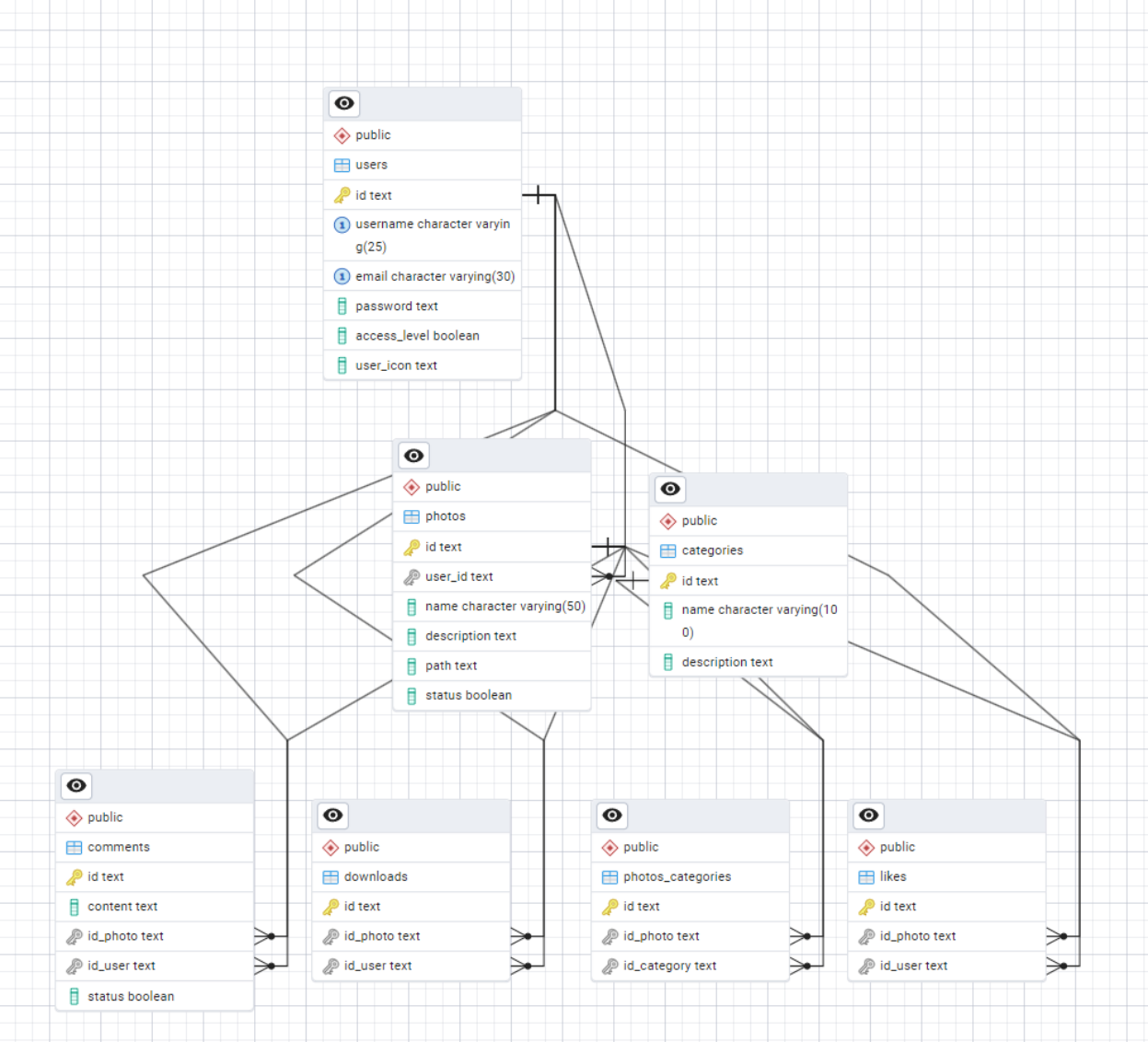


J'ai créé les prototypes en utilisant Adobe XD, comme on m'a spécifiquement demandé de le faire, bien que cette solution soit maintenant obsolète. Une solution supérieure aurait été Figma. Adobe XD offre une multitude de fonctionnalités pour rendre l'expérience de prototypage aussi proche que possible du développement réel. Dans le prototype de la page d'accueil pour le desktop, affiché ci-dessus, nous pouvons apercevoir le design de la plateforme. Le style que j'ai choisi pour le site est minimaliste, car je le trouve intrigant et c'est un choix de design populaire ces dernières années d'après mon expérience. Les choix stylistiques sont décrits dans le "cahier des charges", voici un résumé : La direction visuelle du site est guidée par une palette de couleurs naturelles (#FBB328, #A88C66, #FFF8F0, #DFEODF) générée à l'aide de "mycolor.space" pour compléter le logo. La typographie utilise la famille de polices Inter pour un look moderne et propre, avec le logo en police Star Avenue. Le design met l'accent sur la simplicité et le minimalisme, reflétant les valeurs fondamentales de "photoStockage" et garantissant une expérience utilisateur conviviale.

5. Gestion des données

i. Schéma de la base de données

Le schéma de la base de données existe dans le "cahier des projets" et il ne peut pas être résumé car il contient des informations cruciales. Je fais la référence directe ici :



La base de données se compose de sept tables. La table des utilisateurs comprend six colonnes : un ID qui sert de clé primaire et doit être unique et non nul, de type texte. Le champ nom d'utilisateur est un champ de caractères variable requis avec une longueur maximale de 25 caractères. Un champ email est inclus en tant que caractère variable avec une longueur maximale de 30 caractères, et un champ mot de passe est inclus en tant que texte. Une colonne niveau d'accès est incluse en tant que type booléen. De plus, il y a une colonne icône utilisateur de type texte.

La table des photos comprend six colonnes : ID, user_id, nom, description, chemin et statut. Le champ ID sert de clé primaire et est de type texte. User_id est une clé étrangère référençant la table des utilisateurs et est également de type texte. Le nom est un champ de caractères variable avec une longueur maximale de 50 caractères. La description est un champ texte utilisé pour décrire la photo. Le chemin est un champ texte qui définit où l'image est stockée. Le statut est un champ booléen indiquant l'état actuel de la photo.

La table des likes (qui gère les réactions des utilisateurs aux photos) se compose de trois colonnes : ID, id_photo et id_user. L'ID sert de clé primaire et est de type texte. Le champ id_photo est une clé étrangère référençant la table des photos, et le champ id_user est une clé étrangère référençant la table des utilisateurs, tous deux de type texte.

La table des catégories se compose de trois colonnes : ID, nom et description. Le champ ID est la clé primaire et est de type texte. Le champ nom est un caractère variable avec une longueur maximale de 100 caractères, et la description est de type texte. Ces champs stockent des informations sur les différentes catégories qui peuvent être attribuées aux photos.

La table des photos_catégories sert de table de jonction reliant les photos à leurs catégories. Elle contient trois colonnes : ID en tant que clé primaire de type texte, id_photo en tant que clé étrangère référençant la table des photos, et id_catégorie en tant que clé étrangère référençant la table des catégories, tous deux également de type texte.

La table des commentaires possède cinq colonnes : ID en tant que clé primaire de type texte, contenu de type texte pour stocker le contenu du commentaire, id_photo en tant que clé étrangère référençant la table des photos, id_user en tant que clé étrangère référençant la table des utilisateurs, et un champ statut de type booléen pour indiquer l'état du commentaire.

La table des téléchargements représente la dernière table du schéma, contenant trois colonnes : ID en tant que clé primaire de type texte, id_photo en tant que clé étrangère référençant la table des photos, et id_user en tant que clé étrangère référençant la table des utilisateurs, suivant quels utilisateurs ont téléchargé quelles photos.

ii. Sécurité des données

Les données sont sécurisées à l'aide de diverses bibliothèques qui assainissent et scrutent les données entrant et sortant de la base de données au niveau du backend, ainsi que la sécurité des types dans le frontend. Si le projet est un jour hébergé, il est prévu d'avoir des sauvegardes régulières de la base de données, aussi fréquentes qu'une fois par jour, en fonction du fournisseur de services et de l'espace disponible dans l'environnement hébergé.

6. Tests

i. Stratégie de test

Stratégie de test backend :

Dans le backend, j'ai implémenté des tests unitaires à l'aide de Jest, une bibliothèque pour effectuer divers tests. Dans le dossier d'intégration des tests, j'ai créé un test pour chacune des fonctionnalités de l'API backend. Cela inclut des tests pour les catégories, les commentaires, les téléchargements, les likes, les photos et les utilisateurs. Un fichier testDb est également inclus pour effectuer les tests dans un environnement séparé qui n'affecterait pas la base de données du projet.

Exemple de test unitaire :

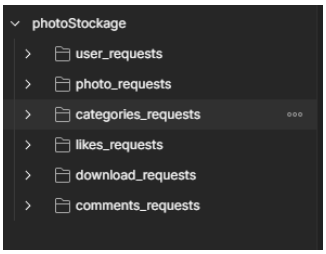
```
// __tests__/integration/user.test.js
const request = require("supertest");
const app = require("../../index");
const { pool } = require("../../utils/db");
const jwt = require("jsonwebtoken");
require("dotenv").config();

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);
    });
  });
});
```

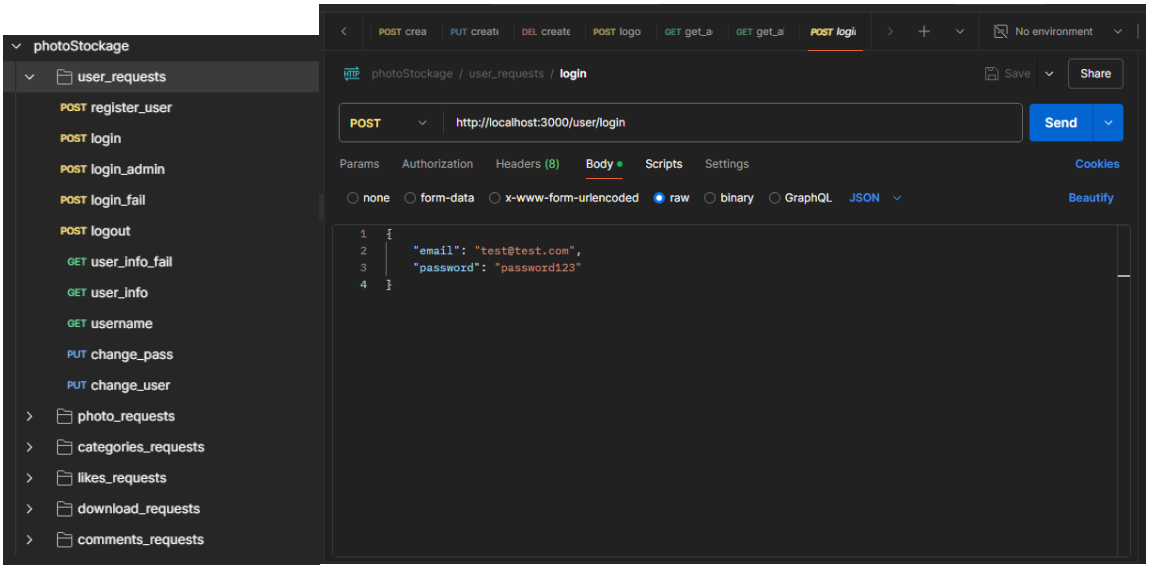
En haut du fichier, j'importe les bibliothèques nécessaires comme supertest qui est un paquet de dépendances de Jest, l'application depuis le fichier index, le pool depuis le dossier utilities, la bibliothèque jwt et les variables

d'environnement .env. Le test commence par définir la fonctionnalité, dans ce cas, il s'agit de la gestion des utilisateurs. Il crée deux variables, l'ID utilisateur et le token d'authentification, elles sont déclarées mais non définies. Le test commence par supprimer la table des utilisateurs et en cascade l'effet sur d'autres tables si nécessaire. Ensuite, un nouvel utilisateur est créé et inséré dans la base de données à l'aide de la requête : "INSERT INTO users (id, username, email, password) VALUES ($1, $2, $3, $4)", [userId, "testuser", "test@test.com", "hashedpassword"]. Un JWT est également stocké dans la variable authToken contenant les champs nécessaires de l'ID utilisateur, du nom d'utilisateur et de l'email, ainsi que le secret jwt. Après avoir exécuté les fonctions mentionnées, il tente d'effectuer la gestion du profil qui devrait réussir si le nom d'utilisateur et l'icône utilisateur peuvent être mis à jour à l'aide d'une méthode PUT et du cookie provenant du authToken. En fin de compte, nous attendons un code de statut 200, ce qui signifie que les actions ont été exécutées avec succès.

Tests des endpoints API :

J'ai effectué des tests minutieux sur les endpoints API en utilisant l'application nommée "Postman". Voici quelques exemples de mes méthodes.





J'ai divisé dans des fichiers séparés les requêtes en fonction de la fonction que je teste à chaque fois. J'ai créé les dossiers des requêtes utilisateur, photo, catégorie, likes, téléchargements et commentaires. Dans ces dossiers,

j'ai défini toutes les méthodes que l'on peut trouver dans les modèles et contrôleurs du backend et des requêtes garanties pour échouer en raison de diverses raisons. Dans l'exemple présenté, j'essaie de me connecter en utilisant un utilisateur et un mot de passe qui existent réellement, en m'attendant à un code de statut 200 contenant les données correspondantes.

ii. Gestion des bugs / erreurs

J'ai intégré la gestion des erreurs à la fois dans le backend et le frontend, comme en témoignent les exemples fournis dans le chapitre précédent. Un exemple typique :

```
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!" });
  }
}
```

Lorsqu'un utilisateur tente de se connecter, en cas d'adresse email incorrecte ou de mot de passe incorrect, un statut 401 contenant le message "Identifiants invalides" est retourné. Si le JWT ne peut pas être créé, un code de statut 500 avec le message d'erreur "Erreur interne du serveur" est retourné. Enfin, si l'utilisateur n'a pas soumis toutes les données nécessaires ou s'il y a une incompatibilité de type, un code de statut 400 avec le message d'erreur "Données invalides" est retourné.

Dans le frontend, si un message d'erreur ou un message d'avertissement est retourné par la requête au backend, il est affiché à l'aide de variables sauvegardées dans l'état du composant pour la page de connexion (en suivant l'exemple de connexion). De plus, les erreurs dans le frontend, comme les données manquantes ou les incompatibilités de type, sont gérées de manière similaire et l'utilisateur reçoit un retour immédiat en affichant ladite erreur à l'écran.

7. Déploiement et maintenance

i. Processus de déploiement

Si un déploiement est requis, il existe plusieurs chemins que je pourrais suivre pour l'effectuer. Les options sont un mélange de choix personnel et de restrictions budgétaires. Si le budget le permet, les plateformes d'hébergement du frontend comme Vercel devraient être choisies. Pour le backend, un VPS qui exécuterait le backend dans un conteneur Docker serait idéal. Quant à la base de données, Supabase pourrait être choisi car il offre une solution robuste. Dans mon cas, avec les limitations budgétaires auxquelles je suis confronté, la meilleure option serait d'utiliser un VPS qui hébergerait tout : le frontend, le backend et la base de données.

Plus en détail, le processus serait le suivant : je devrais d'abord exporter ma base de données si je souhaite conserver mes données ; sinon, le backend contient des fichiers avec la description du schéma et quelques données "démo/factices". Ensuite, je devrais créer un conteneur Docker pour le backend. Il est très important d'exclure les dossiers et fichiers node_modules du conteneur Docker car cela serait un gaspillage d'espace, de même pour le frontend. Pour le conteneur Docker, il est important que le projet ait une structure standardisée, qu'il inclue les fichiers package.json, package-lock.json, le fichier .dockerignore et le fichier Dockerfile, ainsi que tous les fichiers du projet dans un dossier src. Le fichier .dockerignore devrait inclure le dossier node_modules, les fichiers git, le fichier .dockerignore lui-même, le fichier Dockerfile et les fichiers de journalisation des erreurs/bugs. Le fichier Dockerfile devrait inclure la version de Node.js, un répertoire de travail, une commande de copie des fichiers package.json et lock, une commande pour exécuter "npm install", une commande de copie pour copier les fichiers du projet, une commande expose pour notre port souhaité et enfin une commande pour démarrer notre projet. De manière similaire, pour le frontend, les mêmes fichiers seraient requis avec l'ajout de commandes supplémentaires et d'un fichier qui spécifie une commande de build, le type d'environnement et si nous souhaitons redémarrer en cas de modifications ou non. Le dernier fichier est optionnel.

Il existe une option moins populaire consistant à envoyer le projet sur un service d'hébergement traditionnel avec des limitations majeures quant au nombre de connexions simultanées qu'il pourrait supporter en raison de ressources moins disponibles.

ii. Maintenance

Pour la partie maintenance, elle devrait être divisée en plusieurs sections. Une section quotidienne qui impliquerait des vérifications de la santé du système, des problèmes de performance potentiels, des journaux d'erreurs et des journaux de sécurité.

Sur une base hebdomadaire, une analyse de l'utilisation pour déterminer si les ressources sont correctement allouées et des vérifications de la santé de la base de données. Sur une base moins fréquente, des mises à jour

de dépendances car cela pourrait poser un risque de sécurité si elles ne sont pas mises à jour régulièrement, des révisions de code et des refactorisations pour améliorer les performances et l'analyse des retours des utilisateurs et l'ajout de fonctionnalités.

8. Défis et solutions

Pendant le développement du projet, j'ai rencontré de nombreux défis, principalement en raison du manque d'expérience préalable, en particulier concernant le backend. En utilisant diverses ressources, j'ai réussi à surmonter les obstacles et à produire un produit dont je suis fier.

L'un des premiers défis auxquels j'ai été confronté était la phase de prototypage. N'ayant pas de formation en design graphique ou web, l'utilisation d'Adobe XD était fastidieuse. La solution à ce problème est venue de la consultation de vidéos tutorielles sur des plateformes comme YouTube et de la recherche de conseils et de guides sur des forums dédiés à la conception avec Adobe XD ou à la conception web en général.

Un autre défi auquel j'ai été confronté concernant le frontend était la reconversion du style du site. Étant donné qu'il s'agit du premier projet dans lequel j'ai utilisé Tailwind, j'ignorais que le framework est mobile-first. J'ai développé de nombreuses pages et composants sur le moniteur de bureau en premier, pour découvrir plus tard que je devais réécrire mes styles en commençant par un écran mobile et en remontant vers des écrans plus grands. La solution au problème a été de relire les styles et d'essayer de les réimaginer en mobile-first.

Le plus grand défi a été la gestion du token JWT. Étant donné que c'était le premier projet dans lequel j'utilisais JWT et les cookies HTTP uniquement, je ne savais pas comment gérer le JWT dans le frontend, le cookie dans le backend et comment implémenter les fonctions d'authentification. Pendant la formation, nous avons développé un certain nombre de projets utilisant JWT, ce qui a beaucoup aidé à son implémentation.

9. Documentation et conclusion

i. Documentation

J'ai inclus de la documentation et des instructions d'implémentation pour le projet. Les noms de variables et de fonctions sont intuitifs pour aider les autres développeurs à utiliser le projet s'ils le souhaitent. Il existe une documentation pour le frontend et le backend. Voici un exemple de la documentation de l'API.

PhotoStockage Backend Documentation

Table of Contents

1. Project Overview
2. System Architecture
3. Getting Started
4. API Reference
5. Database Schema
6. Authentication & Authorization
7. Security Features
8. Testing
9. Error Handling

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

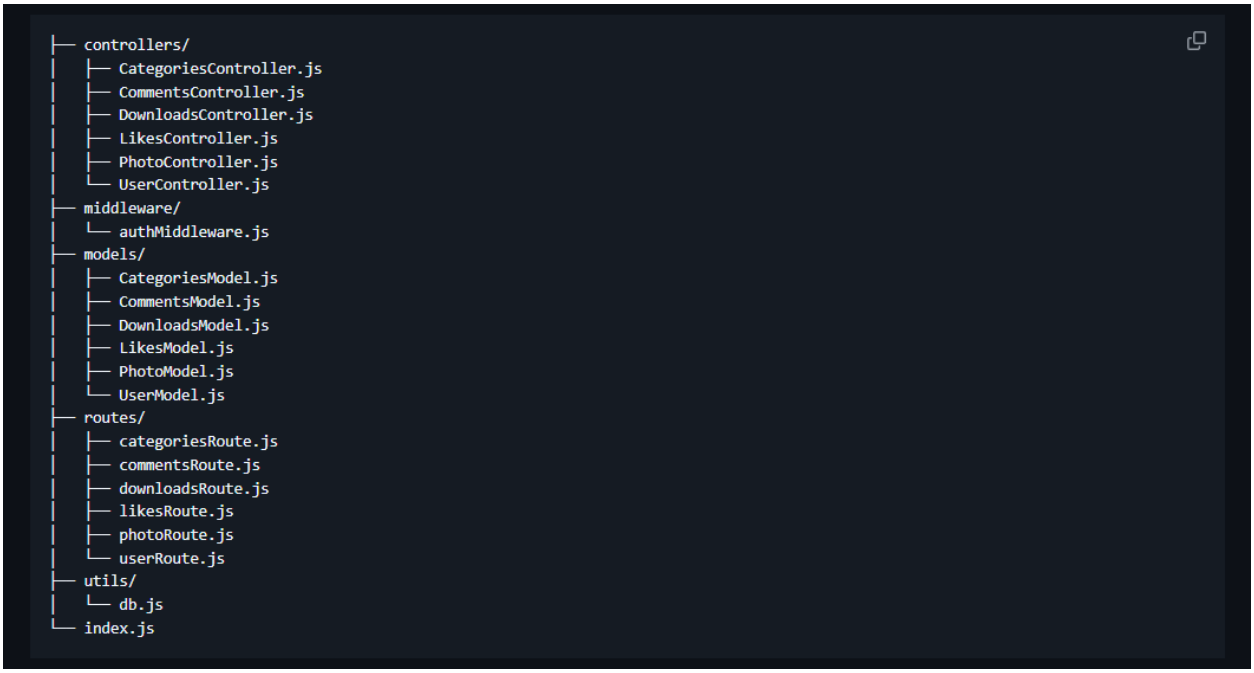
Core Components

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



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

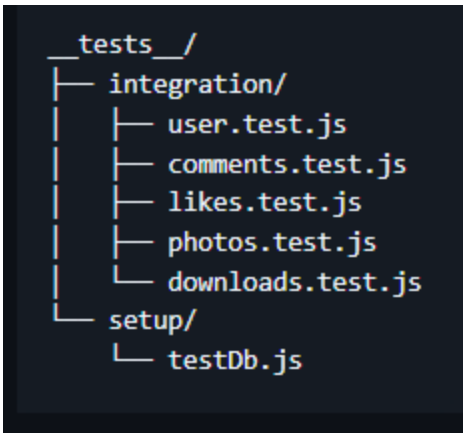
```
const authMiddleware = (req, res, next) => {
  const token = req.cookies.token;
  if (!token) {
    return res.status(401).json({ error: "No token provided" });
  }
  // Token verification and user attachment to request
};
```

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



Running Tests

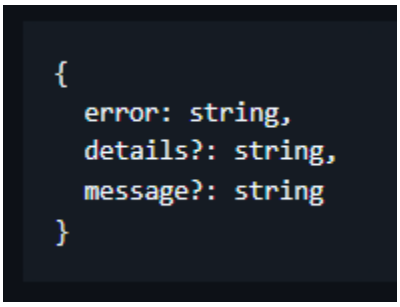
npm test

Coverage Areas

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

Error Handling

Standard Error Response Format



HTTP Status Codes

- 200: Success
- 201: Created
- 400: Bad Request
- 401: Unauthorized
- 403: Forbidden

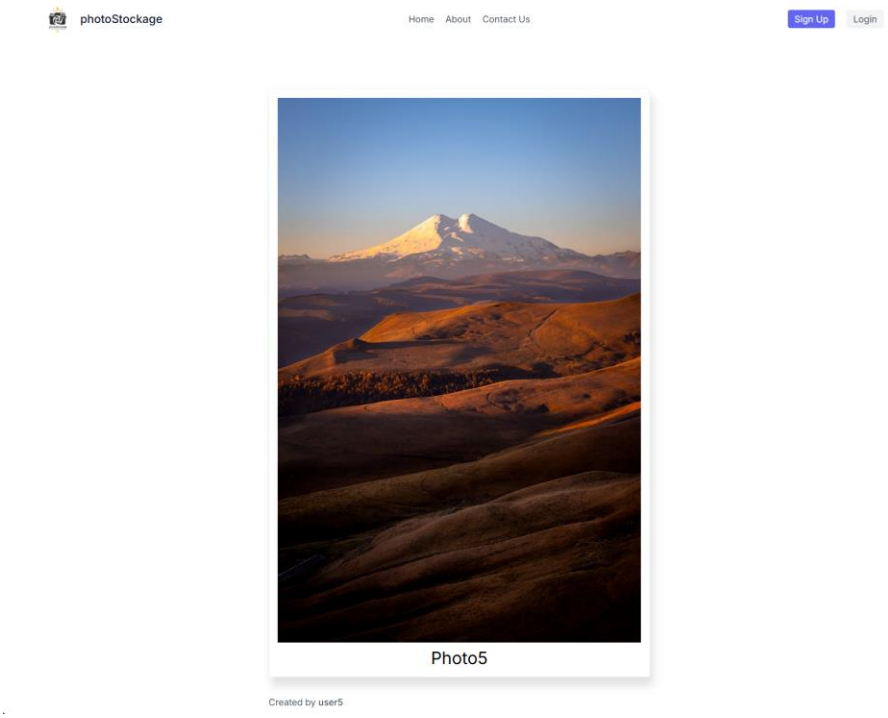
- 404: Not Found
- 500: Internal Server Error

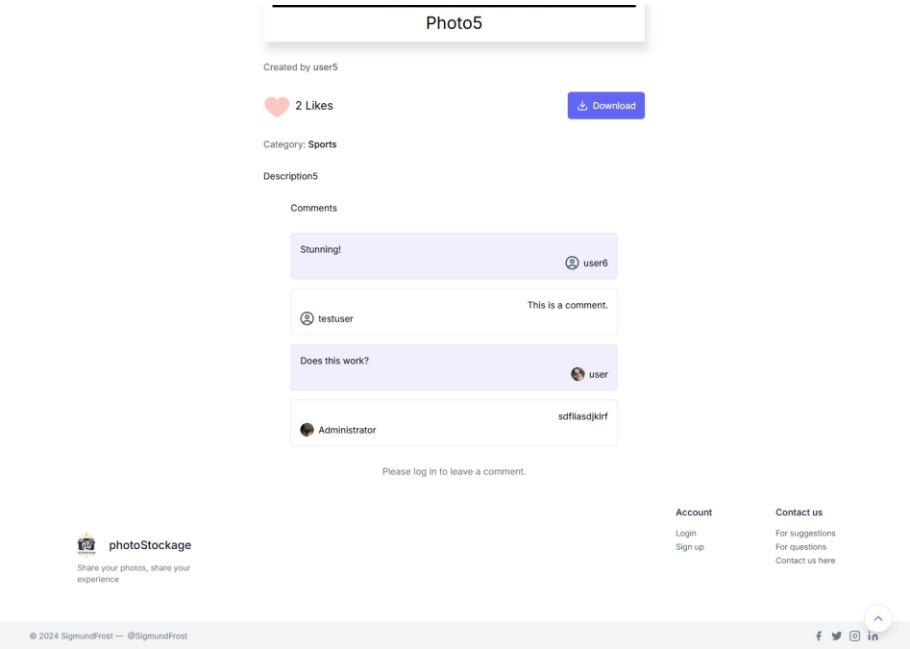
ii. Conclusion

Pendant le développement de ce projet, je me suis beaucoup amélioré en tant que développeur. C'était une excellente opportunité d'apprendre de nouvelles technologies comme Express, Next.js et Tailwind, qui m'ont aidé à créer des projets plus rapidement et de manière plus efficace. J'ai eu l'occasion d'en apprendre davantage sur la sécurité des projets, de travailler sur mes compétences en backend et d'appliquer mes connaissances sur les technologies que je maîtrisais déjà. J'ai également eu la liberté d'utiliser mon imagination et de créer un site que j'avais envisagé, avec mon style personnel et de m'exprimer. À l'avenir, je revisiterai le projet pour améliorer son code, le rendre plus performant, appliquer de nouvelles directives de sécurité et apporter des mises à jour stylistiques.

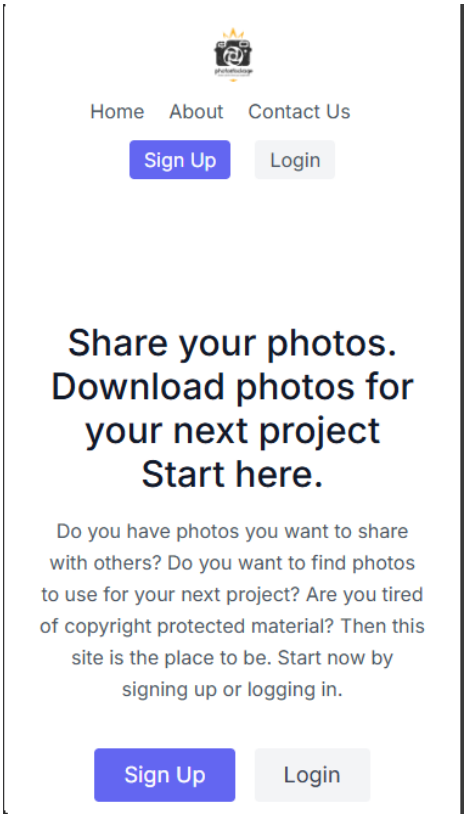
10. Annexes

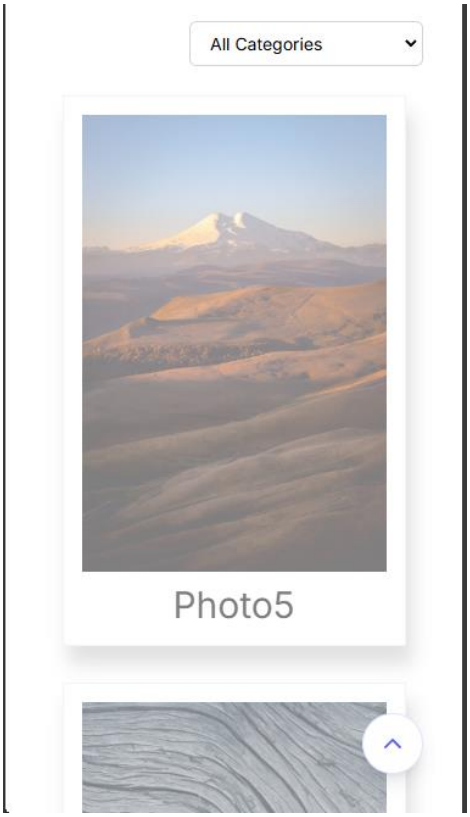
Interface desktop :

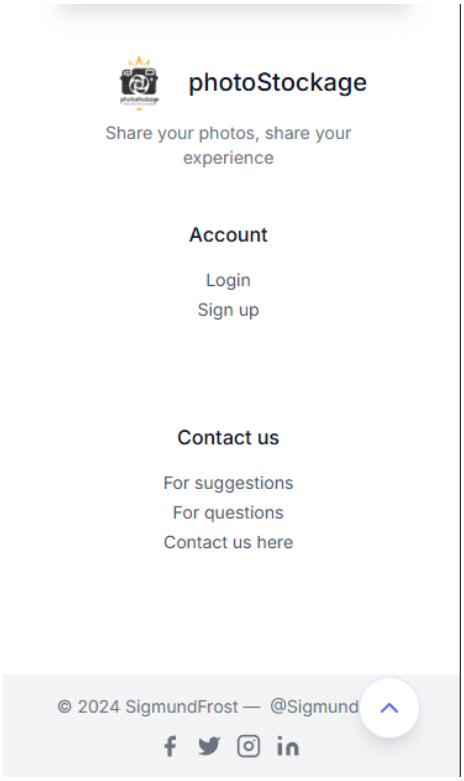




Interface mobile :







Dépôt GitHub :

Voici le lien vers le dépôt GitHub du projet qui inclut l'ensemble du projet, les images, les captures d'écran et les autres sources utilisées dans sa réalisation : https://github.com/EsFrost/m2i_projet . Le projet lui-même est hébergé en tant que démo à l'adresse suivante : https://photostockage.vercel.app .