🌍 Photomap Application

Photomap is a full-stack web application that allows users to upload, organise, and visualise travel photos on an interactive world map. Designed for both personal photo management and educational purposes, Photomap encourages users to reflect on their journeys while learning about global geography and culture.

📑 Table of Contents

Features

- 🌐 Interactive world map with country-level photo highlights
- 🔐 Secure user authentication (JWT-based)
- 🖼️ Photo upload, filtering, deletion, and organization by country/year
- 📅 Chronological travel timeline view
- 📚 Real-time educational data: capital, currency, language, weather, time, and exchange rate
- 📁 AWS S3 for photo storage with Lambda-powered resizing
- 🏞️ Premium features: album creation by country and custom labels

---

Technologies

- **Frontend:** React (Vite), Chakra UI, React Leaflet, React Query
- **Backend:** Java 21, Spring Boot 3.3.3, Spring Security, JWT
- **Database:** PostgreSQL (Render)
- **Image Storage:** Amazon S3
- **Hosting:**
  - Frontend on Vercel
  - Backend on Render

Prerequisites

- Node.js ≥ 16.x
- Java ≥ 17
- Maven
- PostgreSQL (pgAdmin 4 recommended)
- AWS S3 Bucket and LAMBDA
- InteliJ IDE and VScode (optional)

Java 21 | Spring Boot 3.3.3 | Frontend React | Bundler Vite | Database PostgreSQL | Storage AWS S3
Backend Hosted on Render | Frontend Hosted on Vercel | license MIT | Project Status Production Ready

---

Production Deployment

Backend

Before deploying, make sure your development environment is set up with the necessary tools:

1. Clone the repository

```
git clone https://github.com/felixdeveloper87/personal-photo-map-v3-backend
cd personal-photo-map-v3-backend
```

The backend is containerised using a custom Dockerfile for production deployment.
The file `Dockerfile.prod` is used by Render to build and run the Spring Boot application in a secure and scalable environment.

Dockerfile.prod (used by Render)

```
# Build stage
FROM maven:3.9.6-eclipse-temurin-21 AS build
WORKDIR /app
COPY pom.xml ./pom.xml
COPY src ./src
RUN mvn clean package -DskipTests
```

```
# Runtime stage
FROM openjdk:21-jdk-slim
WORKDIR /app
COPY --from=build /app/target/*.jar app.jar
ENV SERVER_PORT=8092
EXPOSE 8092
CMD ["java", "-jar", "app.jar"]
```

Render automatically builds and runs this Dockerfile when you connect your GitHub repository.

Deploying to Render

1. Create the PostgreSQL Database

1. Go to https://dashboard.render.com
2. Login with Github
3. New > PostgreSQL
4. Give it a name (eg., photomap-db)
5. Choose the region
6. Choose Instance type
7. Click in "Create Database"

Save the Connection Credentials: After creation Render will give you:

1. Host
2. User
3. Password
4. Database
5. Port
6. Internal Database URL (for apps hosted on Render)
7. External Database URL (for connecting from your local machine)

You'll use these in your backend configuration

---

2. Create the Backend Web Service

1. Go to https://dashboard.render.com
2. New > Web service
3. Connect your GitHub repo
4. Name it
5. For environment, select "Docker"
6. Dockerfile path: Dockerfile.prod
7. Leave start command blank Set environment variables (important!) Add variables like:
8. SPRING_DATASOURCE_URL=jdbc:postgresql://your-host:5432/your-db
9. SPRING_DATASOURCE_USERNAME=your-user
10. SPRING_DATASOURCE_PASSWORD=your-password

11. SPRING_JPA_HIBERNATE_DDL_AUTO=update You get these values from your Render PostgreSQL database
12. Click in "Create Web Service"

Setting up AWS S3 and Access Keys

To enable secure photo uploads and cloud storage, you need to configure an Amazon S3 bucket and valid AWS credentials.

---

1. Create an S3 Bucket

1. Go to https://s3.console.aws.amazon.com/s3/
2. Click **"Create bucket"**
3. Set a **unique name** (e.g., `photomap-user-images`)
4. Choose a region (e.g., `eu-west-2` for London)
5. **Uncheck** "Block all public access" *(you will manage access with bucket policies or IAM roles)*
6. Leave other options as default or enable versioning (optional)
7. Click **"Create bucket"**

---

2. Create an IAM User with S3 Access

1. Go to https://console.aws.amazon.com/iam/
2. Click **Users** → **Add users**
3. Enter a username (e.g., `photomap-user`)
4. Tick **"Programmatic access"**
5. Click **Next: Permissions**
6. Choose **"Attach policies directly"**
7. Select **AmazonS3FullAccess**
   (Or create a custom policy to restrict actions and bucket if needed)
8. Click **Next** → **Create user**
9. Copy and securely store the **Access Key ID** and **Secret Access Key**

---

3. Add Environment Variables to Render

In your **Render backend service**, go to the "Environment" tab and add the following environment variables:

```
AWS_ACCESS_KEY=your_aws_access_key
AWS_SECRET_KEY=your_aws_secret_key
S3_BUCKET_NAME=your_bucket_name
```

These credentials will be injected into your Spring Boot application at runtime. Make sure they match the names expected in your application.properties.

4. (Optional) Configure AWS Lambda for Image Resizing

If you want to automatically resize images when a user uploads high-resolution files to your S3 bucket, you can use AWS Lambda to trigger a function on upload.

Steps:

1. Go to https://console.aws.amazon.com/lambda/
2. Click **"Create function"**
3. Choose **"Author from scratch"**
4. Set a name (e.g., `resizePhotomapImages`)
5. Choose a runtime:
   - **Node.js** (recommended with `sharp` library)
   - or **Python** (with `Pillow`)
6. Under **Permissions**, choose:
   - **"Create a new role with basic Lambda permissions"**
7. Click **"Create function"**

Add an S3 Trigger:

1. In the **Function overview**, click **"Add trigger"**
2. Choose **S3** as the source
3. Select your S3 bucket
4. Set **Event type** to `PUT`
5. (Optional) Add a prefix (e.g., `uploads/`) to limit what triggers the function
6. Click **Add**

Lambda Logic (`index.js`)

```
const { S3Client, GetObjectCommand, PutObjectCommand, HeadObjectCommand } =
require("@aws-sdk/client-s3");
const sharp = require('sharp');

const s3 = new S3Client({ region: "eu-north-1" });

exports.handler = async (event) => {
  const bucket = event.Records[0].s3.bucket.name;
  const key = event.Records[0].s3.object.key;
  const maxSize = 1 * 1024 * 1024; // 1 MB

  try {
    const metadataParams = new HeadObjectCommand({ Bucket: bucket, Key: key });
    const metadata = await s3.send(metadataParams);

    if (metadata.Metadata?.optimized === 'true') {
      console.log(`Image ${key} has already been optimized.`);
      return { status: 'Image already optimized, skipped' };
    }
    const params = new GetObjectCommand({ Bucket: bucket, Key: key });
    const { Body } = await s3.send(params);

    let quality = 90;
    let resizedImage = sharp(await Body.transformToByteArray())
      .rotate()
      .resize({ width: 1920, fit: 'inside' });

    let outputBuffer;
    do {
      outputBuffer = await resizedImage.jpeg({ quality: quality }).toBuffer();
      quality -= 10;
    } while (outputBuffer.length > maxSize && quality > 20);

    const putParams = new PutObjectCommand({
      Bucket: bucket,
      Key: key,
      Body: outputBuffer,
      ContentType: 'image/jpeg',
      Metadata: { optimized: 'true' }
    });

    await s3.send(putParams);

    return { status: 'Image optimized and saved', key };
  } catch (error) {
    console.error("Error processing image:", error);
    return { status: 'Error', error: error.message };
  }
};
```

IAM Permissions Required

Attach this policy to your Lambda execution role to allow access to your S3 bucket:

```
{
  "Effect": "Allow",
  "Action": [
    "s3:GetObject",
    "s3:PutObject",
    "s3:HeadObject"
  ],
  "Resource": "arn:aws:s3:::your-bucket-name/*"
}
```

Replace your-bucket-name with the actual bucket name. With this setup, high-resolution images uploaded to your S3 bucket will be automatically resized and compressed in-place, keeping storage efficient and frontend performance optimal.

Environment Variables

This project requires a set of environment variables for database connection, AWS S3 integration, and JWT authentication.

Before running the backend locally or deploying it to a cloud provider like Render, make sure to configure the following environment variables.

You can use the `.env.example` file as a reference:

```
.env.example .env
# PostgreSQL Configuration
SPRING_DATASOURCE_URL=jdbc:postgresql://your-host:5432/your-db
SPRING_DATASOURCE_USERNAME=your-username
SPRING_DATASOURCE_PASSWORD=your-password
SPRING_JPA_HIBERNATE_DDL_AUTO=update

# AWS S3 Configuration
AWS_ACCESS_KEY=your-aws-access-key
AWS_SECRET_KEY=your-aws-secret-key
S3_BUCKET_NAME=your-bucket-name
AWS_REGION=your-region

# JWT Configuration
JWT_SECRET=your-jwt-secret
```

✅These values must also be set in the Environment section of your Render Web Service (for production).

Frontend (Vite + React)

The frontend of the Photomap project is built using **React** with **Vite** for fast development and performance. It connects to the backend via REST APIs and handles authentication, photo management, map interaction, and the user interface.

The frontend is located in a **separate GitHub repository**: 👉 Photomap Frontend Repository

---

Frontend Project Setup and Deployment

This guide outlines the steps to set up, run, and deploy the `personal-photo-map-v3-Frontend` project, built with Vite and React, and hosted on Vercel, with a backend on Render.

1. Clone the frontend repository

```
git clone https://github.com/felixdeveloper87/personal-photo-map-frontend
cd personal-photo-map-frontend
```

Installation

2. Install dependencies

```
npm install
```

This installs all required libraries listed in package.json.

3. Set up environmente variable

- Create a .env file in the project root with:

```
VITE_API_URL=https://your-backend-api.onrender.com
```

- Replace the URL with the actual backend API URL hosted on Render.

Running Locally

4. run the development server

```
npm run dev
```

The application will be available at http://localhost:5173 by default.

Frontend Deployment on Vercel

The frontend is hosted on Vercel, a platform optimized for frontend frameworks like Vite and React.

Deployment steps:

1. Go to https://vercel.com and log in with your GitHub account
2. Click **"New Project"** and import the `personal-photo-map-v3-Frontend` repository
3. Vercel will automatically detect the framework as **Vite**
4. Add the following environment variable:

```
VITE_API_URL=https://your-backend-api.onrender.com
```

5. Click "Deploy"

Once deployed, Vercel will provide a public URL like: https://personal-photo-map-frontend.vercel.app

✅ The frontend is now live and connected to the backend hosted on Render.

Note – CORS Configuration on Backend

Ensure your backend allows requests from the Vercel frontend URL by configuring CORS in your Spring Boot controller:

```
@CrossOrigin(origins = "https://personal-photo-map-frontend.vercel.app")
```

This enables secure cross-origin requests between frontend and backend in production.

Additional Notes

- **CORS:** Ensure the backend CORS configuration exactly matches your Vercel frontend URL, including `https://`, to avoid blocked requests.
- **Environment Variables:** Add `.env` to your `.gitignore` file to avoid accidentally committing sensitive information.
- **Vercel Auto-Deployment:** When your GitHub repository is connected to Vercel, every push to the `main` branch will trigger an automatic redeployment.
- **Troubleshooting:**
- If the frontend cannot connect to the backend, double-check:
  - `VITE_API_URL` in your `.env`
  - Backend CORS configuration
- Review deployment logs on Vercel for any build or runtime errors.

License

This project is licensed under the **MIT License** – see the LICENSE file for details.

Author

Leandro Felix
GitHub
LinkedIn