**📌 Vista Album – Project Status**

**Date:** 19 Sept 2025  
**Time:** 12:25 AM IST

**✅ Completed so far**

1. **Project Setup**
   * Created **frontend (Next.js/TypeScript)** and **backend (Node/Express/TypeScript)** folders.
   * Added **ESLint, Prettier, TypeScript** configs.
   * Initialized package.json in both frontend & backend.
2. **Google OAuth Setup**
   * Registered project in **Google Cloud Console**.
   * Created OAuth **Client ID** and configured **Authorized Redirect URIs**.
   * Backend route (/auth/google) implemented for Google login.
   * Verified login flow → **redirect → Google → approve access**.
   * Resolved earlier errors:
     + redirect\_uri\_mismatch
     + No code provided
     + invalid\_request
   * ✅ Now OAuth is **working successfully** → we are receiving **access & refresh tokens**. 🎉
3. **Git Setup**
   * Initialized Git repository locally.
   * First commit: "Upto now we have verified the google auth".
   * Remote repo created: SriPrasannanjaneyuluK/VistaAlbum.
   * Facing push issue (403 error → PAT authentication issue).

**🔄 In Progress**

* **GitHub Push** → resolving PAT (Personal Access Token) setup to push local commits.
* **Backend integration** → tokens acquired but not yet used with **Google Photos API**.

**🚀 Next Steps**

1. **Fix Git push** → sync local repo with GitHub.
2. **Integrate Google Photos API**:
   * Use access token to fetch user albums/photos.
   * Add backend endpoint /photos to return photo data.
   * Extend later to **save selected photos**.
3. **Frontend work (Next.js)**:
   * Add **Login with Google** button.
   * Connect frontend to backend OAuth flow.
   * Display fetched photos in a simple UI.

👉 **Current Position**  
✅ Setup & Google OAuth flow complete.  
⚠️ Blocker: Git push with PAT.  
🎯 Next milestone: **Google Photos API integration + frontend display**.

**Vista Album – Progress Documentation**

**Date:** 19 Sept 2025  
**Time:** 11:55 PM IST

**1. Google OAuth Setup (Backend)**

* Implemented **Express route** (/api/auth/google) to redirect users to Google OAuth login.
* Added required **scopes**:
  + openid
  + email
  + profile
  + https://www.googleapis.com/auth/photoslibrary.readonly
* Implemented **callback route** (/api/auth/google/callback) to:
  + Receive authorization code from Google.
  + Exchange code for **access\_token** + \*\*refresh\_token`.
  + Log/store tokens for API usage.

✅ **Status:** OAuth flow is working → User can log in & we successfully receive tokens.  
⚠️ **Issue faced:** *“No code provided”* → Fixed by ensuring redirect URI matches in **Google Cloud Console** and .env.

**2. MongoDB Setup Decision**

* Evaluated database options:
  + **Local MongoDB** → requires installation on PC.
  + **MongoDB Atlas (Cloud)** → easier, scalable, multi-user ready.
* ✅ Decision: **MongoDB Atlas** chosen.
* Created connection string (to be stored in .env).

Example:

MONGO\_URI=mongodb+srv://vistaAdmin:<password>@cluster0.abcd123.mongodb.net/vistaAlbum?retryWrites=true&w=majority

👉 Database name finalized: **vistaAlbum**.

**3. Next Planned Step**

* Connect backend to MongoDB Atlas using **Mongoose**.
* Create **User Schema** with fields:
  + googleId
  + email
  + accessToken
  + refreshToken
  + name
  + profilePic
* Update /api/auth/google/callback route → instead of console logging, **store tokens in MongoDB**.

**✅ Summary (Today’s Work)**

* 🔑 Google OAuth flow successfully implemented.
* 🗄️ Database selected: **MongoDB Atlas (vistaAlbum)**.
* 🎯 Next milestone: **Connect to MongoDB + implement User Schema & persistence**.

**📒 Vista Album – Development Log (Google OAuth + MongoDB Integration)**

**Date:** 20-Sep-2025  
**Objective:** Implement **Google OAuth authentication** for multiple users and securely store their information in **MongoDB Atlas**.

**1. Project Setup**

* Backend: **Node.js + Express + TypeScript**
* Frontend: **Next.js**
* Installed dependencies:

npm install express dotenv googleapis mongoose ts-node-dev typescript @types/express

* Configured **TypeScript** with tsconfig.json (strict mode).
* Connected backend to **MongoDB Atlas**.

**2. Google OAuth Integration**

**2.1 OAuth Setup**

* Created **OAuth credentials** in Google Cloud Console:
  + Client ID
  + Client Secret
  + Redirect URI → http://localhost:5000/auth/google/callback
* Added scopes:

const SCOPES = [

"openid",

"email",

"profile",

"https://www.googleapis.com/auth/photoslibrary.readonly"

];

**2.2 Backend Routes**

* auth.ts handles Google login & callback:
  + /auth/google → redirect to Google login
  + /auth/google/callback → handle callback, save tokens + user info
* OAuth client setup:

const oauth2Client = new google.auth.OAuth2(

process.env.GOOGLE\_CLIENT\_ID,

process.env.GOOGLE\_CLIENT\_SECRET,

process.env.GOOGLE\_REDIRECT\_URI

);

* User info stored in MongoDB (users collection):

const user = await User.findOneAndUpdate(

{ googleId: userInfo.data.id },

{

googleId: userInfo.data.id,

email: userInfo.data.email,

name: userInfo.data.name,

profilePic: userInfo.data.picture,

accessToken: tokens.access\_token,

refreshToken: tokens.refresh\_token,

},

{ upsert: true, new: true }

);

**3. MongoDB Atlas Integration**

* Database: **MongoDB Atlas** (vistaAlbum)
* .env connection string:

MONGO\_URI=mongodb+srv://<username>:<password>@cluster0.mongodb.net/vistaAlbum?retryWrites=true&w=majority

* Connection file db.ts:

import mongoose from "mongoose";

const connectDB = async () => {

try {

await mongoose.connect(process.env.MONGO\_URI as string);

console.log("✅ MongoDB connected");

} catch (err) {

console.error("❌ MongoDB connection error:", err);

}

};

export default connectDB;

* Verified user documents saved in **MongoDB Atlas dashboard**.

**4. Backend Index Setup**

* index.ts:

import express from "express";

import dotenv from "dotenv";

import authRoutes from "./routes/auth";

import connectDB from "./config/db";

dotenv.config();

connectDB();

const app = express();

const PORT = process.env.PORT || 5000;

app.use(express.json());

app.use("/auth", authRoutes);

app.listen(PORT, () => {

console.log(`🚀 API is running on port ${PORT}`);

});

* Routes:
  + http://localhost:5000/auth/google → start login
  + http://localhost:5000/auth/google/callback → handle callback

**5. Issues & Fixes**

| **Issue** | **Resolution** |
| --- | --- |
| Cannot GET /auth/google | Confirmed route registration with app.use("/auth", authRoutes) |
| *No code provided* error | Fixed redirect URI mismatch in Google Cloud Console |
| ERR\_CONNECTION\_REFUSED after callback | Checked server running & correct localhost port |
| connectDB is not a function error | Fixed db.ts export to use default |
| MongoDB import/type errors | Installed correct mongoose + @types/mongoose |
| 404 on callback | Ensured /auth/google/callback route exists & /auth prefix is correct |

**6. Testing & Verification**

1. Start backend:

npm run dev

1. Visit → http://localhost:5000/auth/google
   * Redirects to Google login
   * Tokens generated successfully
   * User info stored in MongoDB Atlas
2. Verified in Atlas:
   * Users stored in users collection
   * Multiple accounts supported

**7. Next Steps**

1. **Frontend Integration** → Add “Sign in with Google” button (Next.js).
2. **Token Handling** → Store securely, implement refresh flow.
3. **User Profile API** → /profile route to return user data.
4. **Multi-user Testing** → Test multiple logins with unique documents.

**✅ Current Status**

* ✔ Google OAuth working end-to-end
* ✔ User info stored in MongoDB Atlas
* ✔ Backend is ready for frontend integration

# **Vista Album – Development Documentation**

**Version:** 1.0  
**Sprint:** 5 – Google OAuth & Dashboard Integration  
**Date:** 22-Sep-2025  
**Prepared by:** Anand Kalvakolanu  
**Objective:** Fix Next.js runtime errors, implement Google OAuth for multiple users, and display user data securely from MongoDB Atlas.

## **1. Summary**

This development cycle focused on:

1. Resolving **Next.js runtime errors** caused by conflicting App Router and Pages Router usage.
2. Fixing **MongoDB authentication issues** and connecting the backend to **MongoDB Atlas**.
3. Implementing a full **Google OAuth flow**, storing user info and tokens in MongoDB.
4. Updating the **Dashboard frontend** to fetch and render user profile data correctly.

## **2. Issues, Causes & Resolutions**

| **Issue** | **Cause** | **Resolution** | **Reference / Ticket** |
| --- | --- | --- | --- |
| Next.js Runtime Error: “App Router and Pages Router both match path /” | Conflicting App Router (src/app) and Pages Router (pages/) | Fully migrate to **App Router**. Remove conflicting Pages Router files (pages/index.tsx). | #JIRA-101 |
| MongoDB Authentication Failed | Incorrect .env credentials | Updated .env with correct username/password and verified Atlas user has read/write access | #JIRA-102 |
| Google OAuth: Missing redirect\_uri / redirect\_uri\_mismatch | Backend redirect\_uri did not match Google Cloud Console | Registered exact redirect URI in console and used process.env.GOOGLE\_REDIRECT\_URI in backend | #JIRA-103 |
| Cannot GET /api/auth/google/callback | Missing leading slash in route path | Fixed route to /google/callback in routes/auth.ts | #JIRA-104 |
| Unexpected token '<' while fetching images | Attempting to .json() image URLs | Fetch JSON from backend API instead of using image URLs directly | #JIRA-105 |
| Profile picture not rendering | Missing or incorrect profilePic field | Added fallback avatar and ensured profilePic is stored as string | #JIRA-106 |

## **3. Backend Updates**

### **3.1 Google OAuth Flow**

**Routes in routes/auth.ts:**

router.get("/google", (req, res) => {

const url = oauth2Client.generateAuthUrl({

access\_type: "offline",

prompt: "consent",

redirect\_uri: process.env.GOOGLE\_REDIRECT\_URI,

scope: [

"openid",

"email",

"profile",

"https://www.googleapis.com/auth/photoslibrary.readonly"

],

});

res.redirect(url);

});

router.get("/google/callback", async (req, res) => {

const code = req.query.code as string;

if (!code) return res.status(400).send("No code provided by Google");

const { tokens } = await oauth2Client.getToken({

code,

redirect\_uri: process.env.GOOGLE\_REDIRECT\_URI,

});

oauth2Client.setCredentials(tokens);

const oauth2 = google.oauth2({ auth: oauth2Client, version: "v2" });

const userInfo = await oauth2.userinfo.get();

const user = await User.findOneAndUpdate(

{ googleId: userInfo.data.id },

{

googleId: userInfo.data.id,

email: userInfo.data.email,

name: userInfo.data.name,

profilePic: userInfo.data.picture || "",

accessToken: tokens.access\_token,

refreshToken: tokens.refresh\_token,

},

{ upsert: true, new: true }

);

res.redirect(`http://localhost:3000/dashboard?user=${user.\_id}`);

});

**Notes:**

* Ensure profilePic is stored as string.
* Include "profile" scope to retrieve Google profile picture.

### **3.2 MongoDB Integration**

**Connection file:** backend/src/config/db.ts

import mongoose from "mongoose";

const connectDB = async () => {

try {

await mongoose.connect(process.env.MONGO\_URI as string);

console.log("✅ MongoDB connected");

} catch (err) {

console.error("❌ MongoDB connection error:", err);

}

};

export default connectDB;

* Database: vistaAlbum (MongoDB Atlas)
* Verified multiple users can sign up without conflicts.

### **3.3 Users Route**

**File:** routes/users.ts

import express from "express";

import User from "../models/User";

const router = express.Router();

router.get("/:id", async (req, res) => {

try {

const user = await User.findById(req.params.id).lean();

if (!user) return res.status(404).json({ message: "User not found" });

res.json(user);

} catch (err) {

console.error(err);

res.status(500).json({ message: "Server error" });

}

});

export default router;

* Registered in index.ts as:

app.use("/api/users", userRoutes);

## **4. Frontend Updates**

**File:** src/app/dashboard/page.tsx

* Fetch user JSON from backend.
* Render profile picture with fallback avatar.
* Use searchParams safely via useSearchParams().

**Key snippet:**

<img

src={user.profilePic || `https://ui-avatars.com/api/?name=${encodeURIComponent(user.name)}&size=128`}

alt="Profile Picture"

className="mt-2 w-24 h-24 rounded-full object-cover border border-gray-300"

/>

## **5. Outcome**

* Google OAuth login fully functional.
* Backend stores user info and tokens in MongoDB Atlas.
* Dashboard fetches and displays user data correctly.
* All runtime, 400/404, and JSON parsing errors resolved.

## **6. Next Steps**

1. Add **frontend login button** to trigger OAuth flow.
2. Implement **refresh token handling** for longer sessions.
3. Create **profile page** to display user data.
4. Perform **multi-user testing**.