📝 Project Name: DevGallery — A Developer Portfolio with Blog & Media Upload

Type: Mid Trem Exam
Frontend: React.js
Backend: Firebase Realtime Database + Firebase Auth
Media Hosting: Cloudinary

Key Features 👍

Build a small single-page web app called DevGallery where:

- ☐ **Users can register/login**
- ☐ **Users can post blog articles**
- ☐ **Users can upload a profile picture (via Cloudinary)**
- ☐ **Users can write blog posts that are saved to Firebase Realtime DB**
- ☐ **Public blog feed is shown to all users**
- ☐ **Use context, hooks, memoization, and suspense techniques**
- ☐ **The user can change the theme (dark/ light)**

TASK BREAKDOWN

🔐 Task 1: Set up the Authentication System

Use: `useState`, `useEffect`, `useRef`, Firebase Auth

✅ Features:

1. **Email/password signup & login.**

2. **Google sign-in / emailandpassword (use any 1 method, either Google or email).**
3. **Validation must (simple validation, || or you can use React form hooks for dynamic validation)**
4. **Show error messages if login/signup fails**
5. **Redirect to homepage on login**

👉 ***Firebase: Firebase Auth***

📤 Task 2: File Upload to Cloudinary

🎯 **Goal:** Let the user upload a **profile picture** from the **Profile Page**, upload it to **Cloudinary**, and **save the image URL** in Firebase Realtime Database under the current user's record.

Use: `useRef`, `useState`

✅ Features to Implement:

1. **User selects a profile photo (image file).**
2. **The image is uploaded to Cloudinary.**
3. **Cloudinary returns the secure URL of the uploaded image.**
4. **Save the URL in Firebase Realtime DB at path:**
   `users/{uid}/profilePic`
5. **Optionally show a loading/spinner while uploading.**

⚙️ Process (Step-by-Step with Hints):

1. **Create a file input using `useRef` or `onChange` to get the selected image.**
2. **Use `FormData` to prepare the file for upload.**
3. **Make a POST request to Cloudinary's unsigned upload endpoint.**
4. **Get the returned image URL from Cloudinary.**
5. **Use the Firebase Realtime Database API to store the image URL under: (you can design as your recommendation)**

   users/

   └── {userId}/

   └── profilePic: "https://res.cloudinary.com/..." (Demo Database Design)

6. **Show a success message for the uploaded image.**
7. **(Optional) Disable the upload button while the image is uploading.**

📝 *Task 3: Add Blog Post with Realtime Database*

🎯 *Goal:* Allow authenticated users to write and submit blog posts with a title, description, image, and author profile picture. Save the post to Firebase Realtime DB and show a loader during submission.

Use: `useState`, `useEffect`, Firebase DB

✅ Features:

- **A blog form with fields: `title`, `description`, `image`.**
- **If the image is selected, upload it to Cloudinary first.**
- **Save blog post to Firebase at: `posts/{uniquePostId}` with fields:**

  {

"title": "...",

"description": "...",

"image": "https://...",

"authorId": "uid",

“Profile_picture”: “htttp://clou…..”

"createdAt": "timestamp"

}

- **Show a loader while data is being sent.**
- **Reset the form after successful submission.**

🌐 *Task 4: Public Blog Feed with Optimization*

Use: `useMemo`, `React.memo`, `useCallback`, `Suspense + fallback`

✅ **Features:**

- **Show list of all blog posts**
- **Optimize rendering with** `React.memo` **and** `useMemo`
- **If loading, show fallback UI using** `Suspense`

🌍 *Task 5: Global User State Using useContext*

✅ **Goal:** Manage and access authenticated user's data globally (e.g., in Header, Profile, BlogPost components) using Context API and a custom hoo

Use: `useContext`**, Custom Hook**

✅ **Features:**

- Create `AuthContext` to store user login state
- Use it across multiple components (Header, Profile, Blogs)
- Create a custom hook `useAuth()` to access context easily

⚙️ Step-by-Step Process (with Hints):

🔹 **Step 1: Create AuthContext**

- Use `createContext()` to create a new context.
- This context will hold the logged-in user's data and loading state.

🔹 **Step 2: Create AuthProvider**

- Inside a new `AuthProvider` component, use `useState` to manage `user` and `loading`.
- Use Firebase's `onAuthStateChanged()` inside `useEffect` to listen to login/logout events and update the state accordingly.
- Return `<AuthContext.Provider>` wrapping `children`.

🔹 **Step 3: Wrap the App with AuthProvider**

- In your main entry point (`main.jsx` or `App.jsx`), wrap your entire app inside the `AuthProvider`.
- This ensures all components inside the app can access the auth context.

🔹 **Step 4: Create Custom Hook `useAuth()`**

- Create a custom hook that uses `useContext` to easily consume the `AuthContext`.
- This allows cleaner and reusable access to the user state.

🔹 **Step 5: Use the Context in Components**

- **In any component (like Header or Profile), use the `useAuth()` hook to get the current user.**
- **Use `user?.uid`, `user?.email`, or `user?.photoURL` as needed.**
- **Optionally show loading state or fallback if context is still fetching user.**

🧩 *Task 6: Create a Custom Hook for File Upload*

Goal: Build a reusable `useFileUpload` custom hook that handles file uploading to Cloudinary and returns useful states (loading, error, URL, etc.) — then use it in profile and blog forms.

Use: `useRef`, `useState`, Custom Hook

✅ **Features:**

- **Create a `useFileUpload` hook that:**
  - **Accepts file input**
  - **Uploads to Cloudinary**
  - **Returns loading, success/error state**
- **Use it in the profile and blog image upload**

⚙️ Step-by-Step Implementation Hints:

🔹 Step 1: Create the Hook File

- **Inside your `/hooks` directory, create a new file: `useFileUpload.js`**

🔹 Step 2: Set Up Hook States

- Inside the hook, use `useState` to manage:
    - `isLoading`
    - `error`
    - `uploadedUrl` (Cloudinary image URL)

🔹 Step 3: Create Upload Function

- Inside the hook, define an `uploadFile()` function that:
    - Accepts a file (image or video).
    - Creates `FormData` and appends the file.
    - Includes your Cloudinary `upload_preset` and `cloud_name`.

🔹 Step 4: Make Cloudinary API Call

Make a `POST` request to Cloudinary's unsigned upload endpoint:

`https://api.cloudinary.com/v1_1/YOUR_CLOUD_NAME/image/upload`

- On success, extract `secure_url` and save to `uploadedUrl`.

🔹 Step 5: Return Hook Values

- Return `uploadFile`, `isLoading`, `error`, and `uploadedUrl` from the hook so that any component can use them.

🔹 Step 6: Use the Hook in Profile Page

- In the profile page, import and use `useFileUpload`.
- On file input change (via `useRef` or `onChange`), call `uploadFile`.
- After successful upload, save the Cloudinary `secure_url` to Firebase under `users/{uid}/profilePic`.

🔹 Step 7: Use the Hook in Blog Page

- In blog post creation form, use the same `useFileUpload` hook.
- Upload the selected image.
- Use the returned URL in the blog post data saved to Firebase under `posts/{id}`.

🌙✅ *Task 7: Add Dark Mode using useContext*

Use: `useContext`, `useState`, `createContext`, `Custom Hook`

⚙️ *Process Steps of Toggle Theme (with Hints):*

Step 1: Create a `ThemeContext`

- Use `createContext()` to define a new theme context.
- This context will store the current theme (`dark` or `light`) and a toggle function.

Step 2: Create a `ThemeProvider`

- **Inside the provider, manage the theme state using `useState`.**
- **Check `localStorage` to persist theme across sessions.**
- **Use `useEffect` *to apply the* `.dark` *class to the root* `<html>` *or* `<body>` *based on the theme state.***

Step 3: Create a Custom Hook (`useTheme`)

- **Write a custom hook to consume the `ThemeContext` so it's easily accessible from any component.**

Step 4: Wrap the App with `ThemeProvider`

- **In your root file (e.g. `index.js` or `main.jsx`), wrap the entire app inside the `ThemeProvider`.**

Step 5: Create a Toggle Button

- **Make a simple button that toggles the theme using the function from `useTheme()` hook.**
- **Optionally show icons like 🌙 / ☀️ based on current theme.**

Step 6: Enable Tailwind CSS Dark Mode (If using Tailwind)

- **In `tailwind.config.js`, enable `darkMode: 'class'`.**
- **Use `dark:` prefix in your CSS classes to apply dark styles.**

Step 7: Apply Dynamic Styling

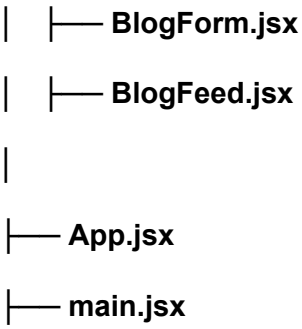
- **Update main layout or wrapper `<div>` to apply dynamic `className="bg-white text-black dark:bg-gray-900 dark:text-white"`**

Step 8: You can also use css root variable

Area	*Marks*
Functional login/auth	**10**
File upload with Cloudinary	**5**
Firebase DB integration	**5**
Proper use of hooks	**25**
useContext + custom hooks	**20**
Optimized rendering (memo, callback, suspense)	**10**
Code organization & composition	**10**
Design	**5**

📁 *Folder Structure (You can organize more and more)*

```
src/
│
├── components/
│   ├── Auth/
│   ├── Blog/
│   ├── UI/
│
├── context/
│   ├── AuthContext.jsx
│   ├── ThemeContext.jsx
│
├── hooks/
│   ├── useAuth.js
│   ├── useFileUpload.js
│   ├── useTheme.js
│
├── pages/
│   ├── Login.jsx
│   ├── Register.jsx
│   ├── Profile.jsx
```


BlogForm.jsx
BlogFeed.jsx
App.jsx
main.jsx