ChatGPT

Private Poetry & Media Sharing Site (Next.js, Node/Firebase, Google Drive)

Project Overview

This project is a **private, password-protected website** for two individuals (Olamide and Adedayo) to share poems, photos, and videos in an intimate setting. Built with **React and Next.js** (for a modern React-based UI and server-side capabilities) and a **Node.js/Firebase backend**, the site will be deployed on **Vercel** for ease of hosting and continuous deployment. Key features include:

- **Secure Login Gate:** A single shared password (known only to the couple) is required to access any content on the site (no individual user accounts or sign-ups)
- **Poetry Collection:** A dedicated section listing all poems by either author, each showing its title, author (Adedayo or Olamide), date written, and the full poem text
- **Media Gallery:** The ability for the couple to upload and view photos and videos. Files are stored in a private Google Drive folder via API integration, keeping storage off the web host
- **Romantic Modern Design:** An elegant, love-inspired visual theme with beautiful typography and a simple, heartfelt UI
- **Responsive Layout:** A mobile-friendly and desktop-friendly design so the experience is smooth on phones, tablets, and laptops alike

Overall, the focus is on **simplicity, emotional tone, and ease of use** – the site should feel personal and special, yet be straightforward to navigate.

Technology Stack and Architecture

Next.js (React) is used for the front-end and site structure. Next.js is ideal here because it supports creating a React UI with server-side rendering and API routes, and Vercel (the hosting platform) is optimized for Next.js projects. We will leverage Next.js for both the **UI components** and for building any necessary **server-side functions** (for example, to handle authentication and form submissions).

Node.js Backend with Firebase: On the backend, we'll use Node.js (via Next.js API routes or Firebase Cloud Functions) to handle any secure operations like verifying the password and interacting with external APIs. Firebase will serve as a backend-as-a-service: - **Firebase Authentication:** (Optional) While we are not using individual accounts, we could use a simplified Firebase Auth mechanism or just a custom check since the requirement is a single shared password. In this case, we'll implement a custom password check without full Firebase Auth, to keep things simple. - **Cloud Firestore (Database):** We'll use Firestore to store the poems (title, author, date, content) as documents, and perhaps references/metadata for uploaded media. This provides a secure, scalable database with minimal setup. Firestore's free tier should suffice for text data (and it allows **client-side or server-side access** as needed). During development we can use Firestore's test mode (open access) for ease [1], and later apply security rules since we have a very controlled use-case (only two users). - **Firebase Cloud Functions:** For heavy lifting tasks like handling the Google Drive uploads,

we can create a Cloud Function (Node.js runtime) that runs with elevated permissions. This keeps secret keys (like Google API credentials) out of the front-end. The Next.js app will call this function (via an HTTP request) when a file needs to be uploaded. Firebase Functions seamlessly integrate with Google services and can use the Firebase SDK on the backend.

Google Drive API: Instead of storing images/videos directly on the web host, uploads will go to a specific Google Drive folder. We'll integrate the **Google Drive API** on the backend (using Google's official Node.js client library `googleapis`). This requires setting up credentials (likely a service account) that has access to the target Drive folder. Using Google Drive gives us ample storage and the couple can also access the raw files via Google Drive if needed. The site will retrieve the files (or their shareable links) from Drive to display to users.

Hosting on Vercel: Vercel will host the Next.js frontend (and its built-in backend API routes, if any). Vercel offers easy Git integration for deployments and handles scaling the Next.js app. We will configure environment variables on Vercel (such as the password, Firebase config, API keys) so that sensitive info is not in the codebase. The site will be private (not indexed) and behind the login gate.

High-Level Architecture: The Next.js app will consist of pages for login, viewing poems, and viewing uploads. When a user accesses the site, Next.js will check for an authentication cookie (set after entering the password). If not present, it will redirect to or show the password entry screen. Once authenticated, the user can navigate the poems section or the media section: - The **poems page** will fetch data from Firestore (either via a direct Firebase client call or via Next.js server-side call using Firebase Admin SDK) and render a list of poems. - The **media page** will list uploaded photos/videos. For each upload, metadata (like a caption, author, or the Google Drive file ID or link) can be stored in Firestore. The actual files will be loaded from Google Drive via shareable links or embedded if possible. - The **upload functionality** (accessible only after login) will have a form for uploading a new poem (text) or a new image/video file. Text submissions can be sent to Firestore directly. File submissions will be sent to a backend route (Next API route or Cloud Function) which handles pushing the file to Google Drive and then returning the result (like the new file's ID or link) to the application, which can then update Firestore with that info.

This architecture ensures that the **frontend remains simple and secure** (all heavy secrets and operations are in the backend), and the **data is persisted** in Google's cloud (Firestore for text, Drive for files). The system leverages managed services to minimize server management.

Implementing Secure Password Protection

To keep the site private, we'll implement a **password gate** that covers the entire site. We do *not* need multi-user accounts or registration – just one shared password that both users know. The implementation will work as follows:

- **Environment-Stored Password:** We define the password in an environment variable (e.g., `PAGE_PASSWORD="YourSharedPass"`), so it's not hard-coded in the public code. Next.js allows defining env vars in a `.env.local` file which we won't commit to Git. This password will only be checked server-side (so it's never exposed on the client). For example, in `.env.local` we might have:

```
PAGE_PASSWORD = "someStrongPassword123"
PASSWORD_COOKIE_NAME = "siteAuth"
```

The cookie name (`siteAuth`) is also set as an env var for consistency [2]. We'll use a cookie to remember that a user has entered the correct password.

- **Login Page or Modal:** When a user hits the site and has not yet authenticated, they should see a **password entry screen**. This could be a dedicated page (e.g., `/login`) or a modal that overlays the content. For simplicity, a standalone login page (`pages/login.jsx`) can be made which contains a form asking for the password. The form will submit the password to an API route (e.g., a Next.js API endpoint at `/api/login`) or use a client-side check.

- **Password Verification API:** On the backend, we create an API endpoint (using Next.js API routes or a Firebase Function) that accepts the submitted password (via POST). This route will compare the entered password with the correct one from env variables. If it matches, it sets an HTTP-only cookie on the response indicating the user is now "authenticated" (this cookie acts like a session flag) [3] [4]. If the password is wrong, it returns an error so the UI can show "Incorrect password" message. For example, using Next.js API route, one can use `res.setHeader('Set-Cookie', ...)` to set a cookie. We mark it `HttpOnly` and perhaps with `SameSite=Lax` and a path of `/` so it covers the whole site.

- **Protecting Pages:** We have a few options to enforce the password protection on all pages:

- A **Next.js Middleware** can intercept requests to any page and check for our auth cookie. Next.js middleware runs on the Edge/Server before rendering pages. If the cookie is missing, we can redirect the user to `/login`. If present, allow them to continue. This approach is efficient for protecting multiple routes in one place [5] [6]. The middleware (in `middleware.ts`) would look for `request.cookies.has(PASSWORD_COOKIE_NAME)` to determine if the user is authenticated [7].
- Alternatively, we could add a check in each page's `getServerSideProps` (if using SSR) or in a layout component. But the middleware provides a neat centralized solution introduced in Next.js 12+.

- We will apply the middleware to all routes except the login page itself. (Next.js allows pattern matching in middleware to exclude certain paths like `/login`.)

- **Skipping Password on Subsequent Visits:** Once the cookie is set, the user won't have to enter the password again on subsequent visits (until the cookie expires or they clear it). We can set a long-lived expiration for the cookie (or no expiry for a session cookie). Additionally, one trick is to allow a special URL query with the password to auto-auth (e.g., visiting `mysite.com?password=YourSharedPass` could set the cookie) [6], but this is optional and must be used carefully (only when sharing the link securely).

In summary, after this setup, when Olamide or Adedayo go to the site: 1. They hit the Vercel-hosted URL. Middleware checks for the `siteAuth` cookie. If not found, they are redirected to the login page. 2. On the login page, they enter the shared password. The site sends it to the backend for verification. If correct, the

backend sets the cookie and responds OK. 3. The login page then redirects them to the home/poems page. Now the cookie is present, so the middleware allows access. All subsequent page loads or refreshes carry the cookie, indicating the user is authenticated. 4. If someone without the password tries to access any page, they'll be stopped at the login screen. The password itself is never exposed on the client; it's checked server-side (or in a serverless function). This simple mechanism effectively locks the site for private use.

(Note: Since this is a simple shared password, it's not as robust as a full user system, but it meets the requirements and avoids complexity. For added security, one could implement rate-limiting on password attempts or use basic auth at the proxy level. However, given the personal nature and limited userbase, this straightforward approach is sufficient.)

Poetry Section Implementation

The **Poetry section** is the heart of the site for sharing written expressions. We will implement it as a page (for example, `/poems` route) that displays a collection of poem entries. Each poem entry will show: - **Title** – the title of the poem - **Author** – a tag or label indicating who wrote it (either "Adedayo" or "Olamide") - **Date** – the date it was written (or posted) - **Content** – the full text of the poem, in a nicely formatted block (preserving line breaks and spacing as needed)

Data Storage for Poems: To allow dynamic updates (adding new poems without redeploying code), we will use **Cloud Firestore** as a database for the poems. Firestore is a NoSQL cloud database ideal for this use-case because it's easy to get started and can be accessed securely from both web and server. We'll create a collection, e.g., `poems`, where each document represents one poem. A sample document structure might be:

```
{
  "title": "My Sunshine",
  "author": "Olamide",
  "date": "2025-02-14",
  "content": "You are my sunshine, my only sunshine...\n[full poem text]"
}
```

We will also store a timestamp or date field in a proper date format so we can sort or display nicely. Firestore automatically gives each document a unique ID, or we can use a slug or title as ID.

Displaying Poems: On the Next.js side, we have a few ways to retrieve and display this data: - We can use **server-side rendering** for the poems page (`getServerSideProps` in Next.js) to fetch the list of poems from Firestore (using the Firebase Admin SDK, which can be initialized with service account credentials on the server). This way, the poems are fetched on each request securely and then the page is rendered. This ensures the content is always up to date. - Alternatively, we can use **client-side data fetching**. For example, when the `Poems` component mounts, use the Firebase Web SDK to query Firestore for all poems. The Firebase web SDK requires initializing with the project config (API key, etc.) and can use Firestore in insecure mode if we've allowed it (or with Firebase Auth if we set up an auth for the two users, but here we might not need to since the site is already gated). - A compromise is to use **Next.js API routes**: e.g., create an API route `/api/poems` that the front-end calls (via `fetch`) to get the poem list in JSON. The API route can

use Firebase Admin to query Firestore without needing any client credentials. This is secure and avoids exposing any database keys to the client. Since our userbase is very small and performance is not a big issue, a simple API call is fine.

For simplicity and security, we'll likely use the **Next.js API route approach or server-side props**: - We will include Firebase service account credentials in the backend (as an environment variable or secret file) to initialize the Admin SDK. (This credential is separate from the Google Drive one; we can generate a Firebase admin service account JSON in the Firebase console and store it securely). - In the API route code, we call Firestore to get all poems documents. We can sort them by date if needed. - Return that data to the front-end. Then render the list of poems.

Poem List UI: The poems could be displayed in a simple list or grid. Perhaps a single-column list is most suitable (for easy reading, like a scrollable list of entries). Each entry might have a styled heading for the title (possibly with an icon or flourish for romance), the author name (maybe color-coded or with a small avatar if we want to incorporate a personal touch), and the date. The content can be shown in a paragraph or `<pre>` style to maintain formatting. We will use elegant typography for the poems – possibly a classic serif font for the body of the poem for readability, and maybe a cursive or decorative font for titles to accentuate the romantic feel.

We'll ensure that each poem is clearly separated (maybe a subtle divider or spacing). If the collection becomes large, we could add filtering by author or a search, but initially just listing them all is fine.

Adding New Poems: We want both users to be able to contribute new poems easily. We can add an interface for this: - A form on the site (perhaps on a page `/add-poem` or even a modal from the poems page) where they can input a title, select author (or it could auto-tag based on who's writing if we had separate logins – but since we don't, we might include a dropdown for "Author" so they can tag it as their own or the other's), and the poem text (possibly a textarea for multi-line content). - When the form is submitted, it will call a backend API (similar to above) that writes the new poem into Firestore. This can be a simple HTTP POST to an API route like `/api/addPoem`. The backend will verify the request (since the site is behind the password, we trust the user, but we could still double-check the author field is one of the two allowed names, etc.), then use the Admin SDK to `add` a document to the `poems` collection [8]. Firestore will timestamp it; we might also explicitly store a `writtenDate` field (which could either be user-provided or the current date). - Once added, the site can refresh the poems list to show the new entry.

Because we're using Firebase for data, all of this can be done **without a dedicated traditional server** – the Next.js serverless functions and Firebase take care of everything. This makes maintenance easier and the site highly scalable (even though we only have two users).

Photo & Video Sharing via Google Drive

One of the core features is allowing Olamide and Adedayo to share **photos and videos** privately. We will implement a **Media Gallery** section for this. The unique part is storing the files on Google Drive and integrating that with the website.

Google Drive Integration Rationale: Vercel (and many web hosts) have limits on file storage and upload sizes. By using Google Drive, we leverage Google's robust storage (potentially free up to certain space, and

easily expandable). It also means the users can have a copy of their media in their own Google Drive account. The site essentially becomes a front-end to view and add to that Drive folder, in a curated way.

Setting Up Google Drive API Access: - We need to create a **Google Cloud project** (if not using an existing one) and enable the Drive API for it [9]. - Create a **Service Account** in the Google Cloud project with access to Google Drive [10]. When creating the service account, grant it the "Drive API –> Drive File access" or appropriate scope. After creation, we'll generate a JSON key for this service account (this is a file with a private key and client email, etc.). - In Google Drive, create a **folder** that will hold all the uploaded media. We can do this in one of the user's Google Drive manually. Then share this folder with the service account's email (service accounts have an email like `<name>@<project>.iam.gserviceaccount.com`). By giving edit access to the service account on this folder, the service account can upload files into it. (Alternatively, one could use a service account's own Drive if part of a Google Workspace domain, but sharing a folder from a personal drive is simplest). - Note the **Folder ID** of this Google Drive folder (it's the long ID in the URL when you open the folder on drive.google.com). We will use this ID in our code so that all uploads go into that specific folder.

Uploading Files (Backend Logic): We will write a backend function to handle uploads: - We'll use the **Google Drive API Node.js client** (`googleapis` library). Using the service account credentials, we'll authorize a JWT client for Drive. For example, using the key JSON, we can authenticate like:

```
const {google} = require('googleapis');
const auth = new google.auth.GoogleAuth({
    credentials: serviceAccountKey,
    scopes: ['https://www.googleapis.com/auth/drive.file']
});
const drive = google.drive({ version: 'v3', auth });
```

This gives us a `drive` object to call Drive API methods. - We will create an API route or Firebase Function (e.g., `uploadFile`) that accepts the file (from an HTML file input on the front-end). Likely, the file will be sent as binary (multipart form data). We might use an `<input type="file" multiple>` for users to select files, and then use JavaScript (Fetch or Axios) to send the file to our API endpoint. - The backend function receives the file stream or file data. We then call `drive.files.create` with the appropriate metadata and media body. For example, in Node.js:

```
const fileMetadata = {
  name: originalFilename,
  parents: [FOLDER_ID]    // ID of the Drive folder to upload into
};
const media = {
  mimeType: fileMimeType,
  body: fileReadStream    // stream or buffer of the file
};
const response = await drive.files.create({
  resource: fileMetadata,
  media: media,
```

```
  fields: 'id, name, webViewLink, webContentLink'
});
```

This will upload the file to Drive. If successful, we get a file resource in response. We requested fields like `id` and links. The `id` is crucial (unique file ID in Drive). The `webViewLink` and `webContentLink` are useful: `webViewLink` is a URL to view the file in Google Drive's viewer, and `webContentLink` is a download link. (Note: these links might only be available if the file's sharing settings allow it.) - We may need to set the file's permission to be viewable by the two users (and by the site). Since the site is private, we *could* keep the files private (only accessible via the service account). But then to display them, we would have to have the backend fetch the file every time. A simpler approach is to use the Drive API to make the file **shareable via link** (i.e., "anyone with the link can view"). We can automatically set the permission right after upload:

```
await drive.permissions.create({
    fileId: response.data.id,
    requestBody: { role: 'reader', type: 'anyone' }
});
```

This is the equivalent of clicking "Share -> Anyone with link can view". By doing this in the API, we ensure the returned `webViewLink` or a constructed link can be accessed by our front-end [11]. (If we skip this, the file would only be accessible to the service account or the owner's account, causing permission issues when embedding.) - After upload (and setting permission), we get a shareable link or at least the file ID. We will then **store a reference** to this file in our app's data. For instance, we can add a document in Firestore like:

```
{
  "fileId": "<Drive file ID>",
  "name": "<filename>",
  "type": "image" [] "video",
  "author": "Adedayo",
  "uploadedAt": Timestamp
}
```

Or we might store the `webViewLink` directly. Storing in Firestore allows us to easily list all uploaded items in the gallery without calling the Drive API every time (which would require authentication each time). It also lets us annotate the file with an author or caption if needed. - The backend returns success (maybe the new file's ID or link) to the front-end, which can then update the UI (e.g., add the new photo to the gallery list).

Displaying Photos and Videos: On the gallery page, we want to show the images and playable videos. - For **images**: We can use the Drive file ID to construct a URL that serves the image. Google Drive has a URL format for raw content via its `uc` (upload content) endpoint. For example:

```
https://drive.google.com/uc?export=view&id=<FILEID>
```

This URL will display the image if it's shared publicly. We can set that as the `src` of an `<img>` tag. Alternatively, since we set "anyone can view", we might use the `webContentLink` (which typically is a link to download). The `uc?export=view` is a trick often used to embed images from Drive. - For **videos**: We have a couple of options. One convenient way is to use an **iframe** with the Google Drive preview. Google Drive allows embedding a video using an iframe with a special URL. For example:

```
<iframe src="https://drive.google.com/file/d/<FILEID>/preview" width="640" height="480" allow="au
```

This will use Google's video player UI inside the iframe. Another approach is to use an HTML5 `<video>` tag and set the source to the `uc?export=preview&id=<FILEID>` URL which can serve the video stream [12]. The Stack Overflow community suggests that using the `uc?export=preview` link works well for embedding videos in a video tag or iframe [12]. We'll have to ensure the MIME type is correct (e.g., Drive will serve an MP4 with `type="video/mp4"` in the `<source>` tag as shown in the example). - We will likely go with the iframe `.../preview` method for videos, as it's straightforward and supports Google's player (with controls, fullscreen, etc.). For images, a simple `<img>` is perfect. - The gallery page can show thumbnails or a grid of images. We might display images as small thumbnails that can be clicked to view larger (maybe using a lightbox effect). Videos might be represented by a thumbnail with a play icon. However, given the simplicity requirement, we could also just list them in a column with perhaps a small preview and the user can scroll.

Uploading UI: On the frontend, the **upload interface** for media will allow selecting a file: - We can have an "Upload Photo/Video" button that opens a file picker. Possibly separate buttons or a toggle to tag whether it's photo or video, but we can actually detect file type by MIME after selection. - After the user selects a file, we might show a preview or the file name, and an "Upload" button to confirm. - When they click upload, we call our backend (API route or cloud function) to handle it. We might show a loading indicator while the file transfers (especially for videos, which can be large). - Once done, we can either refresh the gallery list or directly append the new item to the UI.

File Size & Performance: Since videos can be large, we should be mindful of upload limits. Vercel serverless functions have a body size limit (~4.5 MB by default). Large videos would exceed that. If large uploads are expected, using a Firebase Cloud Function (which can handle larger payloads when called via a callable function or endpoint) is better. Alternatively, we can implement a resumable upload where the front-end uploads directly to Google Drive using a signed URL, but that's more complex. Given the scope, using a Firebase Function for uploading might be the safest route for big files. The Firebase Function URL can be called via an HTTP request from the front-end (with the file in the request body). We'd configure the function with a higher timeout and memory if needed for big files. This way, the heavy lifting is offloaded to Google's servers rather than Vercel.

Security: Because we are exposing files via shareable link, technically anyone with the exact link could access a particular file. However, since the links (Drive file IDs) are only shared between the two users and not public, and the site itself is password-protected, this is reasonably secure for personal use. If desired, we could skip making the file public and instead stream it through our backend (i.e., the site could fetch the file using the Drive API with credentials whenever someone wants to view it). But that would complicate viewing and put load on our server side. So we'll accept the small risk for the benefit of simplicity – after all, the links won't be indexed or guessable (they are long random IDs).

In summary, the media sharing flow is: 1. User navigates to “Gallery” page (after login). 2. The page on load fetches from Firestore the list of media items (each with info like fileID, type, author, etc.). 3. It then displays each item appropriately (images with `<img>` thumbnails, videos maybe with a thumbnail or just an iframe ready to play). 4. The user can click “Upload” to add a new item. They choose a file, the file is sent to the server, the server uploads to Drive and returns the file info, and the page updates to show the new item. 5. Both users can see the uploaded content immediately. They can also go to the Google Drive folder in their Drive account (if they have access) to see the raw files.

UI/UX Design: Romantic & Modern Theme

The visual design will set a **romantic and modern tone** for the site, making it feel like a personal, heartfelt space rather than a generic app. Here are key design decisions and elements:

- **Color Scheme:** We’ll choose a soft, warm color palette. Romantic themes often use colors like gentle pinks, peach, lavender, or cream, complemented by neutral backgrounds. For a modern touch, we might use plenty of white or very light background to keep it clean, with accent colors (maybe a soft rose red or purple) for headers or highlights. The goal is an elegant look, not overly flashy. Possibly a subtle background image or texture (for example, a faint floral pattern or watercolor texture) could be used on the landing page or margins to enhance the romantic feel without distracting from the content.

- **Typography:** Typography is crucial here. We want **elegant, readable fonts**. A good approach is to use a combination of one decorative font and one clean font:

- For titles (e.g., poem titles or the site title “Our Little Library of Love” or whatever we name it), a **cursive or script font** can convey romance. For example, *Annabel Script* is a cursive typeface noted to give a romantic look for headers [13]. Other script fonts or calligraphy styles could work (perhaps something like Great Vibes, Dancing Script, or a classic like Lucida Calligraphy). We must ensure it’s still readable and not too ornate.
- For body text (poem content, descriptions), use a **serif font** that’s easy on the eyes and has a touch of classic elegance (serif fonts often feel more literary, which suits poems). An example might be *Georgia* or *Garamond* or a Google Font like *Libre Baskerville* or *Playfair Display*. Alternatively, a clean sans-serif for body and serif for titles can also work; however, sans-serif might feel too modern/minimalist. A combination like a cursive header, serif body is likely ideal for that romantic literary vibe.
- We will load these fonts via CSS (using Google Fonts CDN if available for the chosen fonts).

- Font sizes will be comfortable for reading – not too small. Perhaps body text at ~1rem (16px) or slightly larger, and titles significantly larger for hierarchy.

- **Layout and Imagery:** The layout will be relatively simple (we’re not making a very complex site structure). Possibly a top navigation (for switching between “Poems” and “Gallery” and maybe a home or about), though even that might be minimal. We could have a welcome message on the home page with a nice quote or dedication. Each section (poems, media) should have a clear heading.

- We may incorporate small decorative flourishes: for instance, a heart icon next to the site title or as bullet points, or a divider that is a cute graphic (like "~ * ~").
- The **poems page** might just be text content, so styling the text nicely is key (maybe using CSS for first-letter styling, etc.). We could also include small author photos or icons near the author name to personalize it (if the couple is okay with that).
- The **gallery page** will be more visual. We should ensure images are nicely sized. Possibly use a grid layout for photos (responsive grid that adjusts number of columns based on screen width). For videos, maybe show a poster image or simply display the video player in the grid.

- Consider adding a slight **hover effect** on gallery items (like scale up or a shadow) to make the UI feel interactive and modern.

- **Emotional Tone:** We want the site to evoke feelings. We might add a tagline on the landing page such as "Welcome to our private corner of the world, filled with our poems, memories, and dreams." The language throughout can be warm and personal. Button texts could be something like "Share a New Memory" instead of a bland "Upload". These little touches make the UX feel custom to the couple.

- If desired, use some personal images (like a faint background photo of the couple, blurred and low-opacity, as a backdrop). But that depends on their preference for privacy and aesthetics.

- **Modern Design Practices:** Ensure plenty of whitespace and a clean layout – this prevents the romantic theme from becoming cloying or cluttered. Use modern CSS techniques (flexbox, grid) to layout content so that it looks good on all screens. We'll also implement a dark mode only if required; likely not needed unless the users want it – a consistent light theme might suffice.

In CSS terms, we will create a global stylesheet or use styled-components/CSS-in-JS (Next.js supports global CSS or CSS modules). We'll define the color variables and fonts at the top. We might also use a UI library for basic components (like Chakra UI or Material-UI) but heavily customized to fit the theme, or just build our own simple components given the simplicity (labels, buttons, text areas, etc.). A library like Chakra could expedite responsiveness and theming, but a custom approach gives more freedom for a unique style.

Finally, we'll test the design on different devices to ensure the font sizes and colors are legible and the feeling is as intended. The outcome should be a website that immediately gives the vibe of a **personal love journal**.

Responsive Design for Desktop and Mobile

From the start, we will adopt a **mobile-first responsive design** strategy. This ensures the site looks and works great on smartphones (which Olamide and Adedayo might use to read or post on the go), as well as on larger screens like tablets and laptops.

Key considerations for responsiveness: - Use flexible layouts: **CSS Flexbox and Grid** will be used to arrange elements so they can stack or rearrange naturally on smaller screens. For example, on desktop the poems list might show entries side by side in two columns (if we wanted a two-column layout), but on mobile it should collapse to one column. - Set appropriate **breakpoints** in CSS. We might decide that any screen below 768px (typical tablet width) will use the mobile layout: e.g., navigation turns into a burger menu (if we

have multiple pages), grid columns collapse, maybe font sizes adjust slightly. On very large screens, we might increase padding to avoid overly long line widths for the poems (for readability, it's often good to limit line length). - **Media queries:** For instance, `@media (max-width: 600px) { ... }` could be used to apply mobile-specific styles. We will ensure that images and iframes are fluid (e.g., set max-width: 100% so they shrink to fit smaller screens). - Next.js by default includes a responsive image component `<Image>` which can optimize images, but since we are embedding from Drive, we might not use Next's Image for those external links. We'll just ensure to give them proper width/height or use CSS to make them responsive. - **Testing the UI**: We will test the site on a mobile viewport to fine-tune things like touch targets (buttons should be easily tappable), maybe use slightly larger text on small devices to account for readability. The design will avoid any hover-only interactions (since on mobile hover doesn't exist) – any interactive element will also be clickable/tappable. - The login modal/page will also be responsive (centered form on desktop, maybe full-screen form on mobile with bigger inputs). - Performance on mobile: We keep the site lightweight. Next.js helps by code-splitting pages. We will also possibly use Firebase's lite SDKs (Firestore has a lite SDK for web that is smaller, if we only need basic reads/writes) to reduce JS bundle. This ensures fast load on mobile networks.

Overall, by leveraging modern CSS and testing across viewports, we ensure the site is **usable and attractive on all screen sizes**. The emotional impact and usability should carry over whether the couple is snuggled together reading on an iPad or apart on their individual phones sharing things.

Firebase Configuration and Backend Integration

We've touched on Firebase and backend bits in earlier sections; here's a focused look at setting up those pieces and ensuring everything works securely:

- **Firebase Project Setup:** First, create a Firebase project (through the Firebase console). In that project, enable **Cloud Firestore** (create a database). We can start Firestore in **test mode** for initial development, which means it's open for reads/writes without auth [1] – since our site is gated, this might actually be okay even in production for this limited use-case, but it's better to add some basic security rules later. For example, we could restrict the `poems` and `media` collections to only allow writes if a certain secret field matches (though without Firebase Auth, rules are tricky – we might just keep it open but rely on obscurity).

- **Firestore Data:** Create two collections: `poems` and `media` (or `uploads`). We don't necessarily have to pre-create them; adding a first document via code will create them. But organizing ahead helps. NoSQL means documents can have varying fields, but we'll keep a consistent structure as defined earlier.

- **Firebase Web Config:** Add a Firebase Web App in the Firebase console to get the config (API key, project ID, etc.). This is needed if we use Firebase directly in the front-end. We will put this config in our Next.js app (in `.env.local` or directly in code). Remember, Firebase API keys are not secret (they are ok to expose), but we still might store in env for convenience. According to Next.js conventions, any variable used on the front-end must be prefixed with `NEXT_PUBLIC_` [14]. So we might have:

```
NEXT_PUBLIC_FIREBASE_API_KEY=<...>
NEXT_PUBLIC_FIREBASE_PROJECT_ID=<...>
```

etc., and initialize the Firebase client SDK with those.

- **Firebase Admin (Server) Setup:** For secure server-side operations, we'll use the Firebase Admin SDK in our Node environment. We need service account credentials for this as well. Easiest is to go to Firebase project settings -> Service Accounts, and generate a new private key. This gives a JSON similar to the Google Drive one but for Firebase admin. We add this JSON (or the necessary fields) as an environment variable on Vercel (since we can't include it in code). For example, we can base64 encode the JSON and set it, or store each field (not recommended, better to store the whole JSON string as one env var).

- Alternatively, since we are already using a Google service account for Drive, we could theoretically use one service account for both Drive API and Firestore access. If we add our service account (the one for Drive) to the Firebase project as an Editor, it can act as admin. Then we could use that same JSON for Admin SDK. This might be convenient – one less credential to manage. We'd just ensure the scopes/permissions cover both. It's a bit advanced but doable.

- **Cloud Functions for Firebase:** Install Firebase Tools (if developing locally) to deploy functions. Initialize a Functions project within our codebase or separately. We'll write a function `uploadToDrive` (as described). Mark it as an HTTPS function. Test it locally with Firebase emulators if needed. Once ready, deploy it via `firebase deploy --only functions`. Firebase will give us a URL for the function (something like `https://us-central1-YOURPROJECT.cloudfunctions.net/uploadToDrive`). We will use that URL in the Next.js app to call the function. (We might also need to enable CORS on that function or in the request since our site is on a different domain (Vercel domain) – or we can add the domain to authorized domains in Firebase config.)

- Note: We could also avoid Firebase Functions by using Next.js API routes on Vercel for uploading, but as discussed, large file uploads might be an issue on Vercel. If the expected media are small (<5MB), Next.js API would suffice. If not, Functions are safer. We could mention both options in documentation and choose one.

- **Google API Credentials in Firebase Functions:** In the Cloud Function code, we'll include the service account credentials for Drive. We should not hardcode the JSON there either. Instead, use Firebase Functions config (via `firebase functions:config:set drive.key="...json content..."`) or use Google's secret manager. For simplicity, one might also upload the JSON as part of code but that's not secure. We'll aim to use config. Then in the function, read the key and initialize the Google Drive API client as shown earlier.

- **Testing the Integration:** After setup:

- Test adding/retrieving a Firestore document (maybe write a sample poem via Firebase console and see if our Next.js can read it).

- Test the upload function with a small file from a local environment (perhaps using Postman or a simple HTML form).
- Ensure the file appears in Drive and our Firestore gets updated with the reference.
- Then test end-to-end: open the deployed site, enter password, go to upload, add a file, see it show up.

All these moving parts (Next.js, Firebase, Google API) require careful configuration, but once set, they work together nicely. The **Firebase codelab for Next.js** shows similar patterns (auth, storage, Firestore usage) where images are uploaded to Firebase Storage and URL saved to Firestore [15] – our case is just swapping Firebase Storage with Drive.

By configuring Firebase and backend this way, we ensure that: - We have a reliable datastore (Firestore) for text and metadata. - We have a secure way to execute privileged code (Functions) for external API calls. - The front-end remains mostly focused on presentation and simple calls, keeping sensitive logic hidden.

Deployment on Vercel (Setup Instructions)

Deploying the app to Vercel will make it live on the internet (but still locked behind the password). We will go through the steps to set up continuous deployment on Vercel:

1. **Prepare the Repository:** Make sure our Next.js project is in a Git repository (e.g., on GitHub, GitLab, or Bitbucket). Vercel integrates with these. The project should include all our pages (`pages` directory for Next.js), the Firebase config (maybe as files or we rely on env vars), etc. We will not include any secret files (no `.env.local` in git). Instead, we will use Vercel's environment variables feature for those.

2. **Vercel Account:** Create a Vercel account (if not already) and link it to the Git provider. On Vercel's dashboard, click "Import Project" and select the repository. Vercel should auto-detect it's a Next.js app. Choose the default project settings or as needed.

3. **Environment Variables on Vercel:** Before deploying, we need to configure the env vars so that the build and runtime have our secrets/keys:

4. In the Vercel dashboard, go to **Project Settings > Environment Variables**. Here, add all necessary keys:
    - `PAGE_PASSWORD` (the shared password for the site login). Mark it as **Secret** (it will be by default, since it's not starting with `NEXT_PUBLIC_` it won't be exposed to browser [14]).
    - `PASSWORD_COOKIE_NAME` (e.g., "siteAuth" as we chose).
    - Firebase config: `NEXT_PUBLIC_FIREBASE_API_KEY`, `NEXT_PUBLIC_FIREBASE_AUTH_DOMAIN`, `NEXT_PUBLIC_FIREBASE_PROJECT_ID`, etc., as provided by the Firebase console for the web app. (Even though our site might not use all of them if not using Auth or Storage, including them doesn't hurt.)
    - Firebase Admin credentials: e.g., we might have something like `FIREBASE_SERVICE_ACCOUNT` containing the JSON. Or individual ones like `FIREBASE_PROJECT_ID`, `FIREBASE_CLIENT_EMAIL`, `FIREBASE_PRIVATE_KEY` from

that JSON. The Medium article suggests adding them in .env and accessing via `process.env.KEY_NAME` [16].
    - Google Drive Service Account JSON or parts of it: we might set an env var for `GOOGLE_SERVICE_ACCOUNT` containing the JSON string of the key. Or at least the path to a key file if we upload it, but Vercel doesn't have a file system for secrets at build, so better to store the content or use something like Vercel's Secret management (but that's similar to env).
    - Google Drive Folder ID (so our code knows where to upload): e.g., `DRIVE_FOLDER_ID=<the ID>`.
5. After adding each variable, Vercel allows scoping to "Production" or "Preview" etc. We'll likely apply to all environments or at least Production. Make sure to **Save** them. (And we'd **uncheck any option to expose them automatically** if using older Vercel UI, but in the current UI, only `NEXT_PUBLIC_` ones get exposed by default [17].)

6. **Double-check:** The variables in Vercel should match the names expected in code. For example, if our code expects `process.env.NEXT_PUBLIC_FIREBASE_API_KEY`, we must name it exactly that in Vercel.

7. **Deploy the App:** Once env vars are set, we can trigger a deployment. If using GitHub integration, just pushing the latest code to the main branch will cause Vercel to build and deploy. On first import, Vercel might do a build automatically. The build logs should show it pulling the env vars. The Next.js app will be compiled and output to `.next` folder, which Vercel serves. The API routes become serverless functions in Vercel automatically.

8. If any env var is missing, the build might warn or fail (for example, if our code tries to use `process.env.SOMETHING` that isn't defined). So ensure all needed ones are there.

9. Vercel will assign a default domain like `your-project-name.vercel.app`. We can use that, or add a custom domain if desired (not necessary for private use, but a nice touch could be a custom domain like `olamide-adedayo.love` – this is optional and requires domain registration).

10. **Deploy Firebase Functions:** If we used Firebase Cloud Functions for the upload, that's deployed via Firebase, not Vercel. So we do that separately. Using the Firebase CLI (`firebase deploy`), deploy the function. After deploy, copy the function's URL. Then, in our Next.js code (or config), ensure the URL is used for the upload calls. We might set it as an env var too (like `NEXT_PUBLIC_UPLOAD_FUNCTION_URL`). If we change that, we'd re-deploy the Next.js to update it.

11. Alternatively, if using Next.js API for uploads, no separate deploy needed beyond Vercel.

12. **Testing Production:** Visit the Vercel app URL. You should see the login screen. Enter the password (make sure it's the one we set in Vercel env). If it works, you get in. Test viewing poems (maybe add some sample poems in Firestore and see if they load). Test uploading an image and video. Check that after upload, the image appears in Drive and on the site.

13. If something fails, check Vercel function logs (Vercel provides function logs for API routes) or Firebase function logs for errors. Common issues might be incorrect credentials or missing

permissions (e.g., service account not having Drive access, or Firestore rules preventing read/write if not in test mode).

14. **Optimization & Maintenance:** Ensure that the Firebase project is on the free Spark plan (if usage is low, it should stay free). Monitor usage occasionally. Vercel's free tier is also usually sufficient for a personal site like this (we won't exceed the bandwidth or function invocation limits with just two users). Set up the Firebase and Vercel CLI on your local machine for any future updates. Document the process for the couple in case they need to update the password (they would update the env var and redeploy) or migrate data.

By following these deployment steps, we will have a live, secure website. Anytime we push new code (e.g., to tweak the design or fix bugs), Vercel will auto-deploy updates. The site will remain behind the password as long as the middleware and checks are in place.

Environment Variables Recap: It's worth emphasizing the handling of env vars since they are critical: - Locally, we use a `.env.local` file for testing (this file contains e.g., `PAGE_PASSWORD`, Firebase configs, etc., and is listed in `.gitignore`). We can run `npm run dev` to test the Next app locally with those variables. - On Vercel, we configured the same vars in the dashboard. As a result, the deployed environment mimics our local dev environment. The Medium article on Next.js + Vercel deployment highlights that after adding the variables in Vercel, you just redeploy and it works [17]. - We should avoid using any secret directly in front-end code. Only `NEXT_PUBLIC_*` ones (like the Firebase API key) will be exposed, which is expected. The actual password or private keys remain server-side only.

Finally, once deployed, the couple can use the site by navigating to the URL, entering the password, and enjoying their private repository of poems and memories. The site can be continuously expanded (they could add new sections or features later, like an audio messages section or a calendar of memories, etc., using the same tech stack).

Conclusion

We have designed a comprehensive solution for a **private, romantic website** using **Next.js, Node.js, Firebase, and Google Drive**. The site will be visually appealing with a modern yet intimate design, and will function smoothly across devices. By using Next.js and Vercel, we get a fast and reliable web app with minimal server maintenance. Firebase provides the backbone for storing data and running secure server-side code (functions), while Google Drive handles large media storage efficiently. The end result is a **password-protected digital space** where Olamide and Adedayo can lovingly share and preserve their poems, pictures, and videos, with full control and privacy. By following the setup and deployment steps above, one can implement this project and deploy it successfully on Vercel for the couple to start using and cherishing. Happy coding and best wishes to the users of this special site!

Sources:

- Next.js Official Docs – guidance on authentication and middleware [5] [6]
- *Medium – Password protect page in Next.js* (Ziya Fenn) – describing using Next.js middleware, env variables, and cookies for simple password gating [2] [7]

- *Dev.to – Upload Files on Drive With Node.js* (Arjun Tripathi) – steps for Google Drive API setup and Node.js upload example [9] [10] [18] [19]
- Stack Overflow – using Drive API to set file permissions for shareable link [11] and embedding Drive videos in a webpage [12]
- Firebase Documentation – Firestore setup and usage, Firebase + Next.js integration (FriendlyEats example) [15]
- *Medium – Setting up Environment Variables and Hosting on Vercel* (Sanyam) – how to configure Next.js env and Vercel deployment settings [14] [17]
- *Elegant Themes* blog – *15 Elegant & Modern Fonts for Web Design* – suggestions on typography for a romantic/elegant look (e.g., Annabel Script) [13].

---

[1] **Quickstart: Create a Firestore database by using a web or mobile client library | Google Cloud**
https://cloud.google.com/firestore/docs/create-database-web-mobile-client-library

[2] [3] [4] [5] [6] [7] **Password protect page in Next.js. In this article, we'll delve into the... | by Ziya Fenn | Medium**
https://medium.com/@ziyafenn/password-protect-page-in-nextjs-5820cd7078ae

[8] [15] **Integrate Firebase with a Next.js app**
https://firebase.google.com/codelabs/firebase-nextjs

[9] [10] [18] [19] **Upload Files on Drive With Node.js - DEV Community**
https://dev.to/mearjuntripathi/upload-files-on-drive-with-nodejs-15j2

[11] **javascript - Nodejs Google Drive API Shareable link - Stack Overflow**
https://stackoverflow.com/questions/47447816/nodejs-google-drive-api-shareable-link

[12] **javascript - How to embed videos from Google drive to webpage? - Stack Overflow**
https://stackoverflow.com/questions/40951504/how-to-embed-videos-from-google-drive-to-webpage

[13] **15 Elegant & Modern Fonts for Web Design**
https://www.elegantthemes.com/blog/resources/15-elegant-modern-fonts-for-web-design

[14] [16] [17] **Setting up Environment Variables and Hosting on Vercel | NEXTJS | by Sanyam | Medium**
https://medium.com/@sanyamm/setting-up-environment-variables-and-hosting-on-vercel-nextjs-54194eae0165