# **טכנולוגיות אינטרנט מתקדמות - 61776 (WEB**)

**הגשת פרויקט**

**<collaborative digital storytelling platform - NorthStory> < B19><19>**

|  |  |
| --- | --- |
| **שם חבר.ת הצוות** | **תז** |
| שלומי זריהן | 206877698 |
| נוי בן עזרא | 211565130 |
| אלכס גרומן | 320648918 |
| יובל כץ | 207504952 |
| שחר נחום | 323964817 |
| איתן זרבל | 315036178 |

**קישור לתיקיית גיט ציבורי-** [קישור גיט](https://github.com/Shlomi645/FinalWebTeam19/tree/main)

**קישור לאתר-** [NorthStory](https://final-web-team19.vercel.app/)

**NorthStory** היא מערכת קהילתית אינטראקטיבית, שנולדה מתוך הצורך לתת מענה חברתי ורגשי לסטודנטים שפונו מצפון הארץ בעקבות **מלחמת חרבות ברזל**. המערכת שואפת ליצור מרחב בטוח, תומך ומכיל שבו כל סטודנט יכול לשתף, להישמע ולהרגיש שייך, גם מתוך מציאות לא פשוטה של ניתוק מהבית והקהילה.

אורח:

כאשר משתמש נכנס לאתר מבלי להתחבר, הוא פועל כאורח עם גישה מוגבלת. המשתמש נחשף לדף הבית, תכני פתיחה ואלמנטים המציגים את מטרת הפלטפורמה. בנוסף, האורח יכול לקרוא מידע כללי על האתר דרך עמוד “About Us” ולנווט בין הדפים הציבוריים של המערכת. עם זאת, ברגע שמעוניין להשתתף באופן פעיל מופנה לתהליך הרשמה או התחברות.

סטודנט רשום:

לאחר התחברות מוצלחת, נפתחות למשתמש יכולות מלאות באתר. הוא יכול ליצור פוסטים חדשים בפורום, לבחור נושא רלוונטי לפרסום ולהחליט אם לפרסם את הפוסט בצורה אנונימית. כל פוסט יכול לכלול גם תמונה שמועלת ישירות. בנוסף, המשתמש יכול לערוך את הפרופיל האישי שלו, לעדכן פרטים אישיים ולהוסיף תמונת פרופיל. מעבר לכך, עומדות לרשותו מעבר בין מצב מוחשך למואר, קבלת משפטי מוטיבציה מהמערכת וכפתור להתנתקות בטוחה מהמערכת.

**מימוש טכנולוגי לפי חלקי המערכת**

Frontend- השתמשנו ב־(Next.js) לבניית צד הלקוח בפורמט מודרני מבוסס רכיבים, עם תמיכה בפריסה דינמית ושימוש ב־ Hooks כמו useState, useEffect, useContext. הממשק כולל קומפוננטות מוכנות מui.shadcn כמוCarousel, Button, Card, Accordion, Avatar המנוהלות בתיקיית components .

Backend- פעולות כמו התחברות, שליפת פוסטים, העלאת פוסט חדש ועדכון פרופיל מתבצעות בעזרת Firebase כולל Authentication ו־Firestore . קריאות נתונים מבוצעות דרך API של Firebase תוך שימוש בפונקציות getDoc, setDoc, addDoc .

Database- נעשה שימוש ב־(Firestore) מסד נתונים של NoSQL לאחסון מידע כמו משתמשים, פוסטים ופרופילים. הנתונים מאורגנים באוספים (collections) כמו users ו־posts ונשלפים לפי מזהה המשתמש (uid) או לפי סדר זמנים (timestamp) כJSON .

Styling - השתמשנו ב־(Tailwind CSS) לעיצוב רספונסיבי ונקי, עם שילוב של מחלקות utilit .

Storage- תמונות פרופיל ותמונות פוסט מועלות ל־(Cloudinary) דרך API חיצוני בשיטת POS התמונות נשמרות בענן ומוחזר לינק לתמונה המאוחסנת.

מהנדסת מערכת - נוי בן עזרא

בתחילת הפרויקט ישבנו יחד כדי להגדיר מטרות, להבין את הדרישות ולבנות תכנית עבודה מוסכמת. במסגרת תפקידי כמהנדסת מערכת, הייתי אחראית על תיאום בין חברי הקבוצה, הקצאת המשימות, מעקב אחר ההתקדמות ומתן תמיכה טכנית בעת הצורך. העבודה התנהלה תוך שיתוף פעולה שוטף, אינטגרציה בין חלקי המערכת ופתרון בעיות בזמן אמת. לאחר השלמת המשימות חיברתי בין כלל התוצרים, ווידאתי תקינות, אחידות ועמידה בדרישות הקורס.

|  |  |  |
| --- | --- | --- |
| **שם חבר הצוות** | **משימות שהוקצו** | **משימות שהושלמו** |
| נוי בן עזרא  תז: 211565130 | use case דף השער | use case דף השער |
| שלומי זריהן  תז: 206877698 | תיק למתכנת דף השער | תיק למתכנת דף השער |
| אלכס גרומן  תז: 320648918 | תיק משתמש | תיק משתמש |
| שחר נחום  תז: 323964817 | ארכיטקטורה מעודכנת של האתר | ארכיטקטורה מעודכנת של האתר |
| יובל כץ  תז: 207504952 | הצגת רשימת דרישות המערכת. | הצגת רשימת דרישות המערכת. |
| איתן זרבל  תז: 315036178 | ארכיטקטורה מעודכנת של האתר | ארכיטקטורה מעודכנת של האתר |

**דרישות:**

דרישות פונקציונליות (Functional Requirements):

|  |
| --- |
| The system provides login, registration, and logout functionality for users |
| The system allows users to create and publish posts with text, image, and category selection |
| The system allows users to post content anonymously |
| The system allows users to comment on posts and express support via likes |
| The system allows users to update their profile details and upload a profile picture |
| The system allows uploading and displaying post images using cloud storage |
| The system provides daily motivational quotes using an integrated AI prompt system |
| The system allows filtering and browsing posts by study group or category |
| The system provides a responsive interface for both mobile and desktop devices |
| The system allows switching between dark mode and light mode |

דרישות לא פונקציונליות (Non Functional Requirements):

|  |  |
| --- | --- |
| The system uses HTTPS for secure communication and encrypts all user authentication and stored data | Security |
| Anonymous posting is supported and hides all identifiable user data | Security |
| The system leverages Firestore with dynamic auto-scaling and cloud storage | Capacity |
| Thousands of stories, comments, and media can be stored and retrieved in real-time | Capacity |
| Compatible with all modern browsers: Chrome, Firefox, Safari, Edge | Compatibility |
| Fully supports Windows, macOS, Linux, Android, and iOS | Compatibility |
| Target system uptime is 99.9%, with automatic Firebase reconnection when disconnected | Reliability |
| Backend is built on Firebase cloud-native architecture with scalable horizontal structure | Scalability |
| Users from different faculties can collaborate via posts in shared discussion boards | Interoperability |
| Modular codebase built with Next.js and React, enabling easy updates and feature expansion | Maintainability |
| Continuous deployment via GitHub with auto-deploy on Vercel | Maintainability |
| Visual feedback provided via animations, toast messages, spinners, and hover effects | Usability |
| Responsive design with mobile-first layout and drawer navigation | Usability |
| Users can upload profile and post images via Cloudinary API | Flexibility & Integrability |
| The system supports uploading and linking of academic files (e.g. PDFs) | Flexibility & Integrability |
| Accessible as both a mobile website and progressive web app | Portability / Accessibility |

דרישות ממשק חיצוניות:

1. User Interface Requirements:

|  |  |
| --- | --- |
| **Element** | **Description** |
| Home Feed | Displays scrollable list of posts with filters by category and faculty |
| Navigation Bar | Fixed top bar with links to Home, Profile, About Us, and Contact pages |
| Create Story Page | Includes input for title, content editor with emotional prompts, media upload buttons |
| Story Detail View | Displays full story content, image attachments, comment section, and like/support interactions |
| Mobile View | Responsive design with drawer-style navigation and collapsible menus |
| Typography | Uses clean sans-serif fonts (e.g., Inter, Roboto) |
| Color Scheme | Calming tones (soft blue, pastel green) with optional dark mode |
| Visual Feedback | Hover effects, loading spinners, toast messages for actions |
| Accessibility | High contrast mode, readable font sizes, keyboard-friendly navigation |

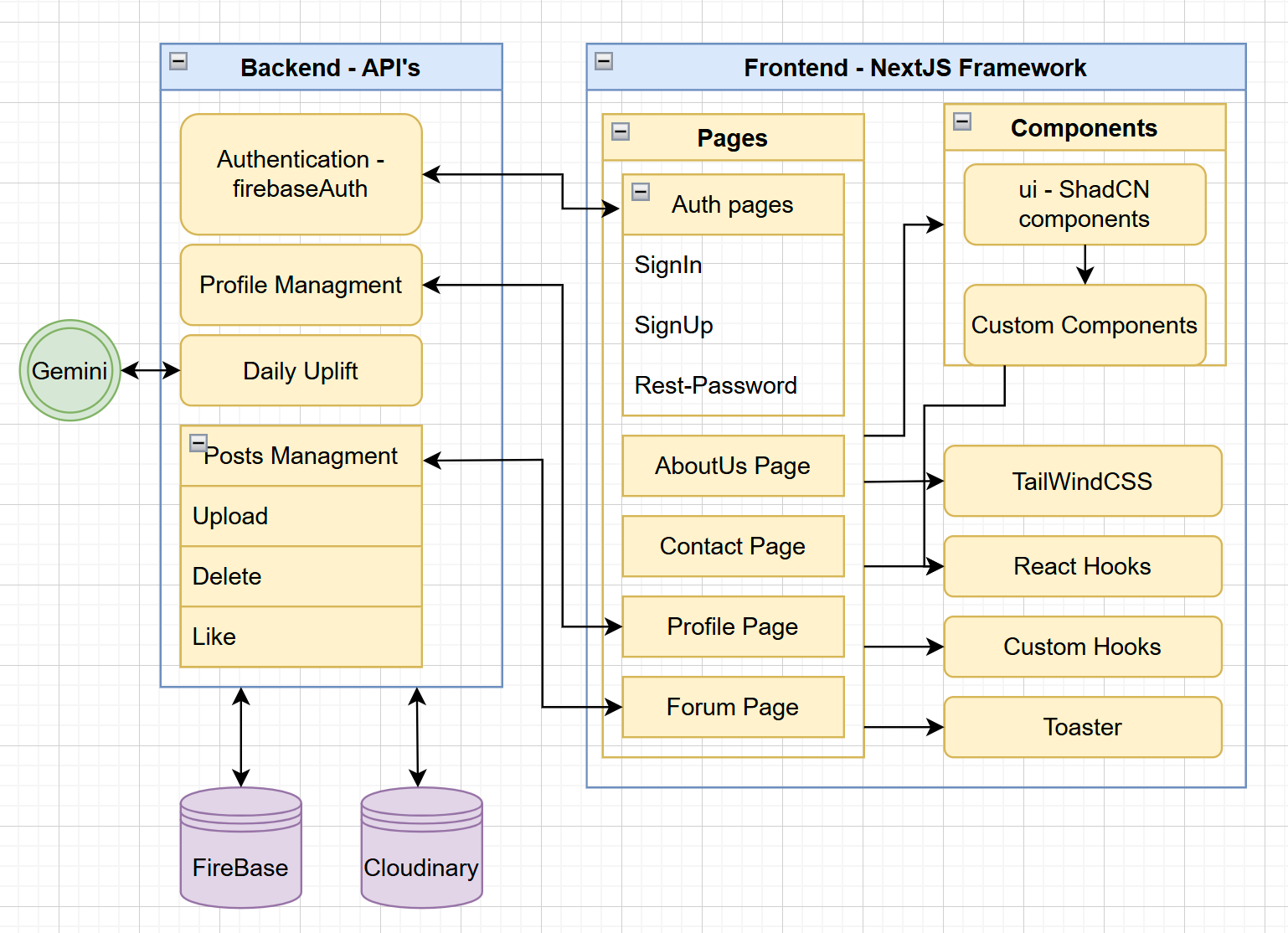
1. Hardware and Operating System Requirements:

|  |  |
| --- | --- |
| **Aspect** | **Details** |
| Supported Devices | Desktops, laptops, tablets (iOS/Android), smartphones (Android 10+, iOS 13+) |
| OS Requirements | Windows 10+, macOS Big Sur+, Android 10+, iOS 13+, Linux (with Chrome/Firefox) |
| Minimum Network Speed | 1 Mbps |
| Recommended Network Speed | 5 Mbps (for optimal real-time experience) |
| Communication Protocols | All data sent via HTTPS; uses REST APIs for user auth, data writes, content retrieval |

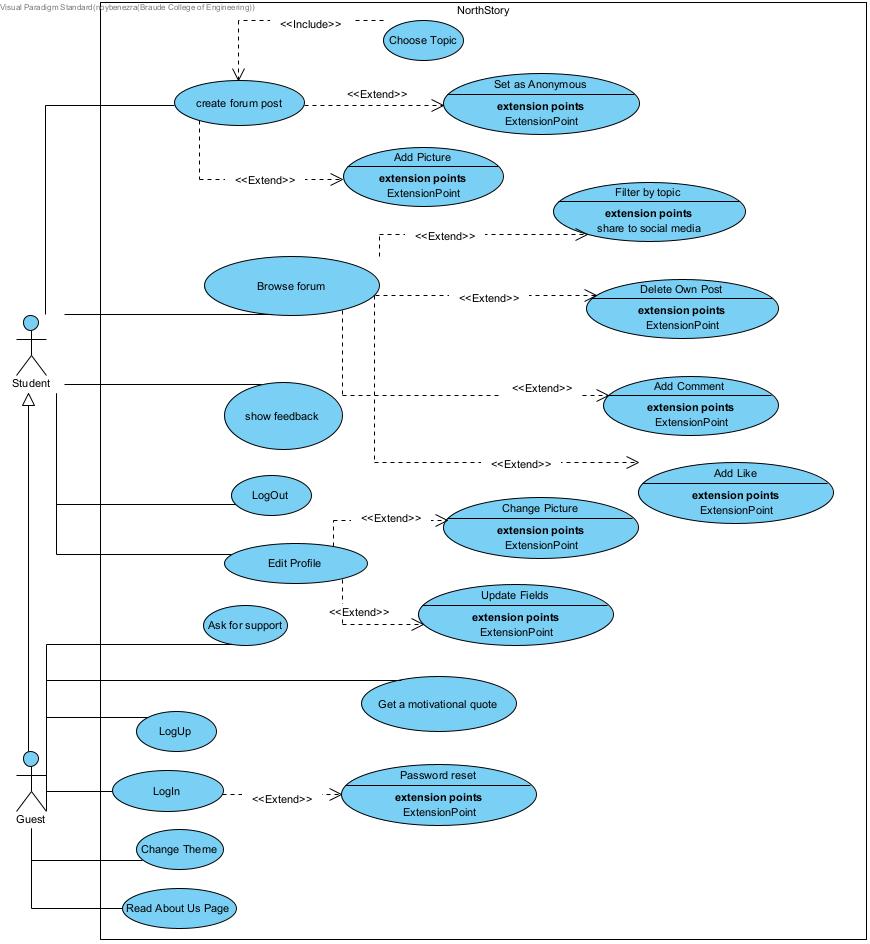
1. Software Interface Requirements:

|  |  |
| --- | --- |
| **Component** | **Technology** |
| Frontend | React.js with Next.js for SSR and routing |
| Styling | Tailwind CSS for responsive design and utility-first layout |
| Authentication | Firebase Authentication |
| Database | Cloud Firestore (NoSQL) with real-time syncing |
| Media Storage | Firebase Storage and Cloudinary for image uploads |
| Hosting | Vercel for deployment |
| Additional APIs | Gemini API for AI-generated prompts; uses Next.js/React Router for navigation |

**ארכיטקטורת האתר:**

****

**דיאגרמת use case :**



מבנה סופי של האתר שלכם:

**A diagram of a company

AI-generated content may be incorrect.מבנה הפרוייקט:**

Project configurations

files

**layout.js**

* **קומפוננטת Layout ראשית של Next.js:**
  + **עוטפת את כל הדפים**
  + **כוללת את ה־Navbar, Footer, ThemeProvider**
  + **מגדירה מבנה בסיסי (header, main, footer)**
  + **מאפשרת שמירה על Global State (למשל AuthContext)**

**page.js דף הבית של האתר:**

* + **מציג תוכן ראשי (ברוכים הבאים, קרוסלה, פיצ'רים)**
  + **כולל קריאות ל־API להצגת פוסטים וסטטיסטיקות**
  + **מכיל קישורים לדפים עיקריים**

**reset-password/page.jsx דף איפוס סיסמה:**

* + **טופס הזנת אימייל**
  + **קריאה ל-Firebase לשליחת מייל איפוס**
  + **טיפול בשגיאות והודעות הצלחה**

**signin/page.jsx דף התחברות:**

* + **טופס אימייל + סיסמה**
  + **כפתור התחברות עם Google/Facebook**
  + **קריאה ל־Firebase Auth**
  + **טיפול בשגיאות (משתמש לא קיים, סיסמה שגויה)**
  + **קישורים לאיפוס סיסמה/הרשמה**

**signup/page.jsx דף הרשמה:**

* + **טופס עם שם, אימייל, סיסמה, תמונה**
  + **קריאה ל־Firebase ליצירת משתמש**
  + **טיפול בשגיאות**
  + **הפניה אוטומטית לדף הבית/פרופיל**

**about-us/page.jsx דף "אודות":**

* + **מידע על הצוות/האתר**
  + **קרוסלת תמונות (AboutCarousel)**
  + **קישורים לרשתות חברתיות**

**contact/page.jsx דף "צור קשר":**

* + **טופס לשליחת הודעה (שם, אימייל, תוכן)**
  + **שליחה ל־API/מייל**
  + **פרטי קשר**

**profile/page.jsx דף פרופיל משתמש:**

* + **הצגת פרטי משתמש**
  + **העלאת תמונת פרופיל**
  + **רשימת פוסטים אישיים**
  + **עריכת פרטים**

**delete-image/route.js**

* **מחיקת תמונה:**
  + **מזהה תמונה או URL**
  + **קריאה ל־Cloudinary/Firebase**
  + **החזרת סטטוס**

**uplift/route.js שליחת uplift עידוד יומי:**

* + **מזהה שולח, מקבל, תוכן הודעה**
  + **שמירה ב־Firebase**
  + **החזרת סטטוס**

**upload/route.js העלאת תמונה:**

* + **קבלת קובץ**
  + **העלאה ל־Cloudinary**
  + **החזרת URL**

**קומפוננטות עיקריות:**

* **AboutCarousel.jsx קרוסלת תמונות לדף "אודות"**
* **AnonymousCommentSection.jsx תגובות אנונימיות**
* **AnonymousForum.jsx פורום אנונימי**
* **AnonymousPost.jsx פוסט אנונימי**
* **CommentSection.jsx תגובות רגילות**
* **CreatePost.jsx יצירת פוסט**
* **DesktopNavbar.jsx / MobileNavbar.jsx / Navbar.jsx ניווט רספונסיבי**
* **Footer.jsx תחתית האתר**
* **Forum.jsx תצוגת פורום**
* **ForumNavbar.jsx / MainForumNavbar.jsx ניווט ייעודי לפורום**
* **ModeToggle.jsx מתג מצב עיצוב (כהה/בהיר)**
* **Post.jsx פוסט בודד**
* **Sidebar.jsx / UnAuthenticatedSidebar.jsx סיידבר (למשתמשים מחוברים/לא)**
* **theme-provider.jsx ספק עיצוב לאפליקציה כולה**
* **UpliftCard.jsx / UpliftPopup.jsx עידוד יומי /פופאפ פידבק**
* **UploadProfileImage.jsx העלאת תמונת פרופיל**

**components/ui/**

**קומפוננטות UI קטנות (שימוש חוזר מספריית SHADCN):**

* **accordion.jsx אקורדיון**
* **avatar.jsx תמונת משתמש**
* **button.jsx כפתור**
* **card.jsx כרטיס תוכן**
* **carousel.jsx קרוסלה**
* **dialog.jsx דיאלוג/מודאל**
* **dropdown-menu.jsx תפריט נפתח**
* **input.jsx שדה קלט**
* **label.jsx תווית**
* **separator.jsx מפריד עיצובי**
* **sheet.jsx פאנל צדדי**
* **textarea.jsx טקסט ארוך**

**AuthContext.js יצירת Context לניהול התחברות:**

* + **שמירת מצב התחברות**
  + **פונקציות התחברות/הרשמה**
  + **הפצה לאפליקציה**

**useAuthUser.js Hook לשליפת מידע על המשתמש:**

* + **שימוש ב־AuthContext**
  + **החזרת אובייקט משתמש/סטטוס התחברות**

**cloudinary.js העלאה/מחיקה של תמונות ב־Cloudinary**

**firebase.js הגדרת Firebase:**

* + **התחברות/הרשמה/איפוס**
  + **עבודה עם Firestore ו־Storage**

**posts.js ניהול פוסטים:**

* + **יצירה, שליפה, עדכון, מחיקה**

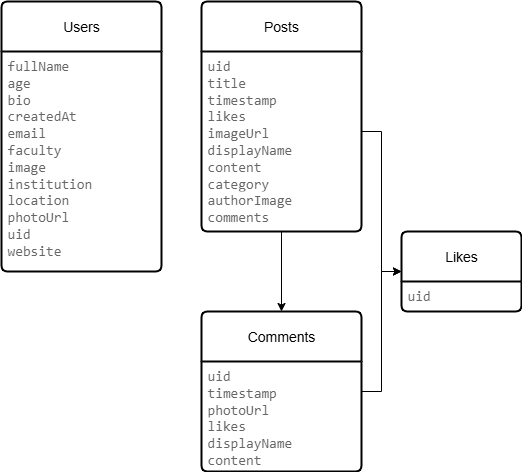
**טכנולוגיות בשימוש הפרוייקט:  
Frontend :**

* React
* NextJS
* TailwindCSS
* ShadCN/UI

**Backend / Remote Services :**

* Firebase Authentication
* Firestore DB
* Cloudinary
* Gemini
* Vercel

**מבנה מסד נתונים (**fireStore):



* **USERS**  
  טבלת המשתמשים שומרת את כל פרטי הפרופיל של חברי הרשת – סטודנטים, מרצים ואחרים. כל משתמש מזוהה לפי uid, וכולל שדות כמו שם מלא, גיל, ביוגרפיה קצרה, תאריך הרשמה, מייל, פקולטה, מוסד לימודים, מיקום, תמונת פרופיל וכתובת אתר אישי. הנתונים משמשים להצגת פרופיל אישי וחברתי באפליקציה.
* **POSTS**  
  הטבלה מכילה את כל הפוסטים שמעלים המשתמשים. כל פוסט כולל uid של המחבר, כותרת, תוכן, תאריך פרסום, קטגוריה אקדמית, תמונה מצורפת, שם ותמונה של הכותב, וכן מספר לייקים ותגובות. היא מהווה את מרכז השיח ברשת החברתית סביב נושאים לימודיים וחברתיים.
* **COMMENTS**  
  התגובות לפוסטים נשמרות בטבלה זו. כל תגובה כוללת uid של הכותב, תאריך, תוכן, שם ותמונת תצוגה, ומספר לייקים. התגובות מקושרות לפוסטים ומאפשרות אינטראקציה ודיון בין משתמשים.
* **LIKES**  
  טבלת הלייקים עוקבת אחרי משתמשים שלחצו לייק על פוסטים או תגובות. היא כוללת רק את uid של המשתמש, ומשמשת לספירת לייקים ולהצגת פידבק חברתי באפליקציה.

**NorthStory Web Application – Programmer Guide**

**Project Overview & Architecture**

**NorthStory** is a web application built with Next.js 13 (App Router) that serves as a social platform for students to share posts and engage with a community. Key features include:

* **User Authentication** – Sign-up, login, logout, and password reset via Firebase Authentication (email/password).
* **User Profiles** – Each user has a profile page with editable information (name, bio, etc.) and a profile picture.
* **Posts & Comments (Forum)** – Logged-in users can create posts (with text, emojis, and optional images), view a feed of posts, and comment on posts in real-time. Basic post interactions like “likes” are prepared in the data model.
* **Image Uploads** – Images (profile pictures and post images) are stored on Cloudinary, with the app integrating Cloudinary’s API for uploading and deleting images.
* **Daily Uplift (AI Integration)** – The app fetches a daily motivational quote from an external AI service (Google’s PaLM **Gemini** API) to display to users, demonstrating integration of third-party AI.
* **Responsive Design & Theming** – The UI is built with mobile-first responsive design (separate mobile vs desktop navigation components) and supports **dark mode** with theme toggle.

The application is structured as a **Next.js App Router** project. There is no dedicated backend server – instead, Next.js API routes (serverless functions) handle server-side tasks like image uploads, image deletion, and external API calls. Firebase Firestore serves as the primary database. The app is deployed on Vercel, and sensitive configuration (API keys, etc.) is managed via environment variables.

**Technology Stack**

**Frontend:**

* **Next.js 13 (React)** – Utilizes the App Router (src/app/) for page routing and layouts. This provides server-side rendering and file-based routing.
* **JavaScript (ES6+) & JSX** – The project is written in modern JavaScript. *(No TypeScript is used.)*
* **Tailwind CSS** – A utility-first CSS framework for styling. Tailwind is configured for dark mode (darkMode: 'class') so that adding a .dark class on the root switches themes. Responsive design is achieved with Tailwind’s utility classes (e.g. md:hidden, lg:grid-cols-12, etc.).
* **Shadcn/UI Components** – Pre-built, themeable UI components generated via Shadcn UI (integrated with Tailwind and Radix UI). The project’s src/components/ui/ directory contains reusable components like **Button**, **Card**, **Dialog**, **Input**, **Avatar**, etc., that come from Shadcn’s library for a consistent design system.
* **Next Themes** – For theme switching, the app uses the next-themes library. A custom <ThemeProvider> wraps the app to enable dark/light mode toggling by applying a dark class to <html>.

**Backend & Services:**

* **Firebase Authentication** – Handles user accounts (email/password). The Firebase Web SDK is used directly in the front-end for login, logout, etc., with secure persistence of auth state.
* **Firebase Firestore** – A NoSQL cloud database for storing app data. Collections used include users (profile info) and posts (with subcollections for comments). Firestore is accessed on the client side via the SDK. Security rules (configured in Firebase, not in this codebase) protect data access (e.g. only owners can edit their data).
* **Cloudinary** – Cloud storage for images. Instead of storing images in Firebase, images are uploaded to Cloudinary and only their URLs are saved in Firestore. This offloads image hosting to Cloudinary’s CDN. The app uses Cloudinary’s REST API for uploads and deletion. For profile images, a server-side upload route (using Cloudinary SDK and secret API key) is provided, whereas post images use an unsigned upload preset for direct client upload.
* **Google PaLM API (Gemini model)** – The “Daily Uplift” feature integrates with Google’s Generative Language API (code-named *Gemini* model) to generate motivational quotes. A Next.js API route calls this external service using a secret API key and returns the AI-generated message to the app.
* **Hosting/Deployment** – The app is designed to be deployed on **Vercel** (leveraging Vercel’s serverless functions for the API routes). Environment variables for Firebase, Cloudinary, and the AI API key are configured in Vercel’s settings (and available locally via an .env.local file).

**Additional Libraries:** The project makes use of a few other libraries and services:

* **Lucide Icons** – Icons from the Lucide library (e.g. used for menu icons, theme toggle icons).
* **React Hot Toast** – Provides non-blocking toast notifications (for success/error messages on actions like login success, errors, etc.).
* **Emoji Mart** – An emoji picker component for adding emojis to posts.
* **Embla Carousel** – Used for the About Us page carousel. The AboutCarousel component uses Embla (with an autoplay plugin) to cycle through slides of app info.
* **UI Avatars** (third-party service) – For users without a profile photo, the app uses UI Avatars to generate a placeholder image with the user’s initials. This is done by constructing a URL with the user’s name (e.g. https://ui-avatars.com/api/?name=John+Doe&background=random), which Cloudinary or the browser fetches as needed.

**Environment Configuration**

All sensitive keys and config are stored in environment variables (not in the repository). The project expects an .env.local file with entries such as:

* **Firebase Config:** NEXT\_PUBLIC\_FIREBASE\_API\_KEY, NEXT\_PUBLIC\_FIREBASE\_AUTH\_DOMAIN, NEXT\_PUBLIC\_FIREBASE\_PROJECT\_ID, NEXT\_PUBLIC\_FIREBASE\_STORAGE\_BUCKET, NEXT\_PUBLIC\_FIREBASE\_MESSAGING\_SENDER\_ID, NEXT\_PUBLIC\_FIREBASE\_APP\_ID. These correspond to the Firebase project credentials. (Keys prefixed with NEXT\_PUBLIC\_ are exposed to the client bundle, which is acceptable since Firebase rules secure the data.)
* **Cloudinary Config:** CLOUDINARY\_CLOUD\_NAME, CLOUDINARY\_API\_KEY, CLOUDINARY\_API\_SECRET for server-side use, and a NEXT\_PUBLIC\_CLOUDINARY\_UPLOAD\_PRESET for unsigned uploads. The upload preset is configured in your Cloudinary account to allow client-side uploads to a specific folder with certain restrictions.
* **AI API Key:** GEMINI\_API\_KEY (or similar) for the generative language API. **Note:** In the current code, the Google API key is unfortunately hard-coded in the uplift route (which should be changed for security). In a production setup, this key should be stored in an env variable and referenced as process.env.GEMINI\_API\_KEY in the code.

During development, create an .env.local with the above keys. On Vercel (production), set these environment variables in the project settings. The app should not run without these configurations, as they are required for connecting to Firebase, Cloudinary, and the external API.

**Project Structure**

The source code is organized under the src/ directory as follows:

* **src/app/** – Next.js **App Router** pages and API routes:
  + **Pages:** The application pages are defined here, leveraging Next.js’ file-based routing. For example:
    - src/app/(auth)/signin/page.jsx – Sign In page (within the (auth) route group for unauthenticated pages).
    - src/app/(auth)/signup/page.jsx – Sign Up page.
    - src/app/(auth)/reset-password/page.jsx – Password reset page.
    - src/app/profile/page.jsx – Profile page (user must be logged in).
    - src/app/posts/page.jsx – (Optional posts listing page; the main feed is also shown on home page for logged-in users).
    - src/app/about-us/page.jsx – “About Us” static page.
    - src/app/contact/page.jsx – “Contact” static page.
    - src/app/page.js – The homepage (shows landing content or feed depending on auth state).
    - src/app/layout.js – Global layout applied to all pages. This defines the <head> metadata, includes the theme and auth providers, navbars/sidebars, etc.
  + **API Routes:** Under src/app/api/ are Next.js API endpoints (serverless functions):
    - src/app/api/upload/route.js – **Image upload** endpoint (for profile images). Runs server-side Cloudinary upload.
    - src/app/api/delete-image/route.js – **Image deletion** endpoint. Deletes an image from Cloudinary by public ID.
    - src/app/api/uplift/route.js – **AI quote** endpoint. Fetches a motivational quote from the external Gemini API.
* **src/components/** – UI Components (client-side React components):
  + **Reusable UI:** src/components/ui/ contains generic UI building blocks from Shadcn (buttons, inputs, cards, modal dialogs, etc.).
  + **Navigation:** Navbar.jsx, DesktopNavbar.jsx, MobileNavbar.jsx, Sidebar.jsx, UnAuthenticatedSidebar.jsx, Footer.jsx, etc. These manage layout and navigation for different screen sizes and auth states.
  + **Feature Components:** e.g. CreatePost.jsx (post creation form), Post.jsx (displays a single post item), CommentSection.jsx (comments thread for a post), UploadProfileImage.jsx (modal for uploading a new profile picture), UpliftCard.jsx and UpliftPopup.jsx (for the daily quote feature), AboutCarousel.jsx (carousel on the About Us page), and others.
  + **Theme:** ModeToggle.jsx (a button to toggle dark/light mode using an icon).
* **src/context/** – React Contexts for global state:
  + AuthContext.js – Provides global authentication state (the current user’s profile data and a loading state) and a setter to update it. Wrapped around the app to make user available in any component.
  + *(Note: Originally one might expect auth actions here, but in this project sign-in and sign-up logic is handled in the pages directly; the context is mainly for state.)*
* **src/hooks/** – Custom React hooks:
  + useAuthUser.js – A hook alternative to AuthContext that listens to Firebase auth changes and ensures a Firestore user profile exists. Returns { user, loading } similar to AuthContext. (The app uses either this or the context in various places to get current user data.)
* **src/lib/** – Utility modules:
  + firebase.js – Initializes the Firebase app and exports configured instances of **Auth** and **Firestore** (auth and db).
  + cloudinary.js – (Optional utility, in this code it exports a client-side uploadToCloudinary function for unsigned uploads, not heavily used since uploads are also implemented elsewhere).
  + posts.js – Contains helper functions for Firestore operations related to posts (e.g. createPost, getAllPosts). In practice, the components directly use Firestore calls, but this module documents the intended logic.

This structure separates concerns: pages define layout and fetch data, components encapsulate UI logic, context/hook provides global state, and lib modules handle external integrations.

**Authentication (Sign In, Sign Up, Logout, Reset Password)**

**Overview:** Authentication is handled by Firebase Authentication (email/password). Users can create an account, log in, log out, and reset their password. Auth-protected pages are grouped under (auth) for routing clarity and to possibly apply separate layout or middleware in the future. The app uses a global AuthContext (and the useAuthUser hook) to track the authenticated user’s state and profile.

**Relevant Files:**

* src/app/(auth)/signup/page.jsx – Registration page with a form (email, password, and name).
* src/app/(auth)/signin/page.jsx – Login page with email/password form.
* src/app/(auth)/reset-password/page.jsx – Password reset page (enter email to get reset link).
* src/context/AuthContext.js – Auth context provider that listens for auth status changes and fetches user profile data.
* src/hooks/useAuthUser.js – Custom hook doing similar logic (auth state + Firestore profile), used in some components.
* Components like Navbar and Sidebar also utilize auth state to show appropriate links.

**Sign Up (Registration):** On the SignUp page, when the form is submitted, the code calls Firebase to create a new user and then creates a Firestore document for that user’s profile. For example:

jsx

CopyEdit

// Inside src/app/(auth)/signup/page.jsx – handle form submission:

try {

const userCredential = await createUserWithEmailAndPassword(auth, email, password);

const user = userCredential.user;

// Build the profile object to store in Firestore:

const userProfile = {

uid: user.uid,

email: email,

fullName: fullName,

createdAt: new Date().toISOString(),

photoURL: "/defaultUserLogo.jpg", // default avatar image

location: "Not set",

institution: "Not set",

faculty: "Not set",

age: null,

website: "",

bio: "This is your bio. Tell us more about you.",

followers: 0,

following: 0

};

await setDoc(doc(db, "users", user.uid), userProfile);

toast.success("Account created!");

router.push("/"); // redirect to home (which will show main app if logged in)

} catch (error) {

// handle errors (e.g., email already in use)

toast.error(error.message);

}

After Firebase returns the new user, a corresponding document in the Firestore users collection is created with default profile fields. This ensures we have a place to store additional info (full name, bio, etc.) since the Firebase Auth user object alone only has email and UID by default. The app uses this Firestore users/{uid} document for profile info display and updates.

**Sign In (Login):** The SignIn page calls Firebase Auth’s signInWithEmailAndPassword:

jsx

CopyEdit

// Inside src/app/(auth)/signin/page.jsx – handle login:

try {

await signInWithEmailAndPassword(auth, email, password);

toast.success("Login successful!");

router.push("/"); // redirect to main app after a short delay

} catch (error) {

toast.error("Invalid credentials");

}

On successful login, Firebase sets the current user (auth.currentUser). The AuthContext (described below) will detect this change and load the user’s profile. The app then redirects to the home page, which for a logged-in user will show the posts feed.

**AuthContext / useAuthUser:** The app wraps its UI with <AuthProvider> (see layout.js), which uses Firebase’s onAuthStateChanged listener to respond to login/logout events. When a user signs in or when the app loads with a user already logged in, the context fetches that user’s profile from Firestore and makes it available to components:

jsx

CopyEdit

// Simplified AuthContext logic (src/context/AuthContext.js):

useEffect(() => {

const unsubscribe = onAuthStateChanged(auth, async (firebaseUser) => {

if (firebaseUser) {

const userDoc = await getDoc(doc(db, "users", firebaseUser.uid));

if (userDoc.exists()) {

setUser(userDoc.data()); // put Firestore profile data in context state

}

} else {

setUser(null);

}

setLoading(false);

});

return () => unsubscribe();

}, []);

The context provides { user, setUser, loading } via React context. Components can access it through a custom useAuth() (or useContext(AuthContext)). The user object contains profile fields like fullName, email, photoURL (or image field – see Profile section), etc., as pulled from Firestore. The setUser function is exposed so that profile updates can immediately reflect in global state.

Alternatively, some parts of the app use the useAuthUser hook which implements a similar mechanism internally (ensuring a Firestore doc exists, creating one if not). In practice, either approach yields the current user’s data and an auth loading state.

**Logout:** For signing out, the UI simply calls Firebase’s signOut(auth). There isn’t a dedicated page for logout; instead, the Logout action is triggered by a button (e.g., in the Navbar or Sidebar for authenticated users). For example, in the desktop navbar:

jsx

CopyEdit

import { signOut } from "firebase/auth";

import { auth } from "@/lib/firebase";

// ...

{user ? (

<Button onClick={() => signOut(auth)}>Sign Out</Button>

) : (

<Link href="/signin">Log In</Link>

)}

When signOut(auth) is called, Firebase clears the current user, and the AuthContext onAuthStateChanged will set the user state to null. This triggers the UI to update (showing the logged-out view).

**Reset Password:** The Reset Password page provides a form to input an email. It uses Firebase’s sendPasswordResetEmail(auth, email) function. On success, Firebase will email the user a password reset link. The UI will typically show a confirmation message like “Check your email for reset instructions.” If the email is not found or invalid, an error is shown.

**Routing & Protection:** All auth pages (signin, signup, reset-password) are in the (auth) folder. They are accessible to anyone (even if already logged in, though we could redirect if logged in – not implemented, but could be a future improvement). Protected pages (like /profile and the main feed / or /posts) ensure the user is logged in before showing content. For example, profile/page.jsx does:

jsx

CopyEdit

if (!auth.currentUser) {

router.push("/signin");

return null;

}

This client-side check immediately redirects a non-authenticated user to the Sign In page if they somehow navigate to /profile without a valid session. Similarly, the main forum page only shows content if user exists; otherwise the landing page is shown. (Alternatively, one could use Next.js Middleware or server-side protection, but here it’s done in the client for simplicity.)

**Profile Management (User Profile & Avatar)**

**Overview:** Each user has a profile page where they can view and update their personal information. This includes changing display name (full name), adding details like age, location, etc., updating their bio, and changing their profile picture. Profile data is primarily stored in the Firestore users collection rather than relying solely on Firebase Auth’s basic user info. The profile page pulls the latest data from Firestore and allows edits, which are saved back to Firestore and reflected in the UI (via context state updates).

**Relevant Files:**

* src/app/profile/page.jsx – The profile page component that displays a form with the user’s current info and allows editing.
* src/components/UploadProfileImage.jsx – A component providing a modal dialog to upload a new profile photo.
* src/components/Sidebar.jsx – Displays the user’s profile summary (name, photo, etc.) in the sidebar when logged in.
* src/lib/firebase.js – Provides the auth and db instances for Firebase.
* src/context/AuthContext.js or useAuthUser.js – Supplies the current user’s data to the profile page.

**Profile Data:** The Firestore **users** document for each user contains fields such as:

json

CopyEdit

{

"uid": "...",

"email": "user@example.com",

"fullName": "John Doe",

"photoURL": "https://.../profile\_pictures/abc123.jpg", // link to profile image

"age": 25,

"location": "City, Country",

"institution": "University Name",

"faculty": "Department or Faculty",

"website": "https://...",

"bio": "Short bio about the user...",

"followers": 0,

"following": 0

}

(New users start with default or empty values for most of these, as seen in the Sign Up logic above.)

The **Profile Page** (profile/page.jsx) fetches the current user’s profile from context or directly from Firestore. It populates an edit form with these values. Users can change their full name, age, location, etc., and submit the form. The code then validates and updates the Firestore doc:

jsx

CopyEdit

// Inside Profile page component:

const [userData, setUserData] = useState(null);

// Load profile on mount:

useEffect(() => {

const loadProfile = async () => {

const snap = await getDoc(doc(db, "users", auth.currentUser.uid));

if (snap.exists()) {

setUserData(snap.data());

}

};

loadProfile();

}, []);

const handleChange = (field, value) => {

setUserData(prev => ({ ...prev, [field]: value }));

};

const handleSubmit = async (e) => {

e.preventDefault();

if (userData.age && (userData.age < 10 || userData.age > 120)) {

return toast.error("Please enter a valid age between 10 and 120.");

}

try {

await updateDoc(doc(db, "users", userData.uid), userData);

setUser(userData); // update global context state with new data

toast.success("Profile updated successfully!");

} catch (err) {

console.error("Failed to update profile:", err);

toast.error("Update failed. Try again.");

}

};

After validation (e.g., ensuring age is within a reasonable range), updateDoc writes the entire userData object back to Firestore. The context’s setUser is called to immediately update the in-memory user data, so the changes reflect in the UI (e.g. the sidebar will now show the new name, etc.). A success toast notifies the user.

**Profile Picture Upload:** Changing the profile picture is handled by the UploadProfileImage component, which is typically rendered as a modal dialog from the profile page. For example, on the profile page there might be an “Edit Photo” button that opens <UploadProfileImage open={true} ... />. This component handles the file selection and uploading process:

1. **Selecting a file:** The component shows an <input type="file"> (or drag-and-drop area) for the user to choose an image. Once a file is selected, handleImageUpload(file) is called.
2. **Uploading to Cloudinary:** handleImageUpload will upload the file to Cloudinary. In this project, the client-side upload approach is used (with an unsigned preset). For instance:

jsx

CopyEdit

// Inside UploadProfileImage.jsx

const handleImageUpload = async (file) => {

if (!file) return;

// Basic validations (file type and size)

if (!file.type.startsWith("image/")) {

return toast.error("Only image files are allowed.");

}

if (file.size > 5 \* 1024 \* 1024) {

return toast.error("Max file size is 5MB.");

}

setUploading(true);

try {

// Prepare form data for Cloudinary unsigned upload

const formData = new FormData();

formData.append("file", file);

formData.append("upload\_preset", process.env.NEXT\_PUBLIC\_CLOUDINARY\_UPLOAD\_PRESET);

const res = await axios.post(

`https://api.cloudinary.com/v1\_1/${process.env.NEXT\_PUBLIC\_CLOUDINARY\_CLOUD\_NAME}/image/upload`,

formData

);

const imageUrl = res.data.secure\_url;

// Save the new photo URL in Firestore:

const currentUser = auth.currentUser;

await updateDoc(doc(db, "users", currentUser.uid), { photoURL: imageUrl });

// Update global state and local state

setUserData(prev => ({ ...prev, photoURL: imageUrl }));

setUser(prev => ({ ...prev, photoURL: imageUrl })); // update AuthContext user

toast.success("Profile picture updated!");

onClose(); // close the modal

} catch (err) {

console.error("Image upload failed:", err);

toast.error("Failed to upload image. Please try again.");

} finally {

setUploading(false);

}

};

In the above logic, the image file is posted directly to Cloudinary’s API endpoint for uploads. An **upload preset** is used to allow client-side uploads without exposing the API secret. Cloudinary responds with a secure\_url of the uploaded image. That URL is then stored in the user’s Firestore document (photoURL field) to update their profile.

1. **Updating UI:** After uploading, the new photoURL is saved in the AuthContext (setUser) so that any component showing the user’s avatar (e.g. the Sidebar or Navbar) will immediately reflect the change. The modal then closes.

Because the profile page passes an onUpload callback (onUpload={(newImageUrl) => {...}}), the parent can also update its state if needed. In our case, we directly update the context and profile state inside the component, so the parent’s onUpload might simply merge the new URL into userData (as shown with setUserData above).

**Displaying Avatars:** The app uses the user’s photoURL for their avatar in the UI. The Sidebar.jsx for example, uses an Avatar component:

jsx

CopyEdit

import { Avatar, AvatarImage, AvatarFallback } from "@/components/ui/avatar";

if (!user) return <UnAuthenticatedSidebar />; // if no user, show login prompts

return (

<Card>

<CardContent>

<Avatar className="h-16 w-16">

<AvatarImage src={user.photoURL} alt={user.fullName} />

<AvatarFallback>

{user.fullName ? user.fullName[0] : 'U'}

</AvatarFallback>

</Avatar>

<p>{user.fullName}</p>

<p className="text-muted-foreground text-sm">{user.email}</p>

{/\* other profile info \*/}

</CardContent>

</Card>

);

If user.photoURL is not set, it may show a placeholder. In some parts of the code, they generate a placeholder URL via UI Avatars service (using the name). This way, even without an uploaded image, every user has some avatar in the UI (either their chosen photo or an initial-based graphic).

**Note:** The Firebase Auth currentUser.photoURL is not explicitly used/updated here. Instead, the app relies on the Firestore profile document’s photoURL. This means that auth.currentUser.photoURL might be stale (or blank) while the user’s actual photo is stored in Firestore. This design is fine as long as all components use the user object from context (which contains Firestore data). If desired, one could also call updateProfile(auth.currentUser, { photoURL: imageUrl }) to keep Firebase Auth in sync, but it’s not strictly necessary when using a separate profile store.

**Posts & Forum (Creating, Displaying, and Managing Posts)**

**Overview:** A core feature of NorthStory is a forum-like feed where users can create posts and view others’ posts. A post typically contains a title, some content (text, which can include emojis), and optionally an image. Each post is associated with its author (user) and a category/tag. Posts are stored in Firestore in a posts collection. The app displays posts in reverse chronological order and allows basic interactions: users can comment on posts (see next section) and “like” posts (a like system is prepared, storing user IDs who liked, though the UI for liking might be minimal). Users can also delete their own posts, which removes the post and its comments, and cleans up any image from Cloudinary.

**Relevant Files:**

* src/app/page.js – The home page: for logged-in users, this renders the <Forum /> component (posts feed); for logged-out users, it shows a landing page.
* src/app/posts/page.jsx – An alternative page for the posts feed (similar purpose to the home page when logged in).
* src/components/Forum.jsx – The main forum component that manages fetching posts (in real-time) and rendering posts and the post creation form.
* src/components/CreatePost.jsx – The form UI for creating a new post (textarea for content, file input for image, emoji picker, etc.).
* src/components/Post.jsx – Displays an individual post, including its content, author, timestamp, and like/comment buttons.
* src/lib/posts.js – Helper functions (not heavily used in final code, but defines logic for add/get posts).

**Data Model (Firestore):** Each post is a document in the **posts** collection. Key fields include:

* uid (author’s user ID),
* displayName (author’s name at time of post),
* authorImage (author’s avatar URL at time of post),
* title (post title or topic, if used),
* content (the body text of the post),
* category (a tag or category name for the post, e.g. "General" or "Questions"),
* imageUrl (URL of an attached image, if any),
* timestamp (Firestore server timestamp when created),
* likes (array of user IDs who have liked the post).

For example, a post document might look like:

json

CopyEdit

{

"uid": "USER123",

"displayName": "John Doe",

"authorImage": "https://ui-avatars.com/api/?name=John+Doe...",

"title": "Need help with calculus",

"content": "I am struggling with calculus integration problems 😕. Any tips?",

"category": "Academics",

"imageUrl": "https://res.cloudinary.com/<cloud\_name>/.../abcxyz.jpg",

"timestamp": { "\_\_datatype\_\_": "timestamp", ... },

"likes": ["ANOTHERUSER456"]

}

*(If a field like title or imageUrl isn’t provided by the user, it may be omitted or left blank. The code sometimes uses default values, e.g., if no authorImage is available from the user’s profile, it uses the UI Avatars URL as shown above.)*

**Creating a Post:** Logged-in users can create a new post using the form in the UI (the <CreatePost> component). This form typically includes a textarea for the content, an optional field for title or category, an emoji-picker button (to insert emojis into the content), and an image upload input.

The process:

1. The user enters text (and possibly selects a category from a dropdown, if implemented).
2. The user can attach an image – on selecting a file, similar to profile image upload, the image is uploaded to Cloudinary using the same unsigned preset approach. In CreatePost.jsx, there is logic like:

jsx

CopyEdit

const [uploadedImageUrl, setUploadedImageUrl] = useState(null);

const handleImageUpload = async (file) => {

// ... validate file type/size ...

const formData = new FormData();

formData.append("file", file);

formData.append("upload\_preset", process.env.NEXT\_PUBLIC\_CLOUDINARY\_UPLOAD\_PRESET);

const res = await axios.post(`https://api.cloudinary.com/v1\_1/${process.env.NEXT\_PUBLIC\_CLOUDINARY\_CLOUD\_NAME}/image/upload`, formData);

setUploadedImageUrl(res.data.secure\_url);

};

The image is uploaded immediately when selected, and the returned URL is stored in component state (uploadedImageUrl). This way, the user could even preview the image or decide to remove it before posting (if the UI supports that).

1. When the user submits the post (e.g., clicks a “Post” button), the component gathers the data:

jsx

CopyEdit

const handleSubmitPost = async () => {

if (!content) return;

const newPost = {

title,

content,

category: selectedCategory || "General",

imageUrl: uploadedImageUrl || null

};

onSubmit(newPost);

};

Here, onSubmit is a prop passed from the parent (Forum component). The Forum will handle actually writing to Firestore.

1. The <Forum> component’s handleCreatePost receives the new post data and performs the Firestore operation:

jsx

CopyEdit

// Inside Forum.jsx

const handleCreatePost = async ({ title, content, category, imageUrl }) => {

// Add to Firestore

await addDoc(collection(db, "posts"), {

uid: user.uid,

displayName: user.fullName || user.email,

authorImage: user.photoURL || `https://ui-avatars.com/api/?name=${encodeURIComponent(user.fullName || "User")}`,

title,

content,

category,

imageUrl: imageUrl || "",

timestamp: serverTimestamp(),

likes: []

});

// (No need to manually add comments field; comments will be a subcollection.)

};

Notice it uses the current user (from context or hook) to fill in author info. The Firestore serverTimestamp() is used to timestamp the post. Initially, likes is an empty array.

After adding, the onSnapshot listener (described below) will automatically include this new post in the UI, so we often don’t need to manually append it to local state. However, one could also optimistically update local state if needed.

1. The form may be cleared after submission and any uploadedImageUrl state reset for the next post.

**Real-time Feed (Listening for Posts):** The forum uses Firestore’s real-time capabilities to display posts immediately as they are created or updated. In Forum.jsx, a Firestore listener might be set up on the posts collection:

jsx

CopyEdit

useEffect(() => {

const q = query(collection(db, "posts"), orderBy("timestamp", "desc"));

const unsubscribe = onSnapshot(q, (snapshot) => {

const postsData = snapshot.docs.map(doc => ({

id: doc.id,

...doc.data()

}));

setPosts(postsData);

});

return () => unsubscribe();

}, []);

This subscription ensures that when a new post is added (by any user) or an existing post is edited/deleted, the posts state updates automatically. The UI will rerender the list of <Post> components accordingly. This provides a live feed experience.

*(The project also has a getAllPosts() function in lib/posts.js using getDocs for one-time fetch, and at one point a page (posts/page.jsx) did a useEffect with getDocs. However, the real-time onSnapshot approach in Forum is the more current implementation for the main feed.)*

**Displaying Posts:** Each post in the feed is rendered via the Post.jsx component. This component receives the post data (either as props or via context if nested). It displays the content, and possibly an image if imageUrl is present:

jsx

CopyEdit

// Simplified Post component structure:

<Card className="post">

<CardHeader>

<div className="flex items-center">

<Avatar src={post.authorImage} alt={post.displayName} />

<div className="ml-2">

<p className="font-medium">{post.displayName}</p>

<p className="text-xs text-muted-foreground">{/\* timestamp, e.g. "2 hours ago" \*/}</p>

</div>

{/\* If current user is author, a Delete button ... \*/}

</div>

</CardHeader>

<CardContent>

{post.title && <h2 className="font-bold mb-2">{post.title}</h2>}

<p>{post.content}</p>

{post.imageUrl && (

<div className="mt-2">

<img src={post.imageUrl} alt="Post image" className="rounded" />

</div>

)}

<div className="mt-4 flex items-center gap-4 text-sm">

<button onClick={handleLikeToggle}>

{hasLiked ? "❤️ Unlike" : "🤍 Like"} {post.likes.length}

</button>

<button onClick={() => setShowComments(true)}>

💬 Comments ({commentCount})

</button>

</div>

</CardContent>

</Card>

The above pseudo-JSX illustrates some possible elements:

* Author’s name and avatar.
* Post timestamp (could be formatted from post.timestamp).
* Post title and content.
* Attached image if present.
* Like and comment buttons with counts.

The **like button** toggles whether the current user’s ID is in the post’s likes array. The code can determine hasLiked as post.likes?.includes(currentUser.uid). On click, it updates Firestore:

jsx

CopyEdit

const handleLikeToggle = async () => {

const postRef = doc(db, "posts", post.id);

if (hasLiked) {

await updateDoc(postRef, { likes: arrayRemove(currentUser.uid) });

} else {

await updateDoc(postRef, { likes: arrayUnion(currentUser.uid) });

}

};

This uses Firestore’s atomic array operations to add or remove the user’s ID. The updated likes array will be reflected in the onSnapshot listener, updating the UI. (Ensure that security rules allow this update – typically one would allow users to update the likes array on posts.)

The **comment Section** component (covered next) would show the list of comments for the post and a field to add a new comment.

If the current user is the author of the post, the UI could show a **Delete Post** option (e.g., a trash icon). The code for deleting a post is implemented in Forum.jsx as handleDeletePost(postId):

jsx

CopyEdit

const handleDeletePost = async (postId) => {

if (!window.confirm("Are you sure you want to delete this post?")) return;

const postRef = doc(db, "posts", postId);

const postSnap = await getDoc(postRef);

const postData = postSnap.data();

if (postData.imageUrl) {

// If the post has an image, delete it from Cloudinary

const publicId = postData.imageUrl.split("/").pop().split(".")[0];

await fetch("/api/delete-image", {

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify({ publicId })

});

}

// Delete all comments subcollection docs

const commentsSnap = await getDocs(collection(db, "posts", postId, "comments"));

const deletePromises = commentsSnap.docs.map(doc => deleteDoc(doc.ref));

await Promise.all(deletePromises);

// Delete the post document

await deleteDoc(postRef);

toast.success("Post deleted");

};

This ensures a clean deletion: it removes the post’s image from Cloudinary (by extracting the Cloudinary public ID from the URL and calling our /api/delete-image route), then deletes all comments under that post, then deletes the post itself. The onSnapshot listener will automatically remove the post from the UI once the post doc is gone.

**Categories:** The code references a selectedCategory in CreatePost/Forum. It implies posts can be tagged with categories and filtered. For instance, the forum might have a category filter (a dropdown or tabs like “All”, “Academics”, “Social”, etc.). If implemented, selectedCategory would be state in Forum that, when changed, filters the query (or filters the in-memory list). In the CreatePost, they pass selectedCategory to tag the new post. In our snippet above, if category is “All”, they substitute “Main” as a default category. This detail suggests a possible category list where “All” is a UI option that corresponds to no filter, and "Main" or others are actual categories stored.

If categories are indeed a feature, one would see UI elements (MainForumNavbar.jsx could be a component with category filters). We won’t dive deeper since it’s straightforward: category is just a field on posts.

**Comments Functionality**

**Overview:** Users can comment on posts, enabling discussion. Comments are stored in Firestore as a subcollection under each post (this is a common pattern to keep them organized by post). Only authenticated users can post comments. The UI displays existing comments and provides a form to add a new comment. Comments also support a basic like/unlike mechanism (if desired), but primarily it’s about creating and listing comments.

**Relevant Files:**

* src/components/CommentSection.jsx – Component that renders the comments for a given post and a form to add a comment.
* src/components/AnonymousCommentSection.jsx – Possibly a variant used for when an unauthenticated user views comments (read-only).
* src/components/Post.jsx – Likely renders CommentSection when needed.
* src/components/Forum.jsx – Might handle toggling comment sections or passing data.
* Firestore structure: Comments are not in a top-level collection but as posts/{postId}/comments.

**Data Model:** A comment document might have fields:

* uid (user ID of commenter),
* displayName (commenter’s name),
* photoURL (commenter’s avatar URL or placeholder),
* text (the comment text content),
* timestamp (server timestamp when created).

Comments are stored under the parent post document for easy retrieval. Example path: posts/POST\_ID/comments/COMMENT\_ID.

**Displaying Comments:** In the Post UI, when comments are to be shown, the CommentSection component is rendered. This component likely does the following:

* Subscribe to the comments subcollection in Firestore using onSnapshot for real-time updates.
* Render a list of comments (each showing user avatar, name, text, time).
* Provide an input field for adding a new comment.

For example, CommentSection.jsx might contain:

jsx

CopyEdit

const CommentSection = ({ postId }) => {

const [comments, setComments] = useState([]);

const [newComment, setNewComment] = useState("");

useEffect(() => {

const commentsRef = collection(db, "posts", postId, "comments");

const q = query(commentsRef, orderBy("timestamp", "asc"));

const unsubscribe = onSnapshot(q, snapshot => {

const commentsData = snapshot.docs.map(doc => ({ id: doc.id, ...doc.data() }));

setComments(commentsData);

});

return () => unsubscribe();

}, [postId]);

const handleAddComment = async () => {

if (!newComment.trim()) return;

await addDoc(collection(db, "posts", postId, "comments"), {

uid: currentUser.uid,

displayName: currentUser.fullName || currentUser.email,

photoURL: currentUser.photoURL || `https://ui-avatars.com/api/?name=${currentUser.fullName}`,

text: newComment,

timestamp: serverTimestamp()

});

setNewComment("");

};

return (

<div className="comments">

{comments.map(comment => (

<div key={comment.id} className="flex gap-2 mb-2">

<img src={comment.photoURL} alt={comment.displayName} className="w-6 h-6 rounded-full" />

<div>

<p className="text-sm font-medium">{comment.displayName}</p>

<p className="text-sm">{comment.text}</p>

</div>

</div>

))}

<div className="mt-2 flex items-center gap-2">

<Input

value={newComment}

onChange={e => setNewComment(e.target.value)}

placeholder="Add a comment..."

/>

<Button onClick={handleAddComment}>Post</Button>

</div>

</div>

);

};

This pseudo-code illustrates the typical flow. The actual code likely handles edge cases and uses the UI components from Shadcn for styling (e.g., Input, Button from the ui library).

**Likes in Comments:** If needed, each comment could also have a likes array and similar like button logic, but there’s no evidence in the code snippet that it was implemented. It appears the like functionality was intended primarily for posts.

**Deleting Comments:** The app likely allows users to delete their own comments as well, though it may not be explicitly shown in the UI. Given that comments are small, a user could delete a comment by finding the comment’s document and calling deleteDoc. There might be a context menu per comment with a delete option, but if not, comments can remain without deletion. (During post deletion, all comments are removed as shown above.)

**Anonymous Posting & Commenting**

**Overview**

NorthStory includes a special category called **"Anonymous"** that allows users to post and comment without revealing their identity. Posts and comments created in this category **do not display the author's name, profile image, or user ID**. This provides a safe space where users can share personal thoughts, ask sensitive questions, or express emotions freely and without fear of judgment.

**How It Works**

**When posting:**

* Users choose the **"Anonymous"** category in the post creation form.
* The following values are saved to Firestore:
  + displayName: "Anonymous"
  + authorImage: "" (empty string) or a default avatar
  + uid: the actual user ID is stored internally but **never displayed**

**When commenting on an anonymous post:**

* The comment stores:
  + displayName: "Anonymous User"
  + photoURL: set to /defaultUserLogo.jpg
  + uid: retained internally

**Firestore Example – Anonymous Post**

json

CopyEdit

{

"uid": "user-abc123", // stored internally

"displayName": "Anonymous",

"authorImage": "/defaultUserLogo.jpg",

"title": "I'm overwhelmed",

"content": "I can't handle the pressure of my finals 😞",

"category": "Anonymous",

"imageUrl": "",

"timestamp": "...",

"likes": []

}

**UI Behavior**

* **Display Name:** Always shows "Anonymous" or "Anonymous User"
* **Avatar:** Always shows /defaultUserLogo.jpg
* **No profile links or hover previews** are displayed
* **Timestamps and likes** still work normally
* Users can still delete their own anonymous posts and comments

**Components Involved**

|  |  |
| --- | --- |
| **Component** | **Description** |
| AnonymousForum.jsx | Renders only posts in the "Anonymous" category using <AnonymousPost /> |
| AnonymousPost.jsx | Displays a single anonymous post and invokes <AnonymousCommentSection /> |
| AnonymousCommentSection.jsx | Manages anonymous comments for each post |
| CreatePost.jsx | Sets metadata like "Anonymous" based on selected category |

**Security Note**

The real uid of the author is **stored in the database** for moderation purposes but is **never shown to other users** in the interface. This ensures both accountability and privacy.

**Image Uploads & Cloudinary Integration**

**Overview:** NorthStory offloads all image storage to **Cloudinary**. This includes user profile pictures and images attached to posts. Integrating Cloudinary provides benefits like optimized delivery (CDN), image transformations if needed, and reduces load on Firebase (no need to use Firebase Storage). The app uses two approaches for Cloudinary: **client-side unsigned uploads** for quick direct uploads, and a **server-side API route** for secure uploads (used for profile pictures to store in a specific folder). Deleting images is handled through a secured API route since it requires the Cloudinary secret.

**Relevant Files:**

* src/lib/cloudinary.js – Cloudinary configuration (for server-side usage) and possibly helper functions.
* src/app/api/upload/route.js – Next.js API route to handle an image upload using Cloudinary’s Node SDK (authenticated upload).
* src/app/api/delete-image/route.js – API route to delete an image on Cloudinary using the SDK.
* Frontend components that upload images:
  + UploadProfileImage.jsx – uses either the /api/upload route or direct upload to Cloudinary.
  + CreatePost.jsx – uses direct upload to Cloudinary via unsigned preset.
* **Environment:** CLOUDINARY\_CLOUD\_NAME, CLOUDINARY\_API\_KEY, CLOUDINARY\_API\_SECRET, and NEXT\_PUBLIC\_CLOUDINARY\_UPLOAD\_PRESET must be set.

**Cloudinary Setup:** In the backend (API routes), the Cloudinary SDK is configured with credentials. For example, in the upload route:

js

CopyEdit

// src/app/api/upload/route.js

import { v2 as cloudinary } from "cloudinary";

cloudinary.config({

cloud\_name: process.env.CLOUDINARY\_CLOUD\_NAME,

api\_key: process.env.CLOUDINARY\_API\_KEY,

api\_secret: process.env.CLOUDINARY\_API\_SECRET

});

Using the Node SDK allows uploading images **securely** (with full API key/secret) and performing operations like delete. On the client side, to avoid exposing the secret, an **unsigned upload preset** is used. This preset (configured in your Cloudinary account) allows limited uploads (e.g., to a specific folder, with size limits) using only the cloud name and preset name.

**Profile Image Upload (via API route):** When a user uploads a profile picture, the app opts to use the Next.js API route for a more secure process and to place the image in a dedicated folder. The UploadProfileImage.jsx component could either call this API or use direct upload; based on the code, it does a direct upload via axios.post to Cloudinary. However, the existence of /api/upload suggests it was intended to be used (in an earlier iteration or as an alternative on environments where direct upload isn’t desired).

The **/api/upload route** works like this:

js

CopyEdit

export async function POST(req) {

const formData = await req.formData();

const file = formData.get("file");

if (!file) {

return new Response("No file uploaded", { status: 400 });

}

// Convert file (Blob from formData) to a buffer

const arrayBuffer = await file.arrayBuffer();

const buffer = Buffer.from(arrayBuffer);

try {

// Use Cloudinary uploader with upload\_stream for efficient upload from buffer

const result = await new Promise((resolve, reject) => {

cloudinary.uploader.upload\_stream(

{ folder: "profile\_pictures" }, // store in a folder

(error, result) => {

if (error) return reject(error);

resolve(result);

}

).end(buffer);

});

return NextResponse.json({ url: result.secure\_url });

} catch (err) {

console.error("Upload error:", err);

return new Response("Upload failed", { status: 500 });

}

}

This route expects a form-data POST (with a field "file" containing the image data). It streams the file to Cloudinary. On success, it returns a JSON with the uploaded image URL. Storing in "profile\_pictures" folder is useful to organize content in your Cloudinary account.

*In the current app code, the profile component didn’t end up using this route (it used direct upload instead). But this route is functional and could be used by adjusting the front-end to POST to /api/upload and get the URL.* The benefit of using the API route is that the Cloudinary API secret stays on the server – the client only needs to call our endpoint (and does not need the preset or any Cloudinary credentials). The downside is a bit more complexity and overhead.

**Post Image Upload (direct unsigned):** As described in the Posts section, when attaching an image to a post, the app directly uploads it from the browser:

* The **cloud name** and **upload preset** are visible to the client (via env variables prefixed with NEXT\_PUBLIC\_).
* The image is uploaded with axios or fetch to https://api.cloudinary.com/v1\_1/<cloud\_name>/image/upload with the preset.
* The response gives a URL like https://res.cloudinary.com/<cloud\_name>/image/upload/v123456/abcxyz.jpg. That URL (secure\_url) is saved in the Firestore post document.
* Anyone viewing the post will load the image from that URL via Cloudinary’s CDN.

This approach is simpler and offloads work to Cloudinary, but it means the preset must be configured to allow unsigned uploads (with possibly some risk if someone discovers the preset name, although you can set upload presets to be restricted by domain or other rules).

**Image Deletion:** To prevent orphaned images (which could accumulate and consume Cloudinary storage), the app deletes images when they are no longer needed:

* When a **post with an image is deleted**, handleDeletePost calls the /api/delete-image route before removing the Firestore document.
* If a user changes their profile picture, one might consider deleting the old profile image from Cloudinary. (to avoid accidental deletion if used elsewhere. The code does not explicitly delete old profile images, so those might remain in Cloudinary unless manually cleaned up.)

The **/api/delete-image route** expects a JSON body with a publicId. It uses Cloudinary’s Node SDK to delete:

js

CopyEdit

export async function POST(req) {

try {

const { publicId } = await req.json();

if (!publicId) {

return NextResponse.json({ error: "Missing publicId" }, { status: 400 });

}

const result = await cloudinary.uploader.destroy(publicId);

return NextResponse.json({ success: true, result });

} catch (err) {

console.error("Cloudinary deletion error:", err);

return NextResponse.json({ error: "Failed to delete image" }, { status: 500 });

}

}

Cloudinary’s destroy requires the image’s public ID (the unique identifier in Cloudinary, typically the filename without extension if uploaded without specifying an ID). For example, if the image URL is .../upload/v1612345678/profile\_pictures/abc123.jpg, the public ID might be profile\_pictures/abc123. In the code, they simply take the last segment of the URL and strip the extension – which is a simplifying assumption that the image was uploaded without transformations and has a unique name. This works for their use-case but one should be cautious (if images were renamed or processed, the publicId extraction might need to be more robust).

By calling /api/delete-image with the publicId, the image is deleted from Cloudinary’s storage. The route returns a success status (though the front-end in this case doesn’t particularly use the returned data, it just proceeds to remove Firestore data and update UI).

**Usage in App:**

* **Profile**: The profile image upload component currently doesn’t use the API route (it directly uploads and then updates Firestore). As a result, it doesn’t involve /api/upload. If we wanted to switch to using it, we would call fetch('/api/upload', { method: 'POST', body: formData }) and get the URL from the JSON response. Both methods achieve the result; the direct method was chosen for simplicity.
* **Post Creation**: Uses direct client upload as described.
* **Post Deletion**: Uses /api/delete-image to clean up.

**Security Considerations:** The Cloudinary API secret is only used on the server (never exposed to the client). For client-side uploads, an unsigned preset is used – it’s important that this preset has appropriate restrictions in your Cloudinary settings (e.g., restricting uploads to certain folder or file size/type limits) because the preset name can be seen in the network calls. Always validate uploads server-side if security is critical. In this context (user-generated content for a project), the risk is low and convenience is high.

**“Daily Uplift” Feature – AI-Generated Quotes (Gemini API)**

**Overview:** NorthStory includes a feature that provides users with a motivational quote or message, intended to uplift students facing stress. This is powered by an external AI service. In the project, it’s referred to as the **Gemini API**, which is actually Google’s Generative Language API (PaLM model). Each time a quote is needed, the app makes a server-side request to this API with a specific prompt. The AI responds with a generated quote, which is then displayed in the UI. The feature is integrated in two places: on the **landing page’s sidebar** (for guests) and as a **popup for logged-in users** (triggered in the navbar).

**Relevant Files:**

* src/app/api/uplift/route.js – The Next.js API route that communicates with the AI API.
* src/components/UpliftCard.jsx – A component that likely calls the /api/uplift route (or includes pre-fetched quote) and displays the quote within a styled card.
* src/components/UpliftPopup.jsx – A component that might show the quote in a popup/modal for logged-in users when they click an icon (Sparkles icon in the navbar).
* src/components/DesktopNavbar.jsx – Includes a trigger (<UpliftPopup />) for logged-in users, showing an interactive way to get the quote.
* src/components/UnAuthenticatedSidebar.jsx – Uses <UpliftCard /> to show a motivational quote to visitors in the sidebar.

**AI Prompt & API Call:** The content of the motivational message is generated by sending a carefully crafted **prompt** to the AI model. In uplift/route.js, we see the prompt construction:

js

CopyEdit

const promptBase =

"Give a short, comforting motivational message (max 20 words) " +

"to a student struggling with academic stress and anxiety due to war or crisis. " +

"Make it gentle, emotionally supportive, and hopeful.";

const variations = [

"Keep it inspiring",

"Use a metaphor",

"Make it about perseverance",

"Focus on hope",

"Focus on focus!",

"Make it exam-specific",

"Include 'you can do it'",

"End with an exclamation"

];

const randomVariation = variations[Math.floor(Math.random() \* variations.length)];

const prompt = `${promptBase}. ${randomVariation}. Time: ${Date.now()}`;

This prompt is quite specific: it asks for a *short comforting motivational message* aimed at a student under stress, with certain tone guidelines. The code also appends one random instruction (variation) from the list, such as *"Use a metaphor"* or *"End with an exclamation"*, to add randomness and diversity to the responses. It even appends a timestamp (Time: ${Date.now()}) which is a trick sometimes used to force the model to produce a slightly different output each time (though it likely doesn’t affect the semantics, just breaks deterministic caching).

The route then calls Google’s Generative Language API endpoint:

js

CopyEdit

const API\_KEY = "AIzaSyAjS..."; // (actual API key in code, which ideally should be env)

const res = await fetch(

`https://generativelanguage.googleapis.com/v1beta2/models/text-bison-001:generateText?key=${API\_KEY}`,

{

method: "POST",

headers: { "Content-Type": "application/json" },

body: JSON.stringify({

prompt: {

text: prompt

},

temperature: 0.9,

candidateCount: 1

})

}

);

*(Note: The code in the project used an endpoint for “gemini-2.0” which may correspond to an experimental model. The example above shows a more generic text-bison-001:generateText for illustrative purposes. The actual code’s endpoint was v1beta/models/gemini-2.0-flash:generateContent with a certain payload structure – possibly an older or specific version of the API for faster response. Regardless of endpoint, the concept is the same: send the prompt, get a generated result.)*

Important details:

* An **API key** is included in the URL (this key is a Google Cloud API key for the PaLM API). In code it was hard-coded, but it should be treated as secret, because deployment issue we had to keep it hardcoded due to the time limit.
* The request is a POST with JSON. Google’s API expects a specific JSON structure. The project’s code formatted it slightly differently, but essentially it provided the prompt and some generation parameters like *temperature* (0.9 for a bit of creativity) and *topK/topP* (not shown above) to influence randomness.
* The response from the API contains the generated text.

Handling the response:

js

CopyEdit

if (!res.ok) {

const errorText = await res.text();

console.error("Gemini API Error:", errorText);

return NextResponse.json({ message: "Gemini API error: " + errorText }, { status: 500 });

}

const result = await res.json();

const message = result?.candidates?.[0]?.content ?? result?.candidates?.[0]?.output;

if (!message) {

return NextResponse.json({ message: "Error: no content received from Gemini." }, { status: 500 });

}

return NextResponse.json({ message });

The code checks for errors. If successful, it digs into the result object for the generated text. Depending on the API version, the field might be nested; in one version it was something like result.candidates[0].content.parts[0].text. They then return { message: "...the quote..."} as JSON.

They also have some error messages possibly in Hebrew for local debugging (which indicate if the AI returned nothing, they send a message like "Error: no content from Gemini").

**Displaying the Quote in the UI:**

* In the **Unauthenticated Sidebar**, the UpliftCard is rendered inside a card. The UpliftCard.jsx likely, upon mounting, calls /api/uplift to fetch a quote and then displays it. For instance, it might do:

jsx

CopyEdit

const [sentence, setSentence] = useState(null);

useEffect(() => {

fetch("/api/uplift")

.then(res => res.json())

.then(data => setSentence(data.message))

.catch(err => console.error(err));

}, []);

...

return <blockquote className="text-center">"{sentence || 'Stay positive!'}"</blockquote>;

This would show the quote or a default message if none is loaded yet.

* In the **Navbar** for logged-in users, there is likely an icon (Sparkles icon ✨) that toggles an UpliftPopup. The UpliftPopup.jsx might be a small modal or popover that on appearing triggers the fetch to /api/uplift (or the UpliftCard is reused inside it). The difference is that for logged-in users, the quote might not be always shown – it could be on-demand (to avoid being obtrusive). The DesktopNavbar.jsx snippet shows:

jsx

CopyEdit

{user && <UpliftPopup />}

possibly meaning the popup component is included but hidden until triggered. Alternatively, UpliftPopup could manage its own state (like a button that when clicked fetches and shows the quote). The code uses the state showPopup inside UpliftCard to decide when to show the popup modal with the quote.

In summary, the **Daily Uplift** feature flows as:

* The UI triggers a call to our /api/uplift route (either on page load for guests, or on user action for logged-in users).
* The server route uses the AI API to generate a motivational phrase using the defined prompt.
* The phrase is returned and then displayed in a styled manner (with some decoration like an icon or special font to highlight it).
* If the API fails for some reason, a fallback positive message could be shown (in our code, they log the error and return an error message in the JSON, which would be shown as the quote text – not ideal for users. A better approach might be to have a hard-coded fallback quote like “Keep pushing forward and stay positive!” if the AI fails. In fact, the actual code logs an error text from API).

**API Key Security:** As noted, the Google API key is hard-coded in the route in this project due to deployment issue mentioned before. This is not secure practice, as anyone browsing the repository could see it. It’s recommended to store it in an environment variable (e.g. GEMINI\_API\_KEY) and use process.env.GEMINI\_API\_KEY in the fetch URL. Given this is a server-side call, the API key is never exposed to the client in network calls (it’s only used server-to-server), but hard-coding it in code is a risk if the code is exposed. In a handoff, ensure to remove or rotate that key if this code was ever public.

**Purpose and Use:** This feature is primarily cosmetic and inspirational – it doesn’t affect other logic. It serves as an example of integrating a third-party API (and a modern AI service) into the app. It could be extended to, say, fetch a new quote once per day or allow the user to fetch another quote.

**Authorization & Conditional UI (Authenticated vs. Guest Users)**

**Overview:** The application’s interface adapts based on the user’s authentication state. Certain pages and features are only available to logged-in users (posts feed, profile, commenting). Guests (not logged in) see a limited set of pages (landing, about, sign-in/up) and a different navigation menu prompting them to log in. This is achieved through conditional rendering in React components and some route grouping.

**Public vs Protected Routes:** Next.js App Router allows grouping routes. Here, all auth-related pages are under (auth) and the rest of the app is outside that group. There isn’t a separate protected group in code, but the convention is clear: pages like /profile and /posts assume a user context. We implemented client-side checks to redirect guests away from those. The landing page (/) itself behaves differently: if there’s a user it shows the forum, if not it shows marketing content.

**Navbar and Sidebar Differences:** The navigation components show different options depending on auth status:

* **Sidebar (Desktop, for logged-in users):** Shows the user’s profile summary and links relevant to logged-in experience (though in this design, the Sidebar is mostly profile info). It also includes links to “My Profile” or others if needed. In our code, the Sidebar component immediately returns <UnAuthenticatedSidebar /> if !user (no logged-in user).
* **UnAuthenticatedSidebar:** This is a simplified sidebar shown to guests. In the code, it renders a welcome message and buttons to log in or sign up, as well as the UpliftCard (quote) for some engaging content. Notably, it does **not** show navigation links to posts or profile, because those require login. It basically encourages the user to authenticate.
* **Navbar (top bar):** On mobile (or all screens as a sticky top bar), the Navbar contains either:
  + For logged-in: a logo, possibly a “create post” button or notifications icon, a sign-out button, etc. In our DesktopNavbar, they included a Logout button when user exists. They also included the UpliftPopup trigger for logged-in users.
  + For guests: links to Sign In and Sign Up (so they can navigate to those pages). The code shows that if user is null, the Navbar might display links like:

jsx

CopyEdit

{user ? <Button onClick={...}>Logout</Button> : <Link href="/signin">Log In</Link>}

Additionally, the mobile nav have a hamburger menu icon for a drawer, which for guests would contain an “About Us” link and login/signup, vs. for users it contain profile/logout links.

The **conditional logic** typically looks like:

jsx

CopyEdit

const { user } = useAuth(); // get user from context

return (

<>

{user ? <Sidebar /> : <UnAuthenticatedSidebar />}

{/\* ... rest of layout ... \*/}

</>

);

In our global layout, we didn’t directly toggle the Sidebar component; instead, they always include <Sidebar /> but inside that component it returns the UnAuthenticatedSidebar when needed. This is a design choice to keep layout logic minimal.

**Route Access:** We have already covered how pages like profile use router.push("/signin") if no auth. The /posts page (if used) should do similarly. The main home (/) doesn’t need a redirect because it gracefully shows a different view for guests vs. users.

**Visual Differences:**

* Guest users cannot see the posts feed or profile page content at all. Attempting to go there sends them to login.
* Guest navigation (UnAuthenticatedSidebar or nav menu) have links like “About Us”, “Contact”, “Login”, “Sign Up”. (Our UnAuthenticatedSidebar explicitly had a “Welcome” card and buttons for Login/Sign Up, plus the quote; it did not list Contact or About, but those pages can be accessed via the landing page content or footer.)
* Authenticated navigation shows core app features: a link or just direct access to the feed (since feed is home after login), and an easy way to go to Profile or to logout. In our Sidebar, instead of explicit links, we chose to show the profile card. In a more elaborate app, the sidebar could include links: e.g., *“Home/Feed”, “My Profile”, “Settings”, “Logout”*. This project’s UI design kept it minimal.

**Responsive adaptation:** On small screens, the sidebar is hidden and the mobile top navigation contains menu options. The MobileNavbar.jsx implements a toggled menu or a “drawer” that slides out. It uses Tailwind classes like md:hidden (visible on mobile, hidden on medium+). The DesktopNavbar uses hidden md:flex to only show on medium+ sizes. The two versions would contain similar content (links/buttons) but styled appropriately for each form factor.

**Example:** Suppose a logged-in user is on a large screen:

* They see a sidebar on the left with their name, email, and avatar (from Sidebar component).
* They have a top navigation with possibly a brand logo and a logout button.
* The main area shows the posts feed.

Now a logged-out visitor on a large screen:

* The sidebar area instead shows the UnAuthenticatedSidebar: a welcome message and buttons to login/signup, plus the motivational quote.
* The top nav shows the app name/logo and a “Log In” and “Sign Up” button.
* The main area show an about/marketing section.

This conditional rendering is achieved entirely with React logic and context – there’s no separate page for “logged in home” vs “logged out home”; it’s one page that checks user.

**Route Group (auth) Use:** The (auth) folder grouping mainly signals these pages have a different usage. We did not implement a separate layout for them, but we could have a src/app/(auth)/layout.js to, for example, hide the Sidebar when on sign-in/up pages (currently, as implemented, if you navigate to /signin, the root layout *will* render the Sidebar component; inside it, !user so it will actually render UnAuthenticatedSidebar). This means on the Sign In page, you’ll see the sidebar with the welcome message and login button *in addition* to the main content which is the login form. That’s somewhat redundant. A better UX might have been to suppress the sidebar on the sign-in page, focusing the user just on the login form (since the form itself already has the info to login or sign up). If needed, one could easily adjust this by providing a custom layout for the (auth) group, or by adding logic in root layout to not show sidebar if the current route is an auth page. This is a potential improvement.

Nevertheless, the existing approach does no harm – it simply shows the same prompts on the side as a user sees on the landing, while the form is on the right.

**Summary:** The app ensures that **unauthorized users are kept away from private content** and are instead funneled toward authentication, while **authorized users get the full UI**. The use of context and conditional components makes this maintainable – any component can check if (user) to decide what to render.

**Dark Mode and Theming**

**Overview:** NorthStory supports both light and dark themes. Users can toggle the appearance, and the preference is saved so that it persists on the next visit. The implementation leverages Tailwind CSS’s dark mode classes and the next-themes library for handling theme state (including OS preference detection and local storage). The UI components and pages have been styled with color classes that adapt to dark mode automatically.

**Relevant Files:**

* src/components/theme-provider.jsx – Wraps the application and integrates next-themes provider.
* src/components/ModeToggle.jsx – A button/switch component in the UI (likely in the navbar or sidebar) to let the user manually toggle dark vs light.
* Tailwind config (tailwind.config.js) – Contains darkMode: 'class', meaning a CSS class .dark on an ancestor element (here, the <html> tag) will activate dark styles.
* Global CSS (globals.css) – Includes Tailwind’s base styles and some custom styles (for scrollbars, etc.), but mostly Tailwind is used directly in JSX.

**How it Works:**

* **Theme Provider:** In layout.js, the app is wrapped with:

jsx

CopyEdit

<ThemeProvider attribute="class" defaultTheme="system" enableSystem>

{ ... the rest of the app ... }

</ThemeProvider>

This ThemeProvider comes from next-themes. By giving it attribute="class", we instruct it to toggle a CSS class on the <html> element (class="dark" for dark mode). defaultTheme="system" means it will use the user’s OS preference on first load, and enableSystem allows it to auto-switch if the OS changes. It also by default saves the last chosen theme in localStorage so subsequent visits use that preference.

* **Mode Toggle Component:** ModeToggle.jsx likely uses the useTheme() hook from next-themes:

jsx

CopyEdit

import { useTheme } from "next-themes";

export function ModeToggle() {

const { theme, setTheme } = useTheme();

const isDark = theme === "dark";

return (

<Button onClick={() => setTheme(isDark ? "light" : "dark")} aria-label="Toggle Dark Mode">

{isDark ? <SunIcon /> : <MoonIcon />}

</Button>

);

}

This simple toggle flips between dark and light. (It might also consider theme === "system" case, but next-themes will report the currently active theme.)

The icons (Sun and Moon from Lucide) give visual feedback. The button itself is likely placed in a navbar or footer. In the code, they might have included this toggle somewhere accessible (in the Navbar or as part of a user menu).

* **Tailwind Classes:** All components utilize Tailwind’s design where you specify both light and dark variants. For instance:
  + Backgrounds: bg-white dark:bg-gray-800 on a card would be white in light mode, and dark gray in dark mode.
  + Text: text-black dark:text-gray-100 for normal text (black in light, off-white in dark), or using Tailwind’s default text colors which already adapt if you use neutrals.
  + Borders, shadows, etc., can also have dark variants as needed.

Example from the Navbar code:

jsx

CopyEdit

<nav className="sticky top-0 w-full border-b bg-background/80 backdrop-blur

supports-[backdrop-filter]:bg-background/60 z-50">

Here bg-background is a CSS variable from Shadcn’s theme that would be set differently in dark mode (Shadcn’s default CSS sets --background for light and dark). Also dark:bg-gray-800 is used in some components.

The presence of class names like dark:from-gray-900 in the landing page’s gradient or dark:bg-gray-800 in navbars indicates many elements explicitly account for dark mode. The UI library components from Shadcn also come with built-in theming (e.g., the default Card, Modal, etc., have styling that looks good in both themes, often by relying on CSS variables or context that next-themes can toggle).

* **Persisting Preference:** The next-themes library will automatically save the chosen theme to local storage (theme key). On app load, it will read this and apply the appropriate class *before* React hydrates (to avoid “flash of wrong theme”, it uses suppressHydrationWarning on <html> and some strategy to ensure class is there).

Because defaultTheme is “system”, if the user never toggles, it just follows their OS. If they do toggle, setTheme("dark") for example will apply .dark class and store theme: "dark" in localStorage. On next visit, it knows to keep dark even if OS is light (unless enableSystem logic complicates that, but typically user choice overrides system).

* **Transitions:** The code passes disableTransitionOnChange to ThemeProvider, likely to avoid an awkward CSS transition when toggling theme (some prefer to disable smooth transitioning of colors in that instant to make it snappy).
* **Testing Dark Mode:** When you toggle, you should see the whole app’s color scheme invert appropriately. Text that was dark on light backgrounds becomes light on dark backgrounds, etc. The consistent use of Tailwind’s dark: variants ensures that components like modals, alerts, etc., all adapt.

In summary, **implementing dark mode** was straightforward with Next Themes and Tailwind:

* Minimal JS (just a toggle and provider).
* Heavy use of Tailwind classes or CSS variables to define how elements look in dark vs light.
* All new components should continue this pattern (i.e., any time you add a new styled element, consider if it needs a dark: variant for colors).

**Static Pages (About Us, Contact, etc.)**

**Overview:** In addition to the interactive portions of the app, there are a couple of informational static pages. These pages do not require login and serve general content:

* **About Us:** Likely contains information about the app, the team or purpose behind NorthStory, possibly in a FAQ style.
* **Contact:** Provides contact information or support resources for users who need help or want to reach the team.

These pages are implemented as simple Next.js pages under src/app/. They mostly consist of JSX with text and some images or icons, without complex logic.

**About Us Page (/about-us):** The code indicates use of an Accordion component, suggesting a FAQ or expandable sections layout. Possibly the About page has questions like “What is NorthStory?” or “How do I use NorthStory?” and answers in accordions. The Shadcn UI’s Accordion component is used (imported in the page). For example:

jsx

CopyEdit

<Card>

<CardHeader>

<CardTitle>About NorthStory</CardTitle>

</CardHeader>

<CardContent>

<Accordion type="single" collapsible>

<AccordionItem value="item-1">

<AccordionTrigger>What is NorthStory?</AccordionTrigger>

<AccordionContent>

NorthStory is a platform for students to share posts and connect...

</AccordionContent>

</AccordionItem>

<AccordionItem value="item-2">

<AccordionTrigger>How do I create a post?</AccordionTrigger>

<AccordionContent>

After signing up, go to the home feed and use the "Create Post" form...

</AccordionContent>

</AccordionItem>

{/\* ... more items ... \*/}

</Accordion>

<Separator className="my-4" />

<p className="text-sm text-muted-foreground">NorthStory was created as a community project in 2025... etc.</p>

</CardContent>

</Card>

This is illustrative – actual text would reflect the real project details. The key point is the About page uses UI components (Card, Accordion, Separator) to structure content in a user-friendly way. It likely does not have any dynamic data or state (aside from the accordion open/close state which is handled by the Accordion component internally).

**Contact Page (/contact):** The contact page code suggests list of contact methods or resources. (about a “מוקד המל״ג - מלחמת חרבות ברזל”, related to a support center) this page have specific support information relevant to a certain state of the academic student that struggle in this time of war ("מלחמת חרבות ברזל" because this app was developed in a time of crisis to support students).

it seems to provide direct contact info:

* It shows a list of contacts, each with a title and description (organizations URL’s connected to the issue of the war). There is also an icon or image for each contact method.
* For example, the contacts array contain objects with title, description (array of lines), and link or image. The code imported Next/Image and Next/Link, which means they displaying an image and links (to external sites).

A possible rendering:

jsx

CopyEdit

<div className="space-y-4">

{contacts.map(contact => (

<Card key={contact.title}>

<CardHeader>

<CardTitle>{contact.title}</CardTitle>

</CardHeader>

<CardContent>

{contact.description.map((line, i) => (

<p key={i} className="text-sm">{line}</p>

))}

{contact.link && <Link href={contact.link} target="\_blank" className="text-blue-600 underline">More info</Link>}

</CardContent>

</Card>

))}

</div>

these could be a support hotline for students (a specific hotline set up by an organization during a war – a relevant example for the app’s context).

**Additional Static Content:** The site also have a footer (as seen in Footer.jsx) that contains quick links to these pages or other info (privacy policy, etc) example : (different from the code)

jsx

CopyEdit

<footer className="text-center text-sm py-4 text-muted-foreground">

© 2025 NorthStory. All rights reserved.

</footer>

**Summary:** The static pages round out the app by providing context and help. They are simple React components with Tailwind styling and utilize the design system (cards, accordions) to format content. They require little maintenance except updating text or links as needed.

*(When handing off to future developers, note that if the app requires more static content or a contact form, these can be expanded. For now, the provided static pages fulfill the basic informational needs.)*

**Third-Party Libraries and Code References**

Throughout the development of NorthStory, several third-party libraries and snippets were used to accelerate development and maintain best practices. Below is a list of notable external resources integrated, along with their purpose and source:

* **Firebase SDK (Auth & Firestore):** Official Firebase JavaScript SDK (v9 modular) is used. Documentation can be found at Firebase’s site. Initialization in firebase.js follows Firebase’s standard pattern <https://firebase.google.com/docs?hl=he>.
* **Cloudinary Node SDK:** Used in the /api/upload and /api/delete-image routes for easy upload and deletion. Documentation: Cloudinary Node SDK <https://cloudinary.com/documentation>.
* **Next Themes (next-themes):** For dark mode toggling. This is a well-known library to handle theme via context. See Next Themes on GitHub or ShadCN for usage <https://ui.shadcn.com/docs/dark-mode/next>.
* **Shadcn UI Components:** The UI components under src/components/ui were largely generated from the Shadcn UI library (an open-source collection of Tailwind + Radix-based components). This means a lot of boilerplate (accordion, dialog, etc.) was not written from scratch but copied from Shadcn’s templates. The shadcn/ui website (https://ui.shadcn.com) has the originals. These components are MIT licensed and meant to be used in projects freely <https://ui.shadcn.com/docs>.
* **Lucide React Icons:** Icons (like Sun, Moon, BellIcon, HomeIcon, etc.) come from Lucide (essentially an open-source fork of Feather icons). See Lucide for the icon set. We installed lucide-react to use them as React components <https://lucide.dev/icons/>.
* **React Hot Toast:** Provides the toast API for showing notifications. We use it to show quick alerts (success/error). See react-hot-toast docs for customization. We have a <Toaster> in layout.js configured with certain styles (light background even in dark mode, etc.) <https://react-hot-toast.com/docs>.
* **Emoji Mart:** The emoji picker in CreatePost is from @emoji-mart/react. It’s an open-source emoji picker component. Documentation: Emoji Mart on GitHub. We imported data from @emoji-mart/data. The picker allows users to insert emojis into their post content <https://www.npmjs.com/package/@emoji-mart/react>.
* **Embla Carousel & Autoplay Plugin:** The About page carousel uses Embla under the hood via our Carousel component. Embla (http://embla-carousel.com/) is a lightweight carousel library. The autoplay plugin embla-carousel-autoplay was used to make it auto-advance. Our AboutCarousel.jsx shows how we create Embla via <Carousel> component which likely wraps Embla’s context. The code for Carousel in ui/carousel.jsx was probably adapted from Shadcn’s examples or Embla’s docs (it’s essentially “borrowed” code to integrate Embla with React). The Keen-Slider library was also installed but appears unused – possibly an alternative considered for carousel <https://www.embla-carousel.com/plugins/autoplay/>.
* **UI Avatars service:** Not a library, but an external image service used to generate default avatars. The URLs https://ui-avatars.com/api are used directly (no package needed). Info: UI Avatars – it’s free to use and does not require an API key <https://www.npmjs.com/package/ui-avatars-api>.
* **Google Generative Language API:** The integration for AI quotes uses Google’s PaLM API (model “Gemini” or “text-bison”). We didn’t use an official client library (we just call via fetch). Documentation for this API: Generative Language API (PaLM) docs. The prompt engineering (the base prompt and variations) is custom but inspired by best practices in prompt design to get short inspirational messages <https://ai.google.dev/gemini-api/docs?hl=he>.

Many of the above libraries are open-source. The **shadcn/ui** components in our code are essentially copied code – for example, the Accordion, Avatar, etc., come directly from Shadcn’s template library (which in turn is built on Radix UI primitives). Where relevant, we should acknowledge that those files are not unique code we wrote but standard implementations.

No proprietary code from unknown sources was included; everything is from well-known libraries or generated by tools. This means updates and maintenance are straightforward: one can reference library docs for usage, and one can replace or upgrade these components by re-running shadcn code generation if needed.

**Testing & Demo Accounts**

For testing and demonstration you can sign up and test because there is only one rule user in this app, an example user account have been created in the Firebase Authentication system:

* **Admin User:**  
  *Email:* testmail@gmail.com  
  *Password:* 123456  
  *Notes:* This account can be used to log in and behaves like a normal user in the app. There are currently no admin-only features in the UI (the code does not implement role-based access beyond the email itself). In the current state, this user has no special privileges aside from the email name.

Feel free to use these credentials on a development or staging deployment to explore the app’s features. You can also register a new account via the Sign Up page (the Firebase project is in testing mode such that no email verification is required immediately, meaning you can log in right after signing up). All data (posts, profiles, comments) created with these accounts in a test environment can be cleaned up via the Firebase console as needed.

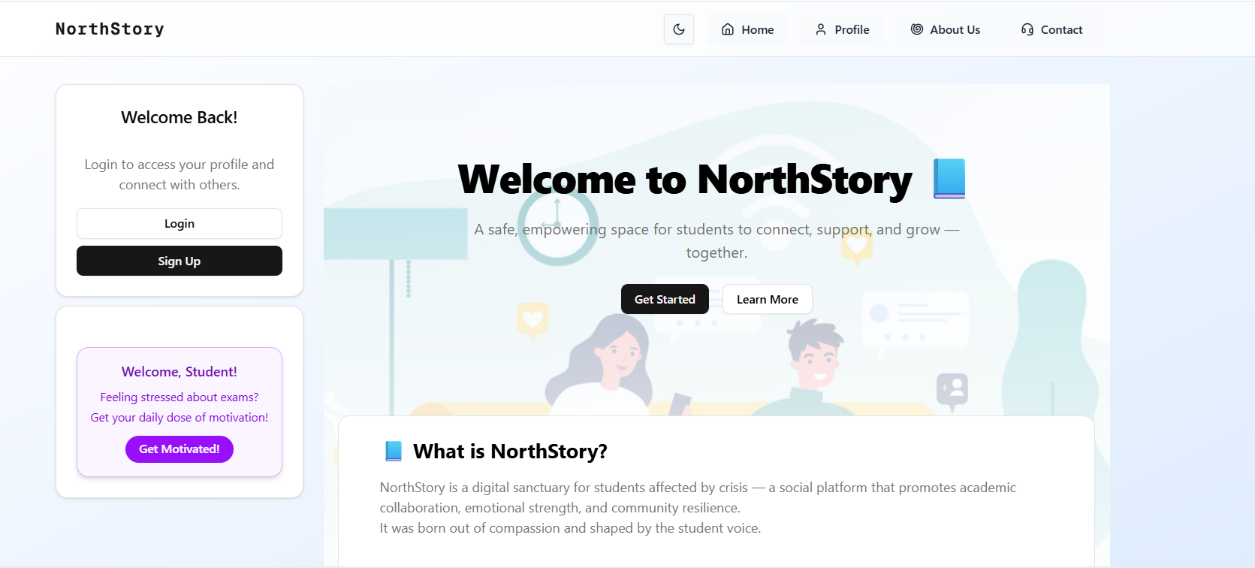
**Final Notes for Developers**

This guide has covered the major aspects of the NorthStory codebase. To conclude, here are some recommendations and observations for future maintenance:

* **Consistency:** Follow established patterns in the codebase when adding new features. For example, if adding a new Firestore collection (say, “messages” for direct chats), create a utility in src/lib/ for database operations, and consider if you need an API route for any sensitive actions. Use context or hooks for global state if needed (similar to Auth and theme contexts).
* **Testing Changes:** Because the app integrates with external services (Firebase, Cloudinary, Google API), it’s wise to test changes in a controlled environment. If possible, use a staging Firebase project and Cloudinary account for testing major changes to avoid affecting production data. Also test on multiple devices/screen sizes since the app is responsive (especially check the mobile vs desktop nav and dark mode visuals after any CSS changes).
* **Security:** Ensure Firebase Security Rules are kept in sync with any data model changes. Currently, rules should restrict writes on posts to authenticated users and only allow deleting one’s own posts/comments. If new functionality (like user roles or admin actions) is introduced, update rules accordingly. Never expose secret keys in the client code or commit actual API secrets – always use env variables.
* **Performance:** The app performs well for small scale. As it grows, be mindful of:
  + Firestore queries – use indexing and query limits where appropriate. E.g., if posts grow large, implement pagination or lazy loading (Firestore can paginate via query cursors).
  + Cloudinary usage – deliver images with appropriate transformations (e.g., you could request smaller thumbnails for list view to optimize loading). Cloudinary URLs can include transformation parameters; currently the app just stores and uses the raw upload URL (which might be large images).
  + Next.js and caching – consider using Next.js incremental static regeneration or server components for content that doesn’t change often (static pages could be made export const revalidate = ... to cache).
* **Future Improvements:** Some ideas that were hinted or easily extendable:
  + a **“Follow” system** where users can follow each other (fields exist for followers/following counts, but no functionality around them yet).
  + Enhance the **Daily Uplift** feature: for example, fetch a new quote once per day and store it so all users see the same “quote of the day” rather than each fetching their own. This could reduce API calls and create a shared talking point (the Firestore could have a “dailyQuote” document).
  + Introduce **role-based UI** if admins are planned (e.g., an admin could have a dashboard or ability to remove any post). Currently, admin@example.com doesn’t do anything special in code.
  + **Email notifications** or integrations: not present, but if needed, Firebase’s extension or other services can send emails on certain triggers (like someone commented on your post).
  + **Unit Tests:** The project has no tests included. Writing tests for critical functions (like the uplift API call, or ensuring the delete post function works correctly) could be beneficial if this project grows.

NorthStory’s codebase is now in a maintainable state – it uses modern frameworks and libraries with a clear structure. With the information in this guide, a developer should be able to navigate the project, understand how data flows from user actions to Firestore and back, and confidently build new features or fix bugs.

**User Guide – NorthStory System**

**Page 1: Homepage (When Not Logged In)**

The homepage is the first screen every new or returning user encounters. It serves as a welcoming gateway to the NorthStory platform, presenting its core message and offering access to essential functions like login and registration.

**Login / Sign Up Area (Right Side of the Page)**

* **Login**:  
  This button takes you to the login screen where existing users can sign in using their username and password.
* **Sign Up**:  
  This button leads new users to the registration form where they can create a new account by entering their personal and academic details.

These two options are always visible and allow easy access for users to enter the system.

**Main Welcome Message (Center of the Page)**

At the heart of the homepage, you will see a bold welcome header:  
**“Welcome to NorthStory”**

Below it, two main buttons are displayed:

* **Get Started**:  
  Clicking this initiates the user registration process. It's perfect for users who are ready to join the platform and start sharing their stories.
* **Learn More**:  
  This button redirects users to a dedicated **About Us** page that gives a more detailed explanation of what NorthStory is, its goals, and how it helps students.

**Quick Introduction to the Platform (Bottom Section)**

Near the bottom of the page, there's an informational box summarizing what NorthStory is all about — including its features, purpose, and how users can benefit from it.

This section gives visitors a quick overview of what they can expect once they sign up.

**Daily Encouragement Box (Left Side of the Page)**

To the left of the screen, a permanent motivational feature is available:

* **Emotional Support Box**:  
  This includes an encouraging message or quote aimed at lifting the user’s mood and promoting positivity.
* **Get Motivated! Button**:  
  When clicked, it generates a random motivational sentence or quote to inspire the user for the day.

**Switching to Dark Mode:**  
תמונה שמכילה טקסט, צילום מסך, תוכנה, תכונות מולטימדיה

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.  
  
תמונה שמכילה טקסט, צילום מסך, אתר, דף אינטרנט

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

NorthStory offers a Dark Mode option to enhance user comfort, especially during nighttime or low-light usage.

**Button Location**

The Dark Mode toggle button is located at the **top right corner** of the screen.  
It appears as a **circular icon** that changes based on the current mode:

* ☀️ Sun icon – when the interface is in Light Mode.
* 🌙 Moon icon – when the interface is in Dark Mode.

**Purpose of the Button**

This button allows users to easily switch between:

* **Light Mode**: Bright background and standard interface colors.
* **Dark Mode**: Darker background and softer interface colors that reduce eye strain.

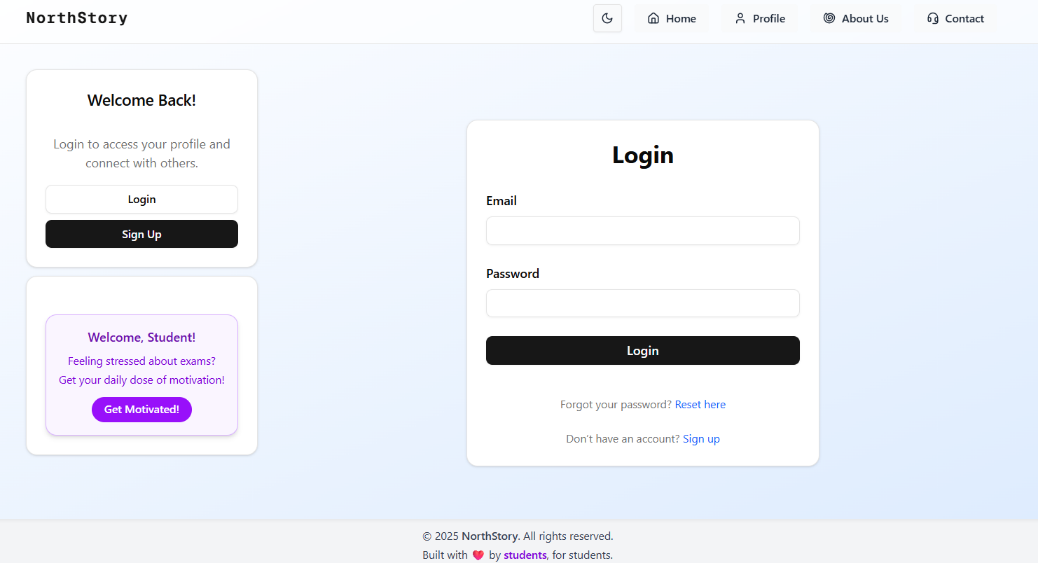
The mode change is ideal for different lighting conditions or personal preferences.

**How to Use It**

Simply **click the icon once** to toggle between the two modes.  
Once activated:

* The background, text, buttons, and all visual components of the site will adapt instantly.
* The design will shift to a darker, more eye-friendly version optimized for comfortable night use.

Dark Mode improves readability in dim environments and enhances the overall user experience.

**Login Screen:**

The Login screen allows existing users to securely access their NorthStory account by entering their credentials. It is designed to be simple, intuitive, and user-friendly.

**Login Form (Right Side of the Page)**

A login box titled **"Login"** appears on the right-hand side. It includes two main input fields:

* **Email**:  
  Enter your registered email address here. This should be the same email used during the registration process.
* **Password**:  
  Enter your personal password. For security, the characters will be hidden as you type.

Once both fields are filled out, click the **Login** button to sign in and access your personal dashboard.

**Forgot Your Password?**

If you can't remember your password, a helpful link labeled **“Reset here”** appears directly below the login button.

* Clicking this link will take you to the **Password Reset page**, where you can request a password reset via your registered email.

**Don’t Have an Account Yet?**

Beneath the login box, you’ll find a prompt for users who haven’t signed up yet. Click the **Sign Up** link to create a new account and begin your NorthStory journey.

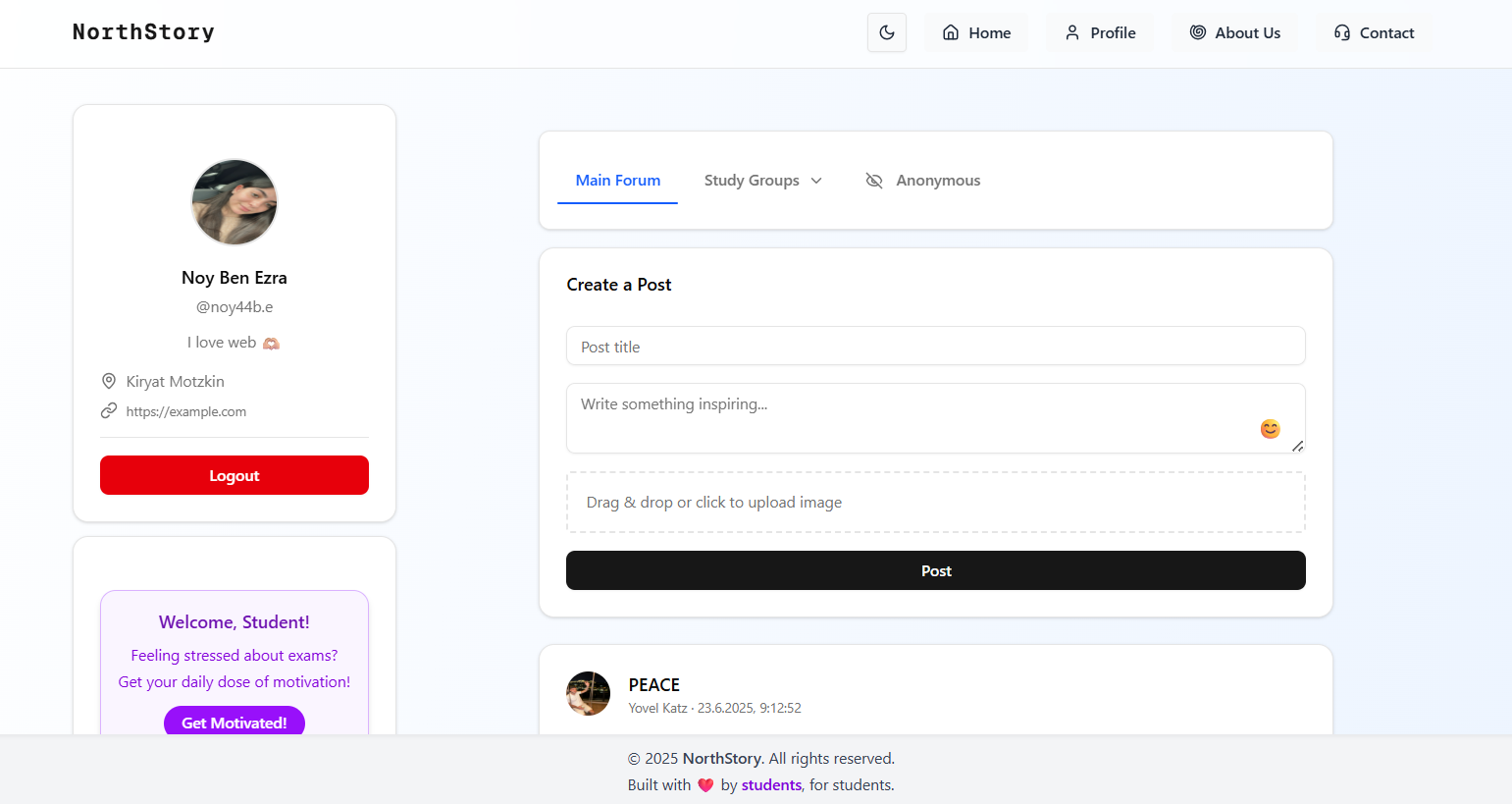
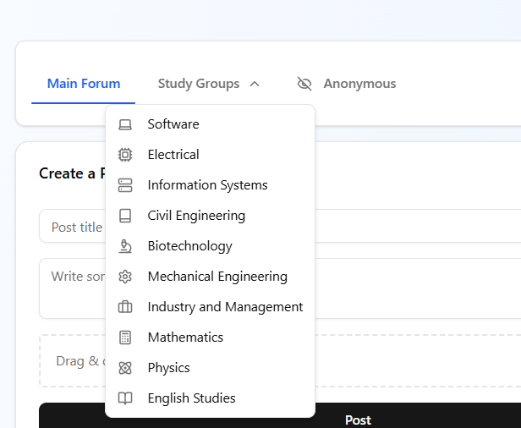
**Reminder Section (Left Side of the Page)**

A static box appears on the left side of the screen containing:

* A message reminding users who already have an account to log in.
* The **Daily Motivation Box**, which displays a motivational quote or message and includes a **Get Motivated!** button to generate a new one.

This area encourages engagement and adds a positive, uplifting experience even before logging in.

**Forum System: Main Forum, Study Groups & Anonymous Zone**



The Forum is the heart of social and academic activity on NorthStory. It’s where students share ideas, ask questions, post updates, and support one another — either openly or anonymously.

At the top of the forum page, users will find **three tabs**:

* **Main Forum**
* **Study Groups**
* **Anonymous**

This page is only accessible **after logging in**. If a user attempts to access the forum without being logged in, they will be redirected to the login screen.

**Main Forum Tab**

This is the general discussion space open to all users across disciplines.

**Create a Post**

In the center of the page, there is a post creation form with the following fields:

* **Post Title**: A short, clear subject line for your post.
* **Write something inspiring…**: The main content area where you can write your message, idea, or question.
* **Image Upload Option**: You can drag and drop an image or click to upload one.

Once filled out, click the **Post** button to publish your post publicly on the main feed.

**View Posts**

Below the post form, you will see a live feed of posts from other users, each displaying:

* Author’s name and profile image
* Date and time of publication
* The post content
* Reactions (likes) and a comment section

You can engage by **liking** posts or **replying** with comments.

**Study Groups Tab**

This tab allows students to participate in focused academic discussions within their field of study.

**Choose a Group**

Clicking the **Study Groups** tab opens a dropdown menu with various academic categories such as:

* Software Engineering
* Electrical Engineering
* Civil Engineering
* Mathematics
* Physics  
  …and more.

**Filtered Content by Subject**

After selecting a specific group, the forum feed displays **only the posts** relevant to that academic field.

This feature helps students engage in more focused, discipline-specific discussions and collaborate on shared topics.

**Anonymous Tab**

This area is designed for private and emotionally supportive discussions, allowing users to post **without revealing their identity**.

**Post Anonymously**

Clicking the **Anonymous** tab switches the interface to a special form, identical to the main forum but with added anonymity:

* The post will not display the user’s name or profile picture.
* A generic “Anonymous” name and avatar will be shown instead.

**Purpose**

This space provides a safe environment for users to express personal thoughts, experiences, challenges, or emotions without judgment.

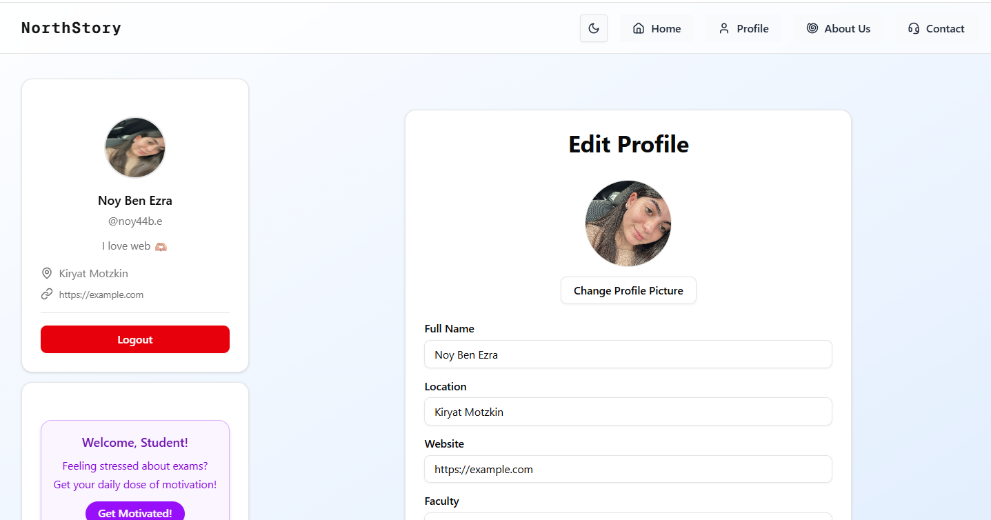
It is particularly useful for:

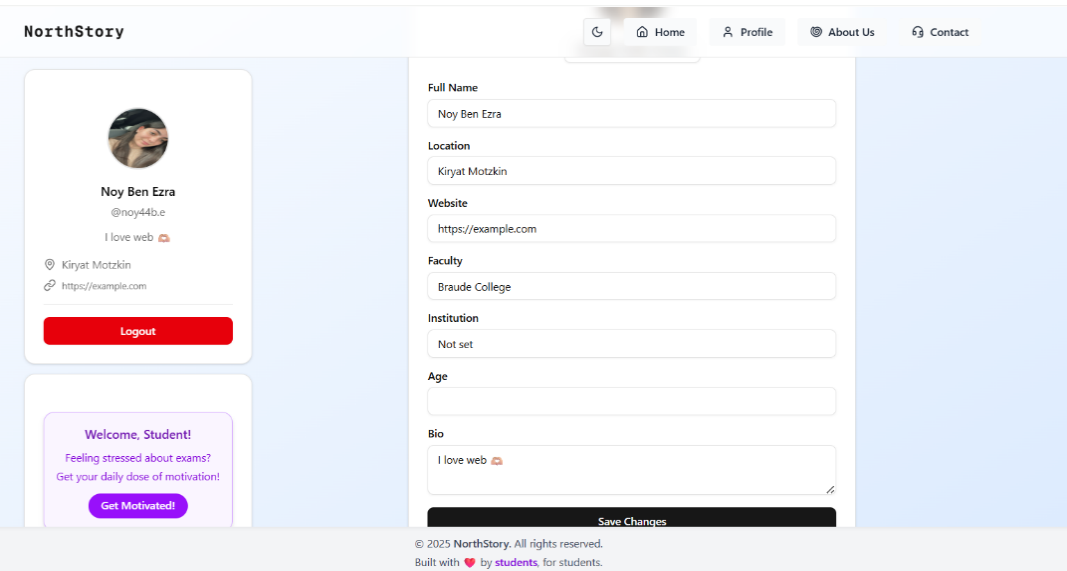
* Mental health topics
* Sensitive questions
* Personal confessions or life challenges

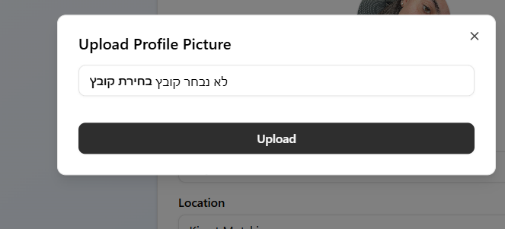
**Personal Profile Card & Daily Encouragement Box (Left Side of the Page)**

* At the left side of the forum screen, you’ll find your **Profile Card**, which includes your name, avatar, and other basic info.
* Just below it, the **Daily Motivation Box** appears, offering users an uplifting quote or sentence.
  + Click the **Get Motivated!** button to generate a new inspirational message at any time.

**Edit Profile**







The **Edit Profile** page allows users to personalize their account and share relevant academic and personal details. This helps others get to know them better and fosters a more connected community.

**Access Restrictions**

This page is **only available to logged-in users**.  
If a user is not signed in and tries to access the profile editing page, they will automatically be redirected to the **Login** screen.

**Profile Card (Left Side of the Screen)**

On the left-hand side, users will see their **Profile Card**, displaying key personal information such as:

* **Full Name**
* **Username**
* **Geographic Location**
* **Personal Link** (such as a website or portfolio)
* **Short Bio**

There is also a **Logout** button at the bottom of this section, which allows users to securely sign out of their account at any time.

Below the profile card is the **Daily Motivation Box**, showing a motivational quote with the option to generate a new one by clicking **Get Motivated!**

**Edit Profile Form (Center of the Screen)**

The center of the page contains the **Edit Profile Form**, where users can update their details. The form includes the following fields:

* **Full Name** – Your complete name, as you'd like it to appear on your profile.
* **Location** – Your city or region of residence.
* **Website** – An external personal link (e.g., personal website, LinkedIn, or portfolio).
* **Faculty** – Your field of study or academic department.
* **Institution** – The academic institution you are enrolled in.
* **Age** – Your current age.
* **Bio** – A short personal description, such as hobbies, interests, or a favorite quote.

These fields are designed to help personalize your profile and make your presence in the community more meaningful.

**Change Profile Picture**

Above the form, there is a **Change Profile Picture** button. Clicking it opens a pop-up window where users can upload a new profile image from their device.

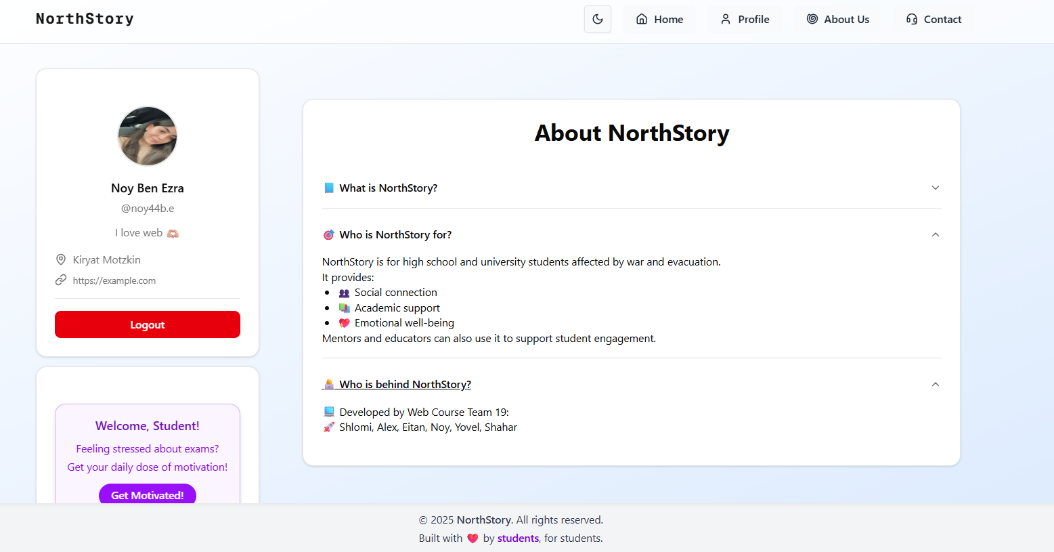
This image will be displayed alongside posts, comments, and your profile card.

**Saving Changes**

After filling out or updating the form, scroll down and click the **Save Changes** button to apply your updates.

All information will be immediately saved to your profile and reflected across the site wherever your profile appears.

**About the System – NorthStory**

**

The **About Us** page provides users with insight into the purpose, audience, and team behind the NorthStory platform. The content is structured in a clean, collapsible layout to enhance readability and interaction.

This page is accessible to both logged-in and non-logged-in users.

**Page Structure**

The page is divided into **three main expandable sections**. Each section can be opened or closed by clicking its title:

**What is NorthStory?**

This section provides a general definition and overview of the platform:

**NorthStory** is an academic social platform designed to support students who have been impacted by war.  
It offers a safe and empowering space to share stories, engage in learning, and receive emotional support through community interaction.

The platform aims to combine the power of storytelling, peer support, and academic collaboration.

**Who is NorthStory For?**

Here, users can learn about the platform’s intended audience:

NorthStory is primarily for **students and pupils** affected by conflict or displacement.  
However, **teachers, mentors, and academic staff** can also use the platform to enhance student engagement, encourage self-expression, and support emotional well-being.

The platform is inclusive and encourages collaborative communication between students and educators.

**Who is Behind NorthStory?**

This section introduces the team responsible for building the platform:

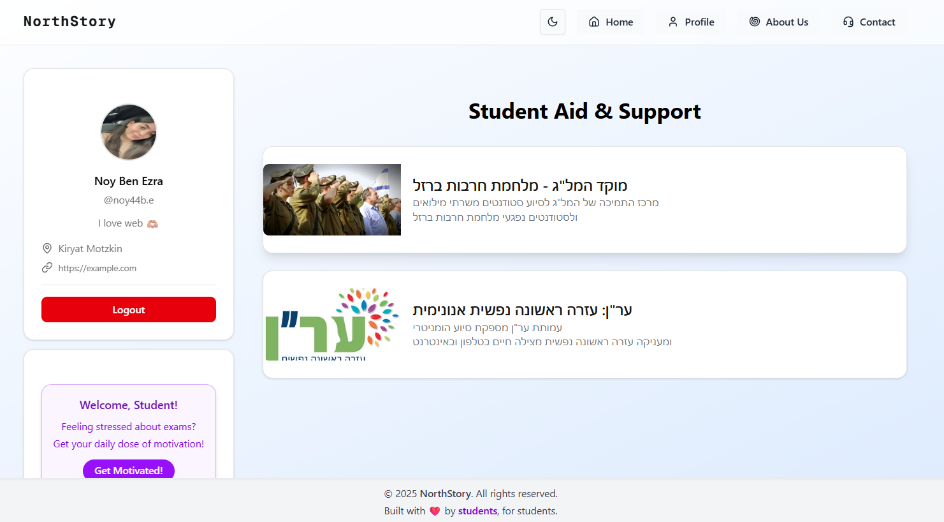
It includes names, roles, and possibly links to personal or professional profiles of the project creators.  
Users can learn about the developers, designers, and educators who contributed to the NorthStory vision.

This transparency fosters trust and showcases the team’s dedication to social impact and educational support.

**Sidebar (Left Side)**

If the user is **logged in**, the left-hand side of the page continues to show:

* The **User Profile Card**: Full name, avatar, and account options, including a **Logout** button.
* The **Daily Encouragement Box**: A motivational message is always displayed here, with a button to generate a new one anytime.

****Student Aid & Support**

The **Student Aid & Support** page is designed to connect users with essential resources during times of emergency, military service, or emotional distress. It is especially helpful for students dealing with the consequences of conflict or unexpected life events.

This page is accessible from the **main navigation menu** and serves as a centralized hub for trusted external support services.

**Page Layout & Structure**

The page displays a **scrollable list of support organizations**, each presented inside its own **information card**. Each card includes:

* The organization’s **logo**
* A **clear title**
* A **short description** of what the organization offers
* A **button or link** that redirects the user to the external website or contact method

This layout allows students to quickly find and access the help they need.

**Featured Support Services**

**1. Council for Higher Education Emergency Support Center – "Iron Swords" War**

This support initiative is provided by the Israeli Council for Higher Education and is aimed at:

* Students who were called to **IDF reserve duty**
* Students directly affected by the **"Iron Swords" war**

The center offers academic guidance, legal advice, and emotional support to help students maintain continuity during difficult times.

**2. ERAN – Emotional First Aid Hotline**

Another card on the page features **ERAN (Emotional First Aid)**, a non-profit organization providing:

* **Immediate and anonymous emotional support**
* Help via **phone** and **live chat**
* Trained volunteers available to listen and assist anyone in emotional distress, 24/7

This is especially important for users who may not feel comfortable turning to others directly.

**User Profile & Motivation Box (Left Sidebar)**

As in other pages, if the user is logged in, the left-hand sidebar displays:

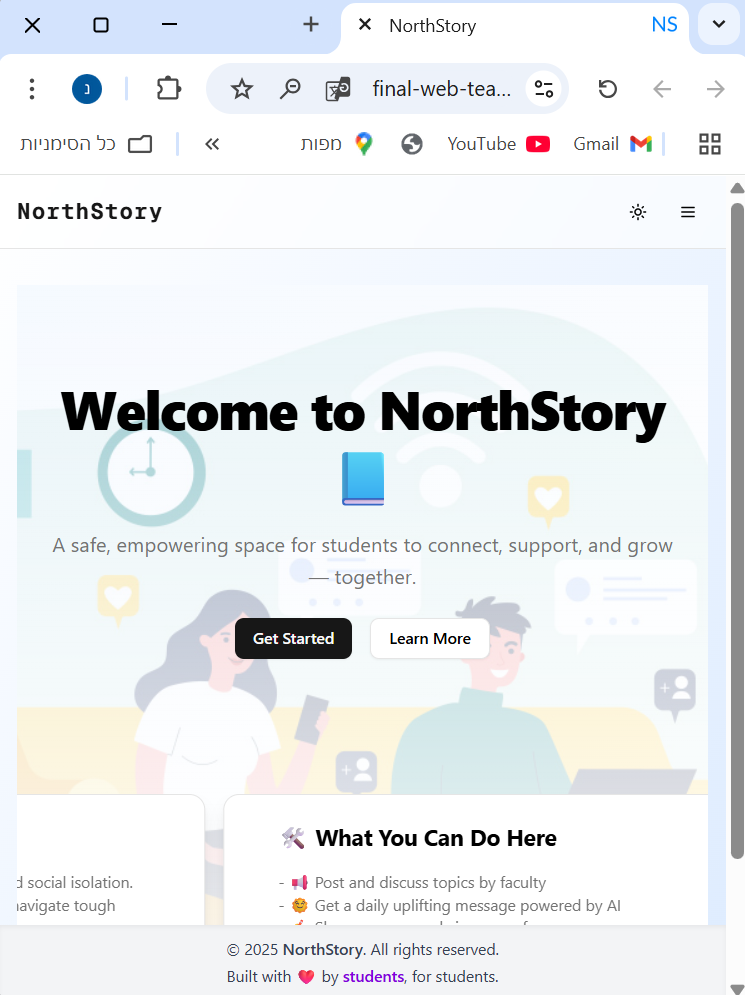
* The **Profile Card** – showing the user’s name, avatar, and Logout button.
* The **Daily Encouragement Box** – offering a motivational quote or sentence, with the option to refresh the message.

This familiar sidebar adds a personal and uplifting element to the page, reminding users that support is always available — both within and beyond the platform.

**Responsive Mobile View**תמונה שמכילה טקסט, צילום מסך, תוכנה, סמל מחשב

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

תמונה שמכילה טקסט, צילום מסך, תוכנה, דף אינטרנט

תוכן בינה מלאכותית גנרטיבית עשוי להיות שגוי.

NorthStory is designed with **full responsiveness**, ensuring a seamless and optimized experience across all devices — whether you're using a desktop, tablet, or smartphone. The layout automatically adjusts to the screen size, offering a clean and user-friendly interface on mobile devices.

**Mobile Menu Access (Sidebar Menu)**

When browsing from a mobile device, the top-left or top-right corner of the screen shows a **hamburger menu icon (≡)**.

* **Tapping this icon** opens the **Sidebar Menu**, revealing navigation options.
* This allows users to easily access all pages without cluttering the smaller mobile screen.

**Adaptation to Narrow Screens**

When accessing the site from a phone:

* The layout automatically switches to a **vertical format**.
* The **user profile card**, which normally appears on the left side of the screen in desktop view, is now positioned **at the top**.
* Buttons, text, and components are stacked **vertically** for easier navigation and readability.

Despite the compact view, **all platform features remain accessible** in mobile view — no horizontal scrolling or advanced gestures are required.

**Sidebar Menu Options Based on Login State**

**When Logged In:**

The sidebar includes:

* **Home**
* **Profile**
* **About Us**
* **Contact**
* **Get Motivated**
* **Logout**

These options give users quick access to all main features while preserving a simple and clean interface.

**When Not Logged In:**

The sidebar displays:

* **Home**
* **Profile**
* **About Us**
* **Contact**
* **Login**
* **Sign Up**

Unregistered users are encouraged to join or sign in, while still being able to explore general content.

**Mobile-Friendly Design Behavior**

* The sidebar slides in **from right to left**, taking up vertical space only.
* The **background of the main page dims** while the menu is open, helping users focus on the navigation options.
* By default, the menu remains **collapsed**, and only opens when the user taps the menu icon. This saves valuable screen space on smaller displays.

Promotional Video – North Story:

<https://drive.google.com/file/d/1f14awqhjJyY309mNBCjXjFDaEUv-Ow97/view?usp=sharing>

חלוקת ניקוד:

