Project Documentation

Overview

Project Name	Take-Home Engineering Challenge: Instagram-style Photo Feed App
Project Dates	Start Date: Aug 25, 2025 End Date: Sep 3, 2025
Background	Built a minimal **Instagram-style photo feed application** where users can view, upload, and interact with posts. This project demonstrates my ability to design a complete system using modern web/cloud/dev tools.
Objectives	• System Architecture • Frontend - Feed UI • Backend - Post Management API • Bonus - Comments, User Profile Page, Activities Page, Explore Page, LLM Caption Generation, Deployment

System Architecture

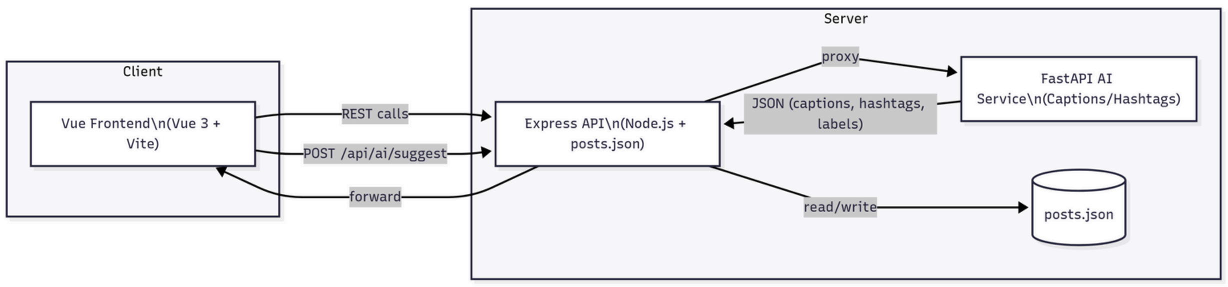
Components

Frontend	• Pages: Auth, Feed, Profile, Explore, Activity, CreatePost. • State: lightweight session via localStorage (fg.token, fg.user). • Config: VITE_API_BASE/VITE_API_URL → base URL for the API. • Calls REST endpoints for posts, users, likes/saves, comments, and activity. AI captioning is invoked from CreatePost.
Backend	• Routes under /api/* with CORS + JSON. • Data store: a single JSON file (posts.json) read/written on each request (simulates a DB). • Responsibilities: ○ Auth: /api/auth/login, /api/auth/signup, /api/auth/me (demo token; password hashing for new signups). ○ Posts: CRUD-ish (GET/POST/DELETE /api/posts, GET /api/posts/:id). ○ Interactions: PATCH /api/posts/:id/like, PATCH /api/posts/:id/save, POST /api/posts/:id/comments, GET /api/posts/:id/comments, POST /api/posts/:id/share. ○ Users: GET /api/users/:username, PATCH /api/users/:username (avatar), POST /api/users/:username/follow. ○ Collections: GET /api/users/:username/{posts|liked|saved}. ○ Activity: GET /api/activity/:username (or ?username=) derived from per-post event logs. ○ AI proxy: POST /api/ai/suggest → forwards to the AI service.
AI Service (FastAPI)	• Endpoints: ○ /api/ai/analyze: detect objects/scenes (CLIP/HF with fallbacks). ○ /api/ai/suggest: return captions, hashtags, labels (with robust fallbacks; optional CLIP re-rank + OpenRouter LLM). • Frontend can call it via the Node proxy; the service is separately deployable.
Media Storage (Simulation)	• No binary uploads. Images are referenced by external URLs and stored inline on the post as imageUrl. • Persistent data lives in posts.json (users, posts, likes/saves/comments, and event logs).

System Architecture

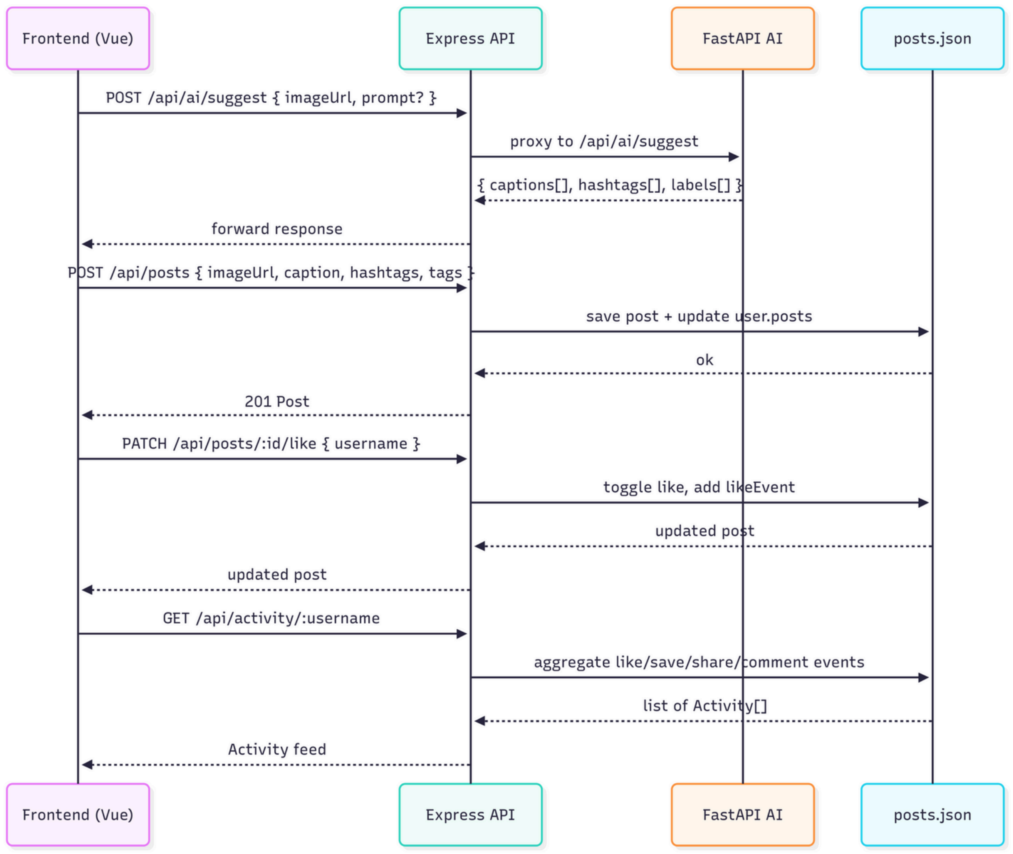
Architecture Diagram

This diagram shows the overall system design in which a Vue 3 frontend communicates with an Express API that stores users and posts in a JSON file (simulating a database). The API also proxies requests to a separate FastAPI AI service, which analyzes images and suggests captions, hashtags, and labels. Media storage is simulated by referencing external image URLs instead of file uploads.


Server
Client
Vue Frontend\n(Vue 3 + Vite)
REST calls
POST /api/ai/suggest
forward
Express API\n(Node.js + posts.json)
proxy
JSON (captions, hashtags, labels)
FastAPI AI Service\n(Captions/Hashtags)
read/write
posts.json

Interaction Diagram

This diagram illustrates a typical user flow in which the frontend requests AI suggestions for captions/hashtags, then creates a new post through the API, which persists data to the JSON store. Users can like, save, comment, and share posts — these interactions are stored as events, which the Activity feed later aggregates and serves back to the frontend.


Frontend (Vue)
Express API
FastAPI AI
posts.json
POST /api/ai/suggest { imageUrl, prompt? }
proxy to /api/ai/suggest
{ captions[], hashtags[], labels[] }
forward response
POST /api/posts { imageUrl, caption, hashtags, tags }
save post + update user.posts
ok
201 Post
PATCH /api/posts/:id/like { username }
toggle like, add likeEvent
updated post
updated post
GET /api/activity/:username
aggregate like/save/share/comment events
list of Activity[]
Activity feed
Frontend (Vue)
Express API
FastAPI AI
posts.json

Typical Flow

Auth.vue	The Auth page handles both login and signup in a single form. 1. Login: sends POST /api/auth/login with username and password (demo users use password demo). 2. Signup: sends POST /api/auth/signup with username, email, password, and optional name fields. 3. On success, the API responds with a { token, user } object. The frontend stores the token and minimal user data in localStorage, then calls /api/users/:username to hydrate the full profile. 4. Finally, the user is redirected to their profile page (/u/:username). 5. Errors (e.g., invalid credentials or duplicate usernames) are displayed inline in the form.
Profile.vue	The Profile page handles the display of user's posts, likes and saves as well as displays relevant details of other users. ● Data loading: ○ Fetches /api/users/:username for profile data. ○ Loads collections in parallel: ▪ GET /api/users/:username/posts ▪ GET /api/users/:username/liked ▪ GET /api/users/:username/saved ○ Lists are sorted by createdAt desc; skeletons show while loading; inline error states on failure. ● Ownership-aware UI: ○ Detects if you're viewing your own profile via the mini session (localStorage.fg.user). ○ Own profile: shows Change avatar action. ○ Other profiles: shows Follow / Unfollow and a placeholder Message button. ● Follow/Unfollow: ○ POST /api/users/:username/follow with { username: <me> }. ○ Optimistically updates local followers and your following list; server response is the source of truth. ● Avatar change: ○ Modal accepts a direct image URL; validates locally. ○ PATCH /api/users/:username with x-user header and { avatarUrl }. ○ On success, updates page state and mini session avatar. ● Header & stats: ○ Derived display name (first/last fallback to @username). ○ Followers, Following, and Total Likes Received (sum of likes.length across your posts). ○ "Joined" date formatted from dateRegistered. ● Tabs & routing:

	○ Three tabs with animated underline; query param ?tab=posts|liked|saved is kept in sync. ○ Each tab displays a uniform post grid with image, like count, date, and one-line caption. ● UX polish: ○ Empty states per tab, subtle animated backgrounds, and accessible button/label text. ○ Route watcher reloads data when navigating to a different
Feed.vue	The Feed page handles the display of posts by people followed by the user. ● Loads & scoping: ○ Fetches GET /api/posts, normalizes arrays, warms author profiles via GET /api/users/:username. ○ If logged in, builds a following set (me + following) and shows only those posts; guests see all. ● Card layout: ○ Header with author avatar/name/time; large photo; footer with actions, caption, and comment UI. ○ "Find similar" jumps to Explore with ?similar=<postId>. ● Interactions (optimistic): ○ Like: PATCH /api/posts/:id/like (toggle, merge server result). ○ Save: PATCH /api/posts/:id/save (toggle, merge server result). ○ Comment: inline composer POST /api/posts/:id/comments; count reflects local list. ● Comments sheet: ○ Opens modal → GET /api/posts/:id/comments; submit via same POST; warms commenter avatars. ● Share sheet: ○ Pick followers (cached) → POST /api/posts/:id/share with { recipients, method:'dm' }. ○ "System Share" uses Web Share API (or copy link) and records method:'native'. ● Options sheet: ○ Copy link, go to post, placeholders for mute/report. ● UX & resilience: ○ Skeletons while loading, empty state when no posts, guarded routes redirect to /auth for actions. ○ Relative time (s/m/h, else date); graceful fallbacks for missing avatars/usernames.
Explore.vue	The Explore page is for discovery outside your graph + quick account search; optional "similar posts" mode. ● Core data:

	◦ Loads GET /api/posts; derives base grid of posts not by you or people you follow. ◦ Warms author profiles; attempts GET /api/users (else derives user list from posts + social graph). • Search accounts: ◦ Focused/searching state shows a users list; filters by username or first+last (client-side). ◦ Each result links to the user profile. • Similar filter: ◦ If routed with ?similar=<postId>: fetches that post (or from cache), extracts tags, shows a banner, and filters grid to posts sharing those tags (excluding followed + self). Clear via × Clear. • Grid behavior: ◦ Non-searching view shows responsive image grid of candidate posts, newest first.
CreatePost.vue	The CreatePost page composes a post from a direct imageUrl, optional "vibe" prompt, caption, and hashtags • AI assist ◦ "Suggest" calls POST /api/ai/suggest with { imageUrl, prompt } → fills caption, hashtags[], and analysis.labels (objects/scenes) used as clickable chips. • Validation & preview ◦ live image preview; bad URL shows inline error; caption limited to 2200 chars. • Tagging model ◦ user-facing hashtags stay as #...; derived tags are kebab-cased labels saved for search/explore. • Submit ◦ POST /api/posts with { username, imageUrl, caption, hashtags[], tags[] }; username from session (fallback demo user if absent). • UX ◦ subtle progress states (AI "Generating...", form "Posting..."), lightweight styling, and return to the new post on success.
PostDetail.vue	The PostDetail page displays a post from the user or other users • Load ◦ GET /api/posts/:id (coerces arrays), warms author + liker avatars, fetches full comments list for preview & modal. • Card ◦ author header, hero image with "Find similar" → /explore?similar=:id, action row, stats (likes/comments/shares/saves), caption, last 4 comments, composer.

	● Optimistic interactions: ○ Like: PATCH /api/posts/:id/like (toggle, reconcile server list). ○ Save: PATCH /api/posts/:id/save. ○ Comment: inline & modal POST /api/posts/:id/comments; preview updates immediately. ○ Share: Sheet to pick followers (cached) → POST /api/posts/:id/share with { recipients, method:'dm' }; native share records { method:'native' }. ○ Delete (author only): DELETE /api/posts/:id then redirect to your profile. ● Sheets ○ comments, share, options (copy link, view profile, delete/report/mute placeholder). ● Time & counts ○ compact relative times; derived counts from arrays; faces from cached profiles.
Activity.vue	The Activity page shows the user all activities that happens to their post(s). ● Primary source ○ GET /api/activity/:username (or GET /api/activity?username=) → normalized to { id,type,actor,postId,date,meta }, newest first. ● Fallback builder ○ if no API, derives notifications from your posts (/api/posts): scans likes[], comments[], saves[], and optional shares[]; follows inferred from /api/users when available. ● UI ○ filter chips (All/Likes/Comments/Shares/Saves/Follows) with counts; each row shows actor, action, relative time, and optional post thumbnail link. ● Data warming ○ caches actor profiles for avatars/display names; also loads your own profile and posts for context. ● States ○ guarded redirect to /auth if not logged in; skeleton loaders and an empty state when nothing's new.

Real World App Improvements

If this were a real-world app, I'd add secure login with email or social accounts, password reset, and stronger session handling. Photos would be uploaded to cloud storage with a CDN for fast delivery and automatic resizing. Instead of a JSON file, posts and users would live in a real database, with search powered by tags and AI for "similar" content. Notifications and AI features would run in the background so the app stays fast. I'd add live updates for likes and comments, tools to report or block content, and basic moderation. For stability, I'd include logging, monitoring, testing, and easy deployment. Besides that, I will improve the AI functionality as it is currently very basic due to memory constraints when using Render's free-tier. Lastly, I'd make sure the app is accessible, works in different languages, respects privacy rules, and stays responsive even with many users.