COLLEGE CODE : 9111

COLLEGE NAME : SRM Madurai College for Engineering and Technology

DEPARTMENT : Computer Science and Engineering

STUDENT NM-ID : 4BF0134536EB1DB9BD0436F56619B6E2

ROLL NO : 911123104010

DATE : 15/09/2025

Completed the project named as Phase 2

TECHNOLOGY PROJECT NAME : IBM-FE-Blog Site with Comment Section

SUBMITTED BY,

NAME : GOKULARAM M

MOBILE NO: 8870213343

Blog Site — Phase 2 Solution Design & Architecture

Scope: Blog site with comments, likes, basic moderation, notifications, and media attachments. This document is a Phase-2 actionable architecture & solution design covering: 1) Tech stack, 2) UI structure & API schema, 3) Data handling, 4) Component / modular diagram, 5) Basic flow diagrams.

Tech Stack (recommended)

Frontend

- React (Typescript) + React Router — component-driven UI, good ecosystem.
- State: React Query (server state) + Context/Redux for auth & UI state.- Styling: TailwindCSS (rapid, consistent design)
- Build & bundling: Vite

Backend

- Node.js + TypeScript (Express or Fastify) — clear typing & ecosystem.
- GraphQL (Apollo) *or* REST (OpenAPI) — pick one. For clarity this doc shows REST endpoints.

Database & Storage

- Primary DB: PostgreSQL (relational; transactions for posts/comments). Use UUID primary keys.
- Search: ElasticSearch / OpenSearch for full-text search (titles, bodies, tags).
- Attachments: S3-compatible object storage (AWS S3, DigitalOcean Spaces). □ Cache: Redis (session cache, rate-limiting, frequently requested data)

Auth & Identity

- JWT access tokens + refresh tokens (HTTP-only secure cookies) or OAuth 2.0 for social logins.
- Optionally 3rd-party auth provider: Auth0 / Clerk if you want managed flows.

Other infra

- API Gateway / Reverse Proxy: Nginx or Cloud Load Balancer.
- CDN: CloudFront / Fastly for static assets + images.
- CI/CD: GitHub Actions (build/test/deploy).
- Containerization: Docker + Kubernetes (or managed containers like ECS, DigitalOcean Apps).
- Observability: Prometheus + Grafana, Loki for logs, Sentry for errors.

Security & Compliance

- HTTPS everywhere (TLS), CSP headers, input sanitization (XSS protection), prepared statements (SQL injection protection), CSRF tokens for non-REST-safe flows.

UI Structure & API Schema Design

UI page / component structure (top-level)

- **Public** ○ HomeFeed (list of posts, pagination, filters) ○ PostView (single post, comments, comment input) ○ SearchResults ○ AuthorProfile
    - SignIn / SignUp / ForgotPassword
- **Authenticated** ○ CreatePost / EditPost (rich text or markdown editor + attachments) ○ MyDrafts ○ Notifications ○ AccountSettings
- **Shared components** ○ Header / Nav ○ PostCard
    - CommentList / CommentItem
    - Modal / Toasts ○ Avatar, RichTextRenderer

API design (REST, resource-driven) — base: `/api/v1`

Authentication

POST /api/v1/auth/register

- Body: `{ name, email, password }`
- Response: `201 { user, accessToken, refreshToken }`

POST /api/v1/auth/login

- Body: `{ email, password }`
- Response: `200 { user, accessToken, refreshToken }`

POST /api/v1/auth/refresh

- Body: `{ refreshToken }` -> rotate token

POST /api/v1/auth/logout

- Invalidate refresh token cookie/server side

Posts

GET /api/v1/posts?cursor=&limit=&q=&tag=&author=&sort=

- Returns paginated list (cursor-based) of posts with summary fields.

GET /api/v1/posts/{postId}

- Returns full post object (includes comment count, author info).

POST /api/v1/posts (auth)

- Body: `{ title, body, tags[], status: draft|published, attachments[] }` - Response: `201 { post }`

PATCH /api/v1/posts/{postId} (auth & owner)

DELETE /api/v1/posts/{postId} (auth & owner)

Comments

GET /api/v1/posts/{postId}/comments?cursor=&limit=

- Returns comments (cursor-based) ordered by createdAt or ranking (see data handling).

POST /api/v1/posts/{postId}/comments (auth)

- Body: `{ parentCommentId?, content }`
- Response: `201 { comment }`

PATCH /api/v1/comments/{commentId} (owner)

DELETE /api/v1/comments/{commentId} (owner or moderator) — marks as deleted or hard delete depending on retention policy

POST /api/v1/comments/{commentId}/report (auth)

- Body: `{ reason }`

Reactions & Moderation

POST /api/v1/posts/{postId}/like (auth) **POST /api/v1/comments/{commentId}/like** (auth)

GET /api/v1/moderation/reports?status=open|resolved (moderator)
POST /api/v1/moderation/actions (moderator action: warn, hide, ban)
Notifications

GET /api/v1/notifications?limit=&cursor= POST /api/v1/notifications/mark-read

Example JSON schema — `Post` and `Comment`

```
Post {   id: "uuid",
title: "string",   slug:
"string",
  body: "string (markdown/html)",
excerpt: "string",   authorId: "uuid",
tags: ["string"],
  attachments: [{url, type, size}],   status:
"draft|published|archived",   createdAt: "iso",
updatedAt: "iso",   commentsCount: 12,
reactions: { like: 42 }
}

Comment {   id: "uuid",   postId:
"uuid",   parentId: "uuid|null",
authorId: "uuid",   content:
"string",   isDeleted: false,
createdAt: "iso",   updatedAt:
"iso",   reactions: { like: 3 }
```

```
}
```

Data handling

Data model (relational tables - simplified)

- `users` (id UUID, name, email, password_hash, role, created_at)
- `posts` (id, author_id FK users, title, body, slug, status, excerpt, created_at, updated_at) `tags` (id, name)
- `post_tags` (post_id, tag_id)
- `comments` (id, post_id, parent_id, author_id, content, is_deleted, created_at)
- `reactions` (id, entity_type, entity_id, user_id, type)
- `reports` (id, entity_type, entity_id, user_id, reason, status)
- `attachments` (id, owner_type, owner_id, url, meta)
- `notifications` (id, user_id, type, payload, is_read)

Pagination & ordering

- Use **cursor-based pagination** for posts/comments to avoid offset problems on large data sets. Cursor = createdAt+id or a dedicated opaque cursor token.
- For comment threads, load top-level comments paginated; load nested replies lazily (on expand).

Search

- Index posts (title, body, tags, author) to ES/OpenSearch. Keep a sync job or use a change-data-capture (logical replication) pipeline.

Concurrency & consistency

- Use DB transactions for operations that update multiple tables (create post + tags + attachments).
- For comment counts / reaction counts: maintain counters in DB and cache in Redis. Use incremental updates in transaction.

Moderation & soft deletes

- Use `is_deleted` flag on comments and `status`/`deleted_at` on posts for retention and auditability.
- Keep reports in a separate table with a review workflow. Moderation actions are recorded as immutable audit log entries.

Rate limiting & abuse prevention

- Rate limit write actions per IP / per user using Redis token buckets (create comment, create post).
- Profanity filter & basic NLP-based toxic content scoring (Perspective API or opensource models) to flag high-risk comments before inserting.

Backups & retention

- Automated daily logical backups of Postgres + periodic snapshots; S3 for attachments with lifecycle policies.

Component / Modular Diagram

Modules

- **Client**: React app (SPA) — handles UI, validation, optimistic updates.
- **API Layer**: Gateway + Backend services (Auth Service, Blog Service, Comment Service, Notification Service).
- **Data Layer**: Postgres, ElasticSearch, Redis, S3.
- **Admin/Moderation**: Dashboard for moderator actions and report triage.
- **Worker Queue**: Background workers (Bull/Sidekiq style) for tasks: send emails, rebuild search index, process images, send push notifications.

Mermaid component diagram (paste into any mermaid renderer):

```
flowchart TB   subgraph Client
A[Browser: React SPA]   end
  subgraph API
    B[API Gateway / Load Balancer]
    C[Auth Service]
    D[Blog Service]
    E[Comment Service]
    F[Notification Service]   end
subgraph Data     PG[(Postgres)]
    ES[(ElasticSearch)]
    RED[(Redis)]     S3[(S3
Storage)]   end    subgraph Workers
W[Background Workers]   end
A --> B
B --> C
  B --> D
  B --> E
  B --> F
  D --> PG
  D --> ES
D --> S3
E --> PG
E --> RED   F --> PG
F --> RED
  W --> ES
  W --> S3
        W --> PG
```

Basic flow diagrams

Flow: Create post with attachments

```
sequenceDiagram   participant User
participant Frontend   participant
API   participant Worker
participant S3   participant DB

  User->>Frontend: Fill post + upload images
  Frontend->>S3: Direct signed upload (pre-signed URLs)
  S3-->>Frontend: Upload complete URLs
  Frontend->>API: POST /posts {title, body, attachments: [urls]}
  API->>DB: Begin transaction: insert post, tags, attachments
  DB-->>API: 201 created
```

```
  API->>Worker: enqueue `index-post` job
  Worker->>ES: index post for search
  API-->>Frontend: 201 {post}
  Frontend-->>User: success
```

Flow: Add comment (with moderation filter)

```
sequenceDiagram   participant User
participant Frontend   participant
API   participant DB   participant
Redis   participant Worker

  User->>Frontend: Submit comment
  Frontend->>API: POST /posts/{id}/comments {content}
  API->>Redis: check rate-limit
  Redis-->>API: allowed
  API->>Worker: async `score-comment` (toxicity) // non-blocking
  API->>DB: insert comment (is_flagged=false)
  DB-->>API: 201 {comment}   API-->>Frontend: 201
{comment}

  Worker->>ExternalService: call toxicity model
  ExternalService-->>Worker: score   alt score >
threshold
           Worker->>DB: mark comment as is_flagged=true     Worker->>NotificationService: notify
moderators   end
```

Operational considerations & non-functional requirements

- **Scalability:** horizontal scale for API; split read/write DB if necessary. Use read replicas for analytics.
- **Availability:** design with health checks, multi-AZ DB, and graceful degradation for search (fallback to DB queries).
- **Performance:** caching for popular posts & author profiles; use CDNs for images.
- **Observability:** tracing (OpenTelemetry), metrics on request latency, error rates, queue length.
- **Privacy:** PII handling, GDPR considerations for user deletion (right to be forgotten). **Cost control:** lifecycle rules for images, pay-as-you-go managed services.