Phase 2 — Solution Design & Architecture

Project:

Contents

---

1. Overview

This document describes the solution design and architecture for a **Blog site with a comment section**. It covers the chosen technology stack, frontend UI structure, backend API schema, data handling, component/module decomposition, and basic flows required to implement a production-ready MVP that is secure, maintainable, and scalable.

The goal is to produce a clear blueprint that developers can implement within Week 7.

---

2. Goals & Non-Functional Requirements

Primary goals - Allow authenticated users to create/read/update/delete blog posts. - Allow users (authenticated and optionally guests) to add comments to posts. - Provide efficient listing and search of posts. - Maintain data integrity and protect against common web vulnerabilities.

Non-Functional Requirements - **Performance:** Page load < 1.5s for homepage (cache-friendly). API response times < 300ms for typical requests. - **Scalability:** Support horizontal scaling of stateless backend and read replicas for DB. - **Availability:** 99.9% uptime target for core read paths. - **Security:** OWASP Top 10 protections, HTTPS only, secure auth tokens. - **Maintainability:** Clean module separation, documented APIs, automated tests. - **Extensibility:** Easy addition of features (likes, tagging, social share).

---

3. Tech Stack Selection

Frontend - Framework: **React.js** (create-react-app or Vite) - Styling: **Tailwind CSS** (utility-first, fast prototyping) or **Bootstrap** if preferred - State management: **Redux Toolkit** or React Context for light apps - HTTP: **Axios** for API calls - Auth UI: JWT stored in HttpOnly cookie or secure storage

Backend - Framework: **Node.js + Express.js** (fast, large ecosystem) - Language: JavaScript / TypeScript (TypeScript recommended for type safety) - ORMs: **Sequelize** (Postgres/MySQL) or **TypeORM** if TypeScript

Database - Primary: **PostgreSQL** (relational, strong consistency for comments & posts) - Optional: **Redis** for caching, rate-limiting counters, and session caching

Storage - Media (images): **AWS S3 / DigitalOcean Spaces**

Auth & Identity - JWT tokens, refresh tokens (or OAuth 2.0 if third-party auth required)

Hosting / DevOps - Frontend: **Vercel / Netlify** - Backend: **AWS Elastic Beanstalk / EC2 / Heroku** - DB: **AWS RDS (Postgres)** or **PlanetScale / Supabase** - CI/CD: **GitHub Actions** - Monitoring: **Sentry** (errors), **Prometheus + Grafana** (metrics)

Optional - Search: **Elasticsearch** or Postgres full-text for post search - Message queue: **RabbitMQ / AWS SQS** for async tasks (email, notifications)

Rationale: Postgres ensures strong relational integrity for posts, comments, users. Node + Express is widely used and pairs well with React on frontend.

---

4. UI Structure & Page Wireframes

Primary pages / components - **Header/Nav:** Logo, Search, Login/Register, Create Post (if authorized) - **Homepage (/):** List of posts (cards), filter and sort controls, pagination - **Post Detail (/posts/:id):** Title, metadata (author, date), content (HTML/Markdown), comment list, add comment form - **Author Profile (/users/:id):** User bio, posts list - **Login / Register:** Forms with validation - **Create / Edit Post (/posts/new, /posts/:id/edit):** Rich text editor or Markdown editor - **Admin Dashboard (optional):** Manage posts, comments, users

Wireframe (textual) - Homepage: Top nav → search bar → featured post carousel → list of post cards (title, excerpt, tags, author, comment count) → pagination - Post Detail: Post header → content → tags → share buttons → comments section (top) → comment input → related posts sidebar

---

5. API Schema Design (REST)

This section gives endpoint list, parameters, example request/response.

Authentication & Users

- `POST /api/auth/register`
- Body: `{ "username": "pavi", "email": "p@example.com", "password": "Secret123" }`
- Response: `{ "id": 1, "username": "pavi", "token": "<jwt>" }`
- `POST /api/auth/login`
- Body: `{ "email": "p@example.com", "password": "Secret123" }`
- Response: `{ "token": "<jwt>", "refreshToken": "<token>", "user": { ... } }`
- `GET /api/users/:id` — Get profile

Posts

- `GET /api/posts?page=1&limit=10&search=react&tag=web` — List posts (paged)
- Response contains `meta: { page, limit, total }` and `data: [posts]`
- `GET /api/posts/:id` — Get single post with comments summary
- `POST /api/posts` — Create post (auth)
- Body: `{ "title": "My Post", "content": "...", "tags": ["react","web"] }`
- `PUT /api/posts/:id` — Update
- `DELETE /api/posts/:id` — Delete

Comments

- `GET /api/posts/:id/comments?page=1&limit=20` — Get comments for post
- `POST /api/posts/:id/comments` — Add comment (auth or guest depending on rules)
- Body: `{ "content": "Nice post!", "parentId": null }` — `parentId` for threaded replies
- `DELETE /api/comments/:id` — Delete comment (moderator or owner)

Admin / Moderation

- `GET /api/moderation/comments?status=pending` — For admins
- `PUT /api/comments/:id/approve` — Approve comment

Notes on responses - Always return consistent envelope: `{ success: true, data: ..., meta: ... }` or standardized error object `{ success: false, error: { code, message, details? } }`.

---

6. Data Model & Data Handling Approach

Core tables (Postgres)

users - id (PK), username, email (unique), password_hash, bio, role (user/admin), created_at, updated_at

posts - id (PK), author_id (FK users.id), title, slug (unique), content (text/markdown), html_content (optional), excerpt, status (published/draft), created_at, updated_at

comments - id (PK), post_id (FK posts.id), user_id (FK users.id, nullable for guest), content, parent_id (self FK for nested comments), status (approved/pending/hidden), created_at, updated_at

tags - id, name

post_tags - post_id, tag_id (many-to-many)

likes (optional) - id, post_id, user_id

Data handling patterns

- **Input validation:** Use middleware to validate bodies (Joi or express-validator)
- **Transactions:** Use DB transactions for multi-step operations (e.g., create post + tags linking)
- **Pagination & Limits:** Cursor or offset pagination. Default page size 10, max 50
- **Caching:** Cache popular posts and expensive queries in Redis. Use HTTP caching headers for static assets.
- **Sanitization:** Store raw markdown but sanitize HTML before sending to clients (e.g., DOMPurify at server or client side when rendering HTML)
- **Search:** Postgres full-text search for title/content or integrate Elasticsearch for complex queries

---

7. Component / Module Diagram

Frontend components (React)

- `App` (router)
- `Header` / `Footer`
- `HomePage` -> `PostList` -> `PostCard`
- `PostPage` -> `PostContent`, `CommentList`, `CommentForm`
- `Auth` -> `LoginForm`, `RegisterForm`
- `Editor` -> `MarkdownEditor` / `WYSIWYG`
- `Profile` -> `UserPosts`

Backend modules (Express)

- `AuthController` (register, login, refresh token)

- `UserController` (profile, settings)
- `PostController` (list, get, create, update, delete)
- `CommentController` (list, add, delete, moderate)
- `MediaController` (upload images)
- `ModerationService` (flagging, approval flows)
- `NotificationService` (email, in-app notifications)

Simple ASCII diagram (component interactions)

```
[Browser/Client]
   |  <--Axios/Fetch-->
[API Gateway / Backend (Express)]
   |--> Auth Service (JWT)
   |--> Post Service (CRUD + search)
   |--> Comment Service (CRUD + moderation)
   |--> Media Service (S3 uploads)
   |--> Cache (Redis)
   |--> Database (Postgres)
```

---

8. Basic Flow Diagrams

Flow 1 — Viewing a post

1. User requests `/posts/:id` from browser.
2. Frontend calls `GET /api/posts/:id`.
3. Backend checks cache (Redis). If cache hit, return cached payload. Else query Postgres for post + top N comments.
4. Backend sanitizes content HTML and returns response.
5. Frontend renders post and comment list.

Flow 2 — Adding a comment (authenticated)

1. User fills comment form and submits.
2. Frontend POSTs to `/api/posts/:id/comments` with JWT in HttpOnly cookie.
3. Backend authenticates token, validates body, creates comment row (status=approved or pending based on moderation rules).
4. If immediate display allowed, return new comment; frontend appends comment to list. Optionally broadcast via WebSocket.
5. For moderation, the comment is queued for review and shown only after approval.

Sequence (text)

```
User -> Frontend: submit comment
Frontend -> Backend: POST /api/posts/:id/comments (JWT)
```

```
Backend -> DB: insert comment
DB -> Backend: confirm insert
Backend -> Frontend: new comment (or pending)
Frontend -> UI: render comment
```

---

9. Security, Performance & Scaling

Security - Use HTTPS everywhere, HSTS header. - Store passwords with bcrypt (salted hashes). - Use HttpOnly + Secure cookies for JWT or short-lived access tokens + refresh tokens. - CSRF protection (double submit cookie or CSRF tokens). - Input sanitization for comments and posts (prevent XSS). Use whitelisting libraries. - Rate limiting per IP or per user (e.g., 10 comments / minute). - Content moderation: profanity filter, spam detection (Akismet or custom heuristics), user flagging.

Performance - CDN for static assets and images (S3 + CloudFront). - Redis for caching frequently-read posts. - Database indexing: index `posts.slug`, `posts.created_at`, `comments.post_id`, full-text indexes for search.

Scaling - Design backend as stateless; scale horizontally behind a load balancer. - Use read replicas for DB read-heavy operations. - Use queue system for background tasks (sending emails, notifications).

---

10. Testing Strategy

Unit tests - Controllers, services, and utility functions. - Use Jest (Node) / React Testing Library (frontend)

Integration tests - End-to-end API tests using Supertest or Postman collections. - Test auth flows, post/comment CRUD operations, permission checks.

E2E tests - Cypress for full UI flows (create post, comment, login)

Load testing - Use k6 or JMeter to simulate traffic spikes for the homepage and comments endpoints.

Security testing - Run automated scanners (OWASP ZAP), dependency vulnerability scans (Snyk/Dependabot).

---

11. CI/CD, Deployment & Monitoring

CI - GitHub Actions pipeline: lint -> unit tests -> build -> integration tests -> deploy to staging

CD - Auto deploy to staging on pull request merge to `develop`. - Manual approval to deploy `main` to production.

Deployment - Containerize backend with Docker, deploy to AWS ECS / EKS or Heroku. - Use DB migrations (Flyway / Sequelize CLI) during deploy.

Monitoring & Logging - Centralized logs (ELK stack or cloud provider logs). - Error tracking with Sentry. - Metrics exported to Prometheus; dashboards in Grafana. - Alerting for error rate, latency, and resource exhaustion.

---

12. Appendix

Sample JSON: Create comment

```
POST /api/posts/42/comments
{
  "content": "Great explanation! Thanks.",
  "parentId": null
}
```

ER diagram (notes)

- users (1) --- (N) posts
- posts (1) --- (N) comments
- posts (M) --- (N) tags (via post_tags)
- users (1) --- (N) comments

Implementation checklist (Milestones)

1. Project scaffolding: React + Express + Postgres skeleton
2. Auth system (register/login + JWT)
3. Post CRUD + DB migrations
4. Comment CRUD + moderation
5. Basic frontend pages: homepage, post page, editor
6. Tests: unit & integration
7. Deployment: staging + production
8. Monitoring + rate-limiting + caching

Final notes

This document is intentionally pragmatic: it lists concrete choices and patterns to implement a reliable blog with comments. If you prefer a different stack (Django, Firebase, or serverless), the same architecture decisions (API surface, data model, caching, security) remain relevant and can be adapted.

Prepared for: Blog site with comment section — Phase 2 (Solution Design & Architecture) *Prepared on:* 2025-09-15