Technical Specification — Community Platform

A lean stack built on AWS (S3 + EC2) and Vercel, with a content management layer so the committee can manage content without coding.

Frontend

- Next.js (latest version) with App Router
- Tailwind CSS for styling
- Fast, SEO-friendly (SSR/SSG), mobile-first, fully responsive
- Dark / light mode
- i18n with next-intl — Bangla default, English toggle
- Deployed on Vercel (global CDN, automatic HTTPS, zero-config Next.js builds)

Backend

- NestJS (Node.js + TypeScript) — modular, structured, testable
- Single REST API as the only access layer (frontend never touches the DB directly)
- Centralises validation, authorisation, and audit logging in one place
- Deployed on AWS EC2

Database

- PostgreSQL
- Hosted on AWS (EC2 self-hosted, or RDS recommended for automated backups and failover)
- Accessed only through the NestJS API via ORM (Prisma or TypeORM) with parameterised queries

Storage

- AWS S3 for all images and media
- Optional CloudFront CDN in front of S3 for faster delivery
- Encrypted at rest

Content Management (CMS — committee manages content without coding)

The committee must be able to edit content without a developer. Two valid options:

- **Option A — Headless CMS (Sanity.io or Contentful):**
  - Committee uses a polished, ready-made editing studio
  - Content fetched into Next.js via API at build/request time

    - Fastest to give a rich editing experience; adds one external vendor and a monthly cost
    - Good fit if the committee wants flexible, magazine-style content editing
- **Option B — Built-in admin dashboard (no external CMS):**
    - Content (pages, announcements, news, events, jobs, donation campaigns) stored in PostgreSQL
    - Committee edits through the same admin dashboard used for members and voting
    - Keeps everything in one system and one login; no extra vendor cost
    - Good fit if content needs are structured and predictable
- **Recommendation:** Sanity.io if the priority is rich, freeform content editing out of the box; the built-in dashboard if the priority is a single unified system. Both fully satisfy "manage content without coding."

Authentication

- JWT-based, issued and verified by the NestJS backend
- Access token (short-lived, ~15 min) + refresh token (longer-lived, ~7 days)
- Refresh token stored in an HttpOnly, Secure, SameSite=Strict cookie
- Refresh tokens rotated on each use and revocable server-side (hashed records in PostgreSQL) — a stolen token can be invalidated
- Login via email + member ID (+ password or one-time code), tied to verified member records
- Optional MFA / one-time email code for committee-admin accounts
- Member identity stays in your own database — no third-party auth vendor, no per-user pricing

Authorization

- Role-based access control (RBAC): roles such as member, editor, committee-admin
- Role claims embedded in the JWT and enforced by NestJS guards on every protected route
- Principle of least privilege — each role gets only the access it needs
- Admin/committee routes (members, content, voting, donations) gated behind admin guards
- Server-side authorization on every request — never rely on the frontend to hide actions
- Audit logging of sensitive admin actions (who changed what, when)

Security

- Passwords hashed with argon2 / bcrypt (never stored in plain text)
- All traffic over HTTPS/TLS; HSTS enabled
- Input validation on every endpoint (NestJS DTOs + class-validator) — blocks injection at the boundary

- Parameterised queries / ORM — eliminates SQL injection
- Output encoding and a Content Security Policy to prevent XSS
- CSRF protection on cookie-based auth flows
- Rate limiting and brute-force protection on auth endpoints
- CORS locked to the frontend domain only
- Security headers via Helmet (CSP, X-Frame-Options, X-Content-Type-Options, etc.)
- Secrets in environment variables / AWS Secrets Manager — never in the repo
- Encryption at rest (database + S3) and in transit
- Automated, encrypted database backups
- Dependency and vulnerability scanning in CI (npm audit / Dependabot)

Internationalisation

- next-intl with Bangla (bn) as default locale, English (en) toggle
- Static UI strings in JSON; committee-editable content from CMS/DB for live edits without redeploy
- Bangla font rendering and fallback tested

Quality Targets

- Lighthouse score 90+
- WCAG 2.1 AA accessibility
- Mobile-first responsive layout

UI / UX Feedback

- **Cultural identity done with restraint:** use Bangladeshi visual cues (colour accents, typography, motifs) as highlights, not as a heavy theme — keeps it modern and avoids clutter
- **Bangla-first typography:** pick a font with strong Bangla glyph support and verify line-height/spacing; Bangla often needs more vertical rhythm than Latin text
- **Language toggle must be obvious and persistent:** clearly visible in the header, remembers the user's choice, and doesn't reset on navigation
- **Mobile-first for real usage:** the diaspora audience is likely majority mobile — design and test on small screens first, large tap targets, thumb-reachable navigation
- **Accessibility is part of the design, not a checkbox:** sufficient colour contrast in both light and dark mode, visible focus states, proper heading hierarchy, alt text on media — aim for WCAG 2.1 AA from the first mockups
- **Consistent design system:** define spacing, colour, and component tokens up front (Tailwind config) so the site feels cohesive and is easy to extend
- **Clear content hierarchy:** announcements/events should be immediately scannable on the homepage; avoid burying key community info

- **Admin/committee UX matters too:** the editing interface should be simple enough for non-technical committee members — clear labels, sensible defaults, confirmation on destructive actions
- **Performance is UX:** optimise images via S3/CloudFront, lazy-load media, and keep the Lighthouse 90+ target visible throughout, not just at launch