Project Overview — Rand Lottery

- Purpose: generate, display, and share lottery results (downloadable images + social share)
- Two-tier app: frontend (UI/UX) and backend (API, DB, social integrations)
- Design emphasis: mobile-first, accessible, and shareable result cards

Repository Layout

- frontend/ — Vite + React + TypeScript + Tailwind; UI components and pages
- backend/ — FastAPI, SQLAlchemy, async repositories, services, and API routes
- public/ or assets/ — images and design assets copied into frontend/src/assets
- backend/docs/ — helper scripts and documentation (slide generator, README)

Frontend Architecture

- Tech: React 18, TypeScript, TailwindCSS, shadcn/ui (Radix), lucide-react icons
- Routing: Vite + React Router (pages under src/pages)
- State/data: React Query for fetching results from backend APIs
- UI patterns: component library under src/components, design-specific components in
- src/components/design9

Frontend Key Files

- src/pages/* — top-level pages (Dashboard, Games, History, PostResults)
- src/components/results/* — result-related UI (cards, share panel, generator)
- src/lib/api.ts — typed API client for backend endpoints
- src/lib/imageGenerator.ts — robust html2canvas capture helper

Backend Architecture

- FastAPI app structured with api/, services/, repositories/, models/
- Async SQLAlchemy session pattern under backend/app/db/session.py
- Pydantic settings for environment variables and secrets
- Service layer handles external integrations (WhatsApp/provider upload + send)

Backend Key Files

- backend/app/api/ — route handlers (social.py, games.py, results.py)
- backend/app/services/ — business logic (social_media.py, games.py, results.py)
- backend/app/models/ — SQLAlchemy models for Game, Draw, Result
- backend/app/repositories/ — DB access patterns encapsulating queries

Data Model (high-level)

- Game: metadata about lottery/game (name, type, assets)
- Draw: scheduled occurrence with draw number and date
- Result: winning numbers, machine numbers, and associations to Draw/Game
- Result attachments: image URLs or media ids for shared assets

APIs & Flows

- GET /games — list available games
- POST /results — submit new results (admin flow)
- GET /results/{id} — retrieve rendered result data
- POST /social/post — accept `image_base64` + `whatsapp_recipient` to send images

Deployment & Dev Ops

- Docker Compose present in `backend/docker-compose.yml` for postgres + app
- Backend Dockerfile and environment variables via `.env`
- Frontend: Vite dev server for local work; build for production artifacts
- Recommendation: host backend on a managed host, use environment secrets store

Security & Secrets

- Do not commit `.env` with tokens; use platform secrets manager
- WhatsApp/Provider tokens required for live sending
- Use HTTPS for all public endpoints and validate incoming webhooks

Testing & QA

- Unit tests for services and repositories (add pytest)
- End-to-end testing: simulate result creation → capture → social POST
- Manual QA: verify captured image alignment and fonts across platforms

Operational Notes

- Rate limits & messaging costs depend on provider (Meta or third-party)
- Monitoring: add logs for message send status and webhook callbacks
- Add retry/backoff for media upload and message send failures

Next Actions

- Decide provider for production (Meta Cloud API vs MessageBird/Infobip)
- Provide credentials for an end-to-end dry-run or grant me permission to run tests
- Add screenshots to slides (I can capture or you can upload) and regenerate PPT/PDF