FULL-STACK • BACKEND • CLOUD DATABASE • SYSTEM DEBUGGING

MEGA888 Casino Application

A deployed, production-style Flutter + NestJS application demonstrating end-to-end ownership across authentication, server-authoritative game/wallet operations, PostgreSQL persistence, cloud migration and evidence-driven debugging.

Flutter / Riverpod	NestJS / TypeScript	Prisma / PostgreSQL	REST / JWT	Render / Neon

Project snapshot

I designed and developed MEGA888 as a full-stack casino-style application rather than a frontend-only prototype. The implementation spans a Flutter mobile client, NestJS REST API, Prisma/PostgreSQL persistence, authenticated user/session flows, backend-controlled game sessions and settlement, wallet transaction recording, live/lottery structures, promotions/rewards/notices, runtime controls, application-version handling, and administrative/audit/fraud data models.

Engineering throughline. The strongest evidence in this project is how I handled boundaries: prove which layer failed, keep financially meaningful state authoritative on the server, give asynchronous authentication one owner, verify infrastructure changes against the target, and explicitly identify where demo functionality stops being production-grade.

Architecture and ownership

```
Flutter / Dart
Riverpod state • GoRouter • session/API layer
        │  HTTPS / JSON / Bearer JWT
        ▼
NestJS / TypeScript
/api • DTO validation • JWT guards • services • Swagger
        │  Prisma ORM / transactions
        ▼
PostgreSQL
identity • games • bets • wallet • content • admin/audit
        │
Render-hosted API  +  Render PostgreSQL → Neon PostgreSQL
```

Area	What I implemented / worked across
Client	Authentication lifecycle, routing, runtime login control, catalog/live/content flows and centralized endpoint contracts.
Backend	NestJS controllers/services, validation, protected resources, game-session composition, wager dispatch and settlement logic.
Data	Relational game/user/financial domains, Prisma transactions, constraints/indexes and migration history.
Cloud / operations	Render deployment context, PostgreSQL backup/restore, Neon migration verification and deployed API troubleshooting.

Authentication concurrency bug

What the project evidence shows. The earlier client architecture allowed authentication refresh responsibility to exist in more than one place. The corrected API client explicitly states that token refresh is now owned by dio_provider.dart only, uses a single refresh lock, and reads the latest token from storage for each request to prevent stale-token retries.

User-visible failure mode. The live-table UI explicitly handles HTTP 401 by presenting: “Your session may have expired. Please go back and log in again.” This establishes the concrete client-visible symptom associated with an unrecovered authentication failure.

Root risk. With single-use/rotating refresh credentials, two independent refresh owners can race: one consumes and rotates the refresh token while the other attempts to use the now-stale credential. The same architecture also risks retrying requests with an outdated access token.

```
Before
AuthController ──────┐
                     ├── backend token refresh
Network layer ───────┘
        ↓ competing ownership / stale-token risk

After
AuthController → restore local session only
Dio interceptor → sole backend refresh owner
                → single refresh lock
                → attach latest stored token per request
                → refresh on protected-request 401
```

Fix. I removed backend refresh ownership from the AuthController and centralized it in the Dio network layer. The AuthController restores persisted session state; the interceptor owns refresh and retry behavior. This converts an ambiguous asynchronous lifecycle into a single-owner state machine.

Server-authoritative betting and wallet integrity

The client proposes an action; the backend decides whether it is valid. CatalogService loads the authoritative Game and User, rejects missing/inactive games, applies stake/limit/balance checks, selects the correct settlement path, and returns the server result.

```
Flutter wager
  → JWT-authenticated API request
  → load authoritative Game + User
  → validate active state / stake / limits / balance
  → dispatch live / mini-game / slot / generic path
  → persist related state
  → return settlement reference + updated balance
```

In the evidenced slot path, the balance mutation and WalletTransaction record execute within the same Prisma transaction. The transaction record includes source/status, net amount, balance-before, balance-after, reference ID and settlement context.

Trust boundary. The Flutter client does not authoritatively determine balance, game activation, betting limits or settlement. Financially meaningful validation and persistence remain server-side.

API evolution without breaking deployed clients

The catalog controller retains both the current Flutter route shape and older aliases for game sessions, bets and spins. This avoids turning an API cleanup into a client-breaking deployment.

Current	Compatibility alias
/api/catalog/:gameId/session	/api/catalog/games/:gameId/session
/api/catalog/:gameId/bets	/api/catalog/games/:gameId/bets
/api/catalog/:gameId/spin	/api/catalog/games/:gameId/spin

Breadth of the persisted system

Domain	Representative persisted entities
Identity / recovery	User, Session, VerificationCode, PasswordResetSession
Casino / live	GameCategory, Game, GameConfig, GameRound, LiveTable, LiveBet, LiveEvent, BetSettlement

Lottery	LotteryBet, LotteryResult
Financial	WalletTransaction, DemoPaymentInvoice
Engagement	Notice, Promotion, Reward, ReferralReward, InboxMessage
Operations	RuntimeConfig, AppVersion, EmailLog, SmsLog, FraudEvent
Administration	AdminRoleKey, AdminApprovalRequest, AdminAuditLog, AdminNote
Schema history	_prisma_migrations

This schema breadth is useful evidence that the project extends beyond gameplay screens into identity, operational controls, financial records, content, administration and auditability.

Database migration

I migrated the application's PostgreSQL database from Render PostgreSQL to Neon while preserving application data, keys, indexes, foreign-key relationships and Prisma migration history. I did not treat a restore command returning to the prompt as proof of success.

Failure and correction. An initial restore through the Neon pooler produced little useful output and the target still lacked application tables. I validated the custom archive with pg_restore -l, switched to Neon's direct endpoint, restored with verbose/no-owner/no-ACL options, then inspected the target again.

Verification. The restored target contained the application tables and _prisma_migrations; Prisma Client generation and migration-status checks succeeded, followed by successful NestJS development startup.

Dedicated evidence record. The separate MEGA888 Render→Neon Database Migration Technical Record contains the full command sequence, 192-entry archive inspection, troubleshooting matrix and cutover procedure. This record intentionally avoids duplicating that document.

Layered debugging: the 404 that was not a database failure

A deployed request to /api/admin/me returned NestJS HTTP 404. Rather than interpreting any failed API request as a database problem, I used the response itself as evidence: DNS/TLS/network connectivity and the Render service were reachable, and NestJS processed the request. The next investigation layer was therefore route registration, HTTP method, global prefix/controller path or deployed build — not PostgreSQL.

```
Client command → DNS/TLS → Render service → NestJS routing
                                          ↑
                                  404 proved reachability

Database connectivity is downstream and was not implicated by that 404.
```

A separate PowerShell issue reinforced the same approach: PowerShell's curl alias can behave as Invoke-WebRequest, so I used curl.exe / native PowerShell header syntax to separate a local shell problem from the actual HTTP response.

Administrative authentication

Current implementation. The project includes an x-admin-key header mechanism using a static shared administrative secret. I would not describe that as production-grade administrator identity.

Why it is weak. A shared static secret is not inherently attributable to one administrator, provides weak role granularity, and is harder to revoke/rotate safely than individually authenticated admin sessions.

Production direction. Use individual administrator identities, MFA, role/permission-based authorization, short-lived sessions, privileged-action approval where appropriate and complete audit attribution. Existing AdminRoleKey, AdminApprovalRequest, AdminAuditLog and AdminNote structures provide foundations for this direction, but do not by themselves prove the complete control plane is implemented.

FEDI'S COPY

Production boundary

The project contains demo randomized game behavior. The evidenced slot logic uses Math.random(). I would not represent that as regulated real-money casino RNG.

Current project boundary	Production real-money direction
Demo randomization	Certified/provider RNG or cryptographically appropriate audited implementation; independently validated game mathematics.
Transactional wallet write	Idempotent settlement, stronger immutable/accounting-grade ledger and reconciliation.
Atomic writes	Concurrency protection against simultaneous overspend and duplicate settlement.
Static admin key	Individual admin identity, MFA, RBAC/permissions and auditable privileged actions.
Application API	Rate limiting, abuse/fraud controls, structured monitoring and alerting.
Casino-style product	Applicable KYC/AML, age/geographic controls, responsible gaming, privacy and licensing.

Four stories that best represent my work

Story	What it demonstrates
Refresh ownership	Concurrency reasoning, asynchronous state ownership, stale-token prevention and recovery design.
Server-authoritative settlement	Trust boundaries, backend validation and transactional data integrity.
Render → Neon migration	Database operations, verification discipline and recoverable infrastructure change.
RNG/admin limitations	Security judgement and the ability to distinguish working functionality from production-grade controls.

Evidence-driven engineering philosophy

Prove the layer. Inspect the target. Keep one owner for shared state. Name the boundary of the implementation. That philosophy connects the project's most important decisions: I verified the database rather than trusting process completion, used an HTTP 404 to narrow the failing layer, centralized refresh rather than allowing competing token owners, kept settlement authority on the backend, and documented why demo RNG/static admin credentials are not sufficient for regulated production.

Capability summary

FULL-STACK OWNERSHIP

Flutter state/navigation through NestJS services to PostgreSQL and cloud operations.

BACKEND ENGINEERING

Protected APIs, DTO validation, server-side business rules and compatibility contracts.

CONCURRENCY / STATE

Single-owner token refresh with a central lock and latest-token request attachment.

DATA INTEGRITY

Transactional balance/ledger writes and

DEBUGGING

Layer-by-layer diagnosis across shell, HTTP

SECURITY JUDGEMENT

Explicitly identifies static-key, RNG, ledger,

JESUS AMEN'S COPY

preserved relational/migration state.	routing, deployment and database restoration.	concurrency and compliance boundaries.

Evidence note

This record was revised against the project files rather than inventing an authentication incident. The source confirms: (1) the updated API client makes dio_provider.dart the sole token-refresh owner, uses a single refresh lock and avoids stale-token retries; and (2) the live-table UI maps an HTTP 401 to a user-facing session-expired/login-again message. The files reviewed do not independently establish that a particular historical 401 was caused by the prior competing refresh architecture, so the document preserves that distinction.

Public-facing positioning intentionally uses “deployed” and “production-style” rather than claiming regulated real-money readiness. Credentials and live secret values are omitted.

FESTUS AWE'S COPY