SELECTED ENGINEERING CASE STUDY

MEGA888

Cloud PostgreSQL Migration & Backend Reliability

Render PostgreSQL → Local Windows migration tooling → Neon PostgreSQL

PROJECT OUTCOME

MEGA888 is a casino application I designed, developed, and deployed. This case study focuses specifically on the engineering work involved in migrating its PostgreSQL persistence layer from Render to Neon while preserving application data, relational integrity, indexes, and Prisma migration history.

Recruiter snapshot

31	192	680 KB	TLS 1.3
application / Prisma tables	archive TOC entries	validated custom-format backup	observed Neon connection

Core technologies

PostgreSQL 18.3	Neon	Render	NestJS	Prisma	Node.js	PowerShell

What I owned

- Prepared the Windows migration environment by installing PostgreSQL client tooling and validating pg_dump, pg_restore, and psql availability.
- Extracted the remote Render database into a reusable local custom-format archive, validated the archive before restore, and restored it into Neon using the direct endpoint.
- Preserved schema, table data, primary/foreign keys, indexes, unique constraints, and _prisma_migrations rather than recreating an empty schema.
- Diagnosed restore syntax, endpoint, PowerShell, API-routing, and prior database-connectivity issues using evidence-driven checks rather than assuming command success.
- Verified the migrated target through psql, Prisma Client generation, Prisma migration status, and NestJS development startup.

PORTFOLIO POSITIONING • Backend Engineering • Cloud Databases • Migration • Troubleshooting

FESTUS AWE

01 | The Engineering Challenge

Move an existing database without losing its history or relational integrity

MEGA888's backend originally depended on Render PostgreSQL. The migration objective was not simply to create new tables in Neon; it was to transfer the existing production database state while keeping the deployed NestJS/Prisma application compatible and preserving an explicit rollback path.

Preserve real database state

Move existing table data together with schema definitions, indexes, keys, constraints, and Prisma migration history.

Keep the application compatible

The destination needed to remain usable by the existing Prisma schema and NestJS backend without rebuilding production history from scratch.

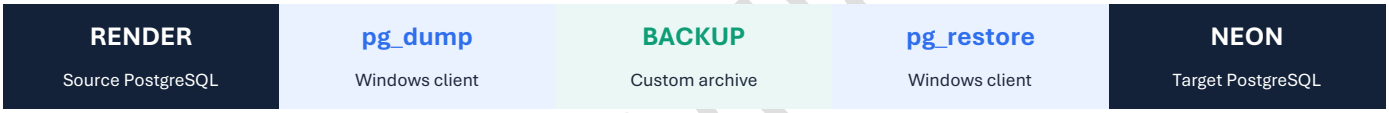
Create a recoverable migration path

Use a local, inspectable backup artifact so the target could be retried or restored without repeatedly depending on the source database.

Verify every layer

Prove source extraction, archive validity, target restoration, ORM compatibility, and backend startup independently.

Migration architecture



PostgreSQL/TLS → logical extraction → local recovery point → logical restore → PostgreSQL/TLS

WHY LOCAL POSTGRESQL INSTALLATION MATTERED

PostgreSQL was downloaded and installed on the local Windows machine specifically to obtain the native client utilities used for the migration. PowerShell provided the shell, but pg_dump performed the extraction, pg_restore validated/restored the archive, and psql verified the target database.

Data movement, precisely

- Stage 1: pg_dump ran locally, authenticated to the externally reachable Render PostgreSQL endpoint, read database objects and rows over the PostgreSQL connection, and wrote a custom-format archive to the Windows filesystem.
- The command used a relative output filename while PowerShell was operating in C:\Users\<LOCAL_USER>, so Windows resolved the file to C:\Users\<LOCAL_USER>\render_backup.dump. The local username is redacted in this public portfolio.
- Stage 2: pg_restore opened that local archive, connected to Neon over PostgreSQL/TLS, recreated database objects, and loaded the preserved data into the destination.

02 | Implementation & Technical Depth

A controlled migration with an inspectable recovery point

1. Prepare the migration workstation

- Installed PostgreSQL for Windows and made the command-line utilities available to PowerShell.
- Used pg_dump 18.3 for source extraction, pg_restore for archive validation/restore, and psql for interactive target verification.

2. Extract Render PostgreSQL

COMMAND PATTERN

```
pg_dump "<RENDER_EXTERNAL_DATABASE_URL>" -Fc -f render_backup.dump
```

The -Fc option produced PostgreSQL CUSTOM format rather than a plain SQL script. That gave pg_restore an inspectable archive suitable for controlled restore and retry.

3. Validate before touching the target

ARCHIVE VALIDATION

```
pg_restore -l "C:\Users\<LOCAL_USER>\render_backup.dump"
```

18.3	CUSTOM	gzip	192
pg_dump version	archive format	compression	TOC entries

4. Restore into Neon

SUCCESSFUL RESTORE PATTERN

```
pg_restore --verbose --clean --if-exists --no-owner --no-acl -d "<NEON_DIRECT_DATABASE_URL>"
"C:\Users\<LOCAL_USER>\render_backup.dump"
```

Restore choice	Engineering rationale
--verbose	Make each restore phase observable for diagnosis.
--clean + --if-exists	Support a controlled retry after an earlier ambiguous restore attempt, when some target objects might already exist while others might not.
--no-owner	Avoid recreating Render-specific ownership on Neon.
--no-acl	Avoid transferring provider-specific access-control metadata.
Direct Neon endpoint	Remove the pooler from the bulk restore path after the earlier ambiguous attempt.

03 | Troubleshooting & Engineering Judgment

The strongest evidence was not the happy path - it was how failures were isolated

Malformed restore connection string

Observed: pg_restore reported: missing "=" after "psql" in connection info string.
Diagnosis: psql had been incorrectly mixed into pg_restore connection syntax; the tools are separate executables.
Resolution: Use pg_restore directly with -d "<NEON_DATABASE_URL>" and the dump path.

Restore appeared to do nothing

Observed: The first attempt returned with little useful output, while Neon still showed no application tables.
Diagnosis: Shell return was not treated as proof of success. The backup was validated independently and the target was re-inspected.
Resolution: Switch to the direct Neon endpoint, enable --verbose, then verify the target with psql.

PowerShell curl behavior

Observed: Header syntax behaved unexpectedly when curl resolved to PowerShell Invoke-WebRequest.
Diagnosis: The command environment, not the API itself, changed the meaning of curl flags.
Resolution: Use curl.exe explicitly or PowerShell-native -Headers syntax.

Deployed API route returned 404

Observed: GET /api/admin/me returned a NestJS 404 after the database work.
Diagnosis: A 404 proves the request reached the application but does not establish database failure.
Resolution: Check global prefix, controller decorators, known valid routes, local/deployed build parity, and Render deployment state.

SECURITY JUDGMENT

Credentials exposed during troubleshooting were treated as compromised. The Neon password was reset, the exposed admin key was flagged for rotation, real secrets were excluded from documentation, and the local database dump was identified as sensitive because it contains application data.

Reliability decisions

- Kept the old Render database available during cutover verification to preserve a rollback path.
- Separated database-migration success from deployed API-route verification instead of conflating unrelated symptoms - explicit fault-domain isolation.
- Recognized the risk of data drift if new writes occur after the dump; final retirement should account for the backup timestamp and any post-backup writes.

What this case study demonstrates

Outcome	Evidence
Database migration	Render PostgreSQL successfully extracted, validated as a local archive, and restored into Neon.
Integrity preservation	Archive evidence included table data, PK/FK constraints, indexes, unique indexes, and _prisma_migrations.
Target verification	Application tables became visible in Neon and were owned by neondb_owner.
ORM compatibility	npx prisma generate and npx prisma migrate status completed successfully against the migrated configuration.
Backend compatibility	NestJS development startup compiled with “Found 0 errors. Watching for file changes.”
Production cutover	Database restoration and local backend compatibility checks succeeded. Final verification of the deployed Render backend against Neon remained a separate cutover task at the recorded checkpoint; the old Render database was retained as a rollback source until that verification was complete.

Competencies demonstrated

Database & Cloud Engineering PostgreSQL logical backup/restore, Render-to-Neon migration, TLS, direct/pooled endpoints, archive inspection, and schema/data verification.	Backend Integration NestJS/Prisma compatibility, migration-history preservation, target verification, and deployed API-route diagnosis.
Systematic Troubleshooting Fault-domain isolation across database, endpoint, shell, routing, and deployment layers using targeted evidence-driven checks.	Reliability & Security Rollback retention, data-drift awareness, secret rotation, credential redaction, and sensitive-backup handling.

Summary

For MEGA888, an application I designed and deployed, I migrated the existing PostgreSQL database from Render to Neon using PostgreSQL's native logical backup and restore tooling. I installed PostgreSQL tooling locally on Windows, used pg_dump 18.3 to create and validate a custom-format backup, and restored it through Neon's direct endpoint with pg_restore. I preserved the relational structure, application data, and Prisma migration history, verified the destination with psql and Prisma, and confirmed the NestJS backend compiled successfully. I also isolated an API-route 404 from the database migration itself, accounted for data-drift risk, and retained the old database as a rollback source during cutover verification.

PREVIEW COPY