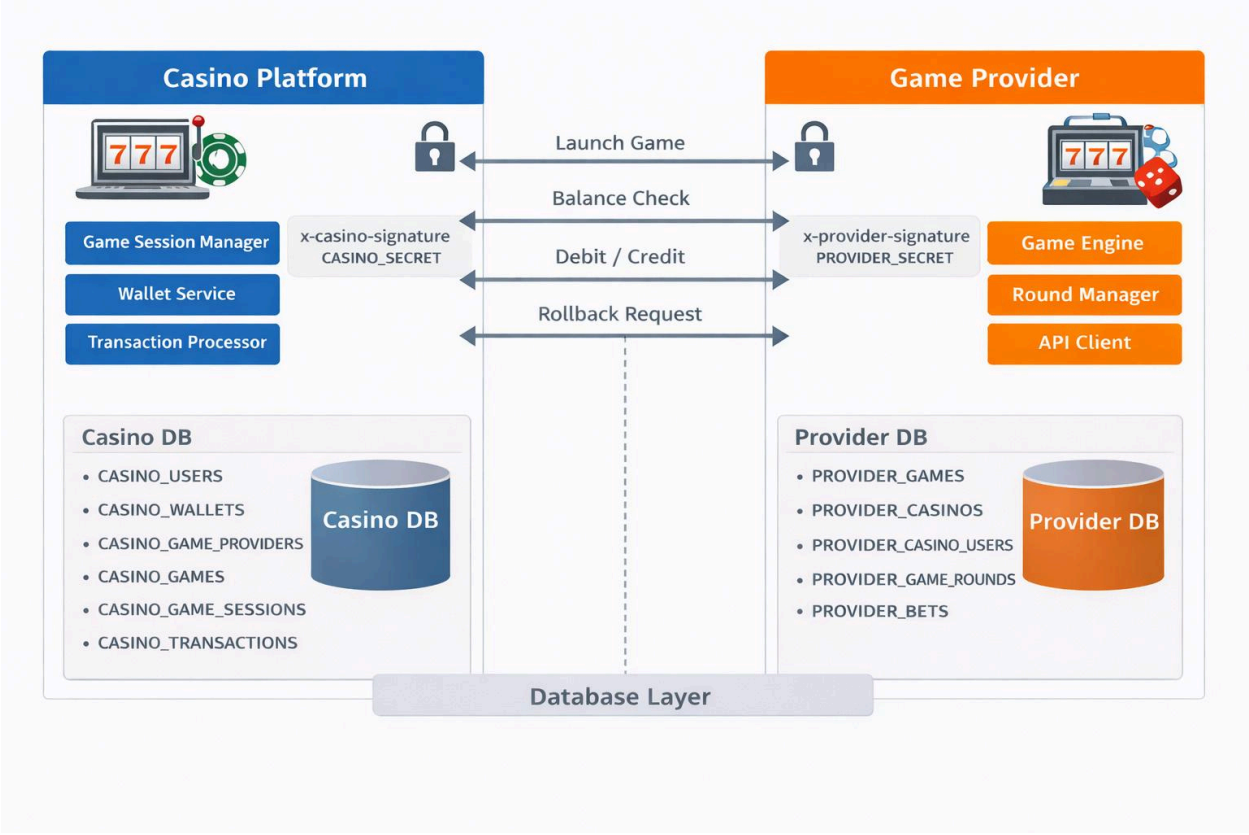
Backend Technical Test – Casino & Game Provider Integration

Company: Jaqpot Games
Role: Backend Engineer (Mid–Senior Level)
Estimated Time: 8–12 hours

High-Level Architecture Overview

The following diagram illustrates the high-level architecture and interaction flow between the Casino Platform and the Game Provider, including trust boundaries, responsibilities, and secure communication paths.



Overview

This technical test simulates a real-world integration between an online casino platform and an external game provider. You are required to implement both sides of the integration to demonstrate your ability to design APIs, handle transactional systems, ensure data integrity, and manage secure communication between distributed services.

Objectives

• Design and implement bidirectional API communication.
• Handle wallet balances and transactions safely and atomically.
• Implement idempotent transaction processing.
• Apply HMAC-based request authentication.
• Model a relational database schema suitable for gaming and financial systems.

System Scope

• **Casino Platform**: Hosts the player wallet, validates bets, and manages balances.
• **Game Provider (Jaqpot)**: Simulates game rounds and calls casino APIs during gameplay.

Implementation Scope Clarification

For the purposes of this technical test, **both the Casino Platform and the Game Provider may be implemented within the same codebase and the same physical database**. This is acceptable and expected in order to reduce setup complexity.

• Even when sharing a project and database, the two systems must remain **logically separated**.

• API endpoints must be clearly namespaced (**/casino/*** and **/provider/***).

• Database tables must be clearly prefixed (**CASINO_*** and **PROVIDER_***).

• Casino logic must not directly modify Provider tables, and vice versa.

Casino Platform – Responsibilities

• Launch game sessions initiated by a frontend client.
• Expose secure callback APIs for balance, debit, credit, and rollback.
• Maintain wallet balances with full transactional integrity.

• Guarantee idempotency for all provider-initiated transactions.
• Provide a test-driver endpoint to orchestrate a full simulated round (launch → provider simulation).

Game Provider – Responsibilities

• Accept game launch requests from the casino.• Create and manage game rounds.
• Simulate player bets and payouts.
• Call casino APIs for debits, credits, rollbacks, and balance checks.
• Expose a simulation endpoint that, after launch, validates session/user data and runs a full demo flow.

Required API Endpoints & Descriptions

Casino APIs

• **POST /casino/launchGame** – Initiated by a frontend or client application. Validates the player and wallet, creates a casino-side game session, and calls the Provider to initialize the corresponding provider-side session.

• **POST /casino/simulateRound** – Test-driver endpoint. Executes a launch flow and then calls the Provider simulation endpoint. Used to demonstrate a complete end-to-end round (at least: balance check, one or more bets/debits, one or more wins/credits, and at least one rollback).

• **POST /casino/getBalance** – Provider callback to retrieve the authoritative player balance. This endpoint must not mutate the state.

• **POST /casino/debit** – Provider callback to deduct funds for a bet. Must validate available balance, apply the debit atomically, and be strictly idempotent.

• **POST /casino/credit** – Provider callback to credit funds for a payout. Must be atomic, linked to the corresponding round/bet, and idempotent.

• **POST /casino/rollback** *** – Provider callback to reverse a previously accepted bet. Must enforce rollback rules (bets only, no payouts, tombstones) and be idempotent.

Provider APIs

• **POST /provider/launch** – Called by the Casino during game launch. Creates a provider-side game session and returns data required to start gameplay.

• **POST /provider/simulate** – Simulation endpoint called by the Casino (typically via **/casino/simulateRound**). Confirms session/user identifiers and performs a scripted demo flow that includes: at least one balance check, one or more bet debits, one or more payout credits, and at least one rollback request.

Technical Requirements

• Node.js (v18+)
• PostgreSQL (required)
• Express.js or equivalent framework• ORM or raw SQL (Sequelize, Prisma, TypeORM, etc.)
• JavaScript or TypeScript

Idempotency Requirements (IMPORTANT)

All provider-initiated money-moving endpoints on the Casino (/casino/debit, /casino/credit, /casino/rollback) must be idempotent. The Provider may retry requests due to timeouts, network errors, or uncertain outcomes.

• Each request must include a unique transactionId generated by the Provider.
• The Casino must store the first successful result and return it for duplicates.
• Duplicate requests must not create additional balance movements.
• The Provider simulation flow must be safe to re-run without double-charging or double-paying.

Rollback Rules (IMPORTANT) (OPTIONAL*)**

• Only bets can be rolled back. Payouts/credits must never be rolled back.
• No rollbacks are allowed for rounds that already have a payout.
• Tombstone rule: if the original bet transaction cannot be found, record a rollback idempotency marker and return success with no balance change.
• The tombstone provides auditability and prevents inconsistent retry behavior.

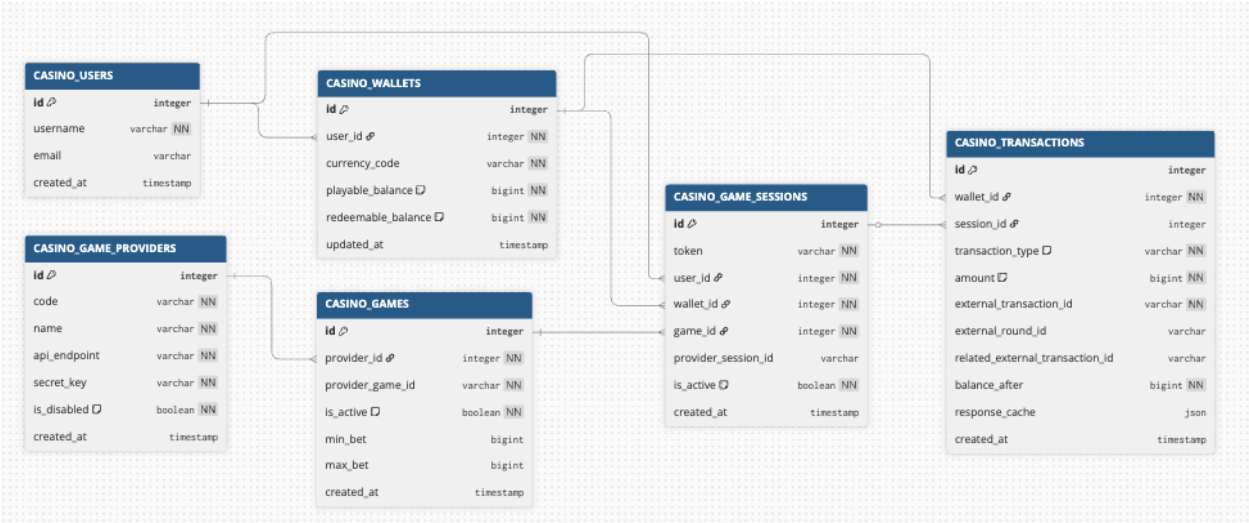
Minimum Database Requirements

You must design and implement the minimum set of database tables required to support the full flow. You are free to use raw SQL or any ORM of your choice. The list below defines the minimum conceptual schema expected for evaluation.

Casino Domain (CASINO_*):

Table	Purpose	Suggested fields (examples)
CASINO_USERS	Player identity and account metadata.	id (INT/UUID PK), username (VARCHAR), email (VARCHAR), created_at (TIMESTAMP)
CASINO_WALLETS	Authoritative balances per user and currency.	id (INT/UUID PK), user_id (FK), currency_code (VARCHAR), playable_balance (BIGINT), redeemable_balance (BIGINT), updated_at (TIMESTAMP)
CASINO_GAME_PROVIDERS	Provider registry and credentials.	id (INT/UUID PK), code (VARCHAR UNIQUE), name (VARCHAR), api_endpoint (VARCHAR), secret_key (VARCHAR), is_disabled (BOOLEAN)
CASINO_GAMES	Casino games mapped to provider games.	id (INT/UUID PK), provider_id (FK), provider_game_id (VARCHAR), is_active (BOOLEAN), min_bet (BIGINT), max_bet (BIGINT)
CASINO_GAME_SESSIONS	Session linking user, wallet, game, and provider session.	id (INT/UUID PK), token (VARCHAR UNIQUE), user_id (FK), wallet_id (FK), game_id (FK), provider_session_id (VARCHAR), is_active (BOOLEAN), created_at (TIMESTAMP)

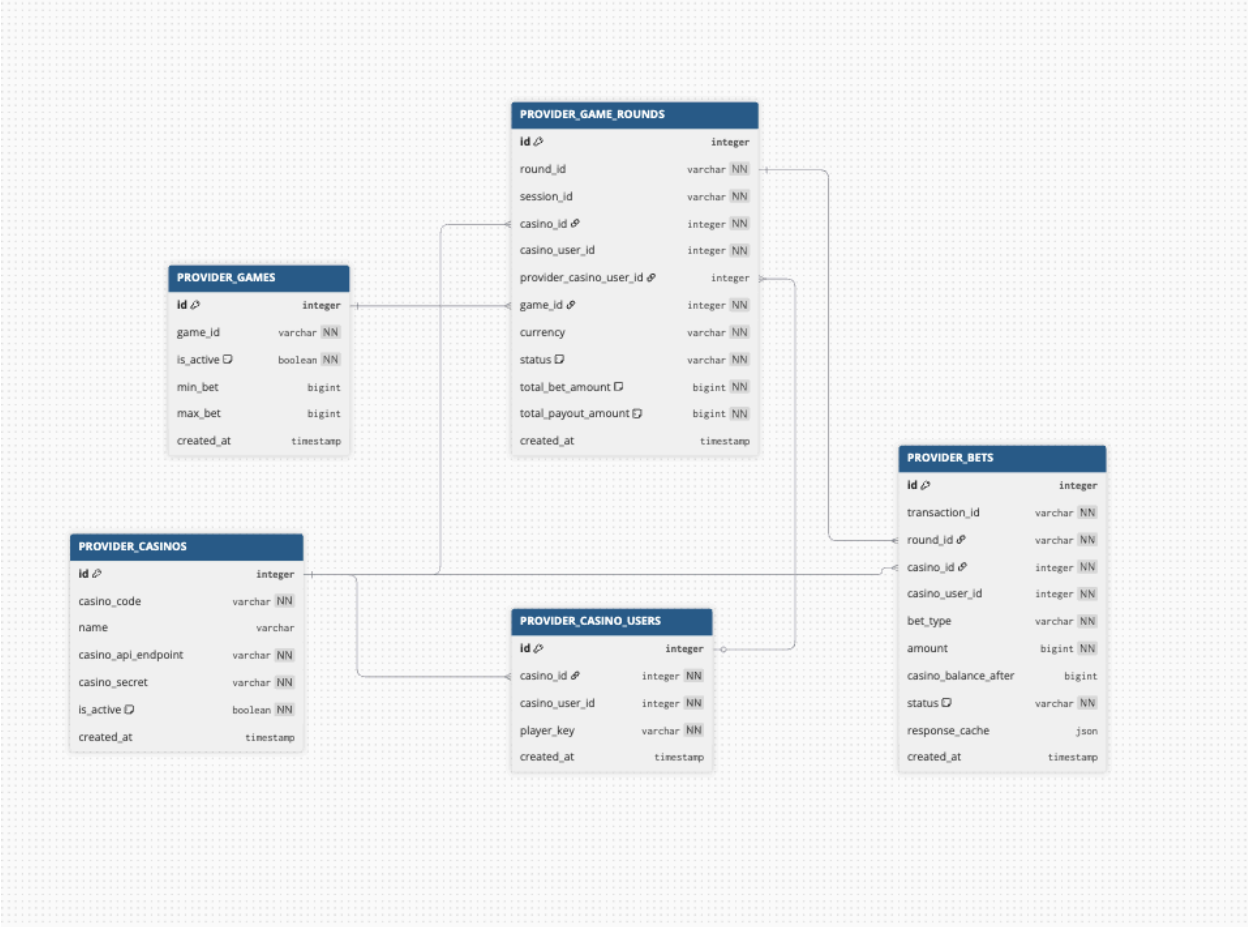
CASINO_TRANSACTIONS	Ledger of bets, payouts, rollbacks and idempotency cache.	id (INT/UUID PK), wallet_id (FK), session_id (FK), transaction_type (VARCHAR), amount (BIGINT), external_transaction_id (VARCHAR UNIQUE), related_external_transaction_id (VARCHAR), balance_after (BIGINT), response_cache (JSON), created_at (TIMESTAMP)



Provider Domain (PROVIDER_*):

Table	Purpose	Suggested fields (examples)
PROVIDER_GAMES	Provider game catalog.	id (INT/UUID PK), game_id (VARCHAR UNIQUE), is_active (BOOLEAN), min_bet (BIGINT), max_bet (BIGINT)
PROVIDER_CUSTOMERS	Mapping of casino players to provider customers.	id (INT/UUID PK), player_id (VARCHAR UNIQUE), casino_code (VARCHAR), external_user_id (VARCHAR), created_at (TIMESTAMP)

PROVIDER_GAME_ROUNDS	Grouping of bets and payouts per round.	id (INT/UUID PK), round_id (VARCHAR UNIQUE), session_id (VARCHAR), player_id (FK), game_id (FK), currency (VARCHAR), status (VARCHAR), total_bet_amount (BIGINT), total_payout_amount (BIGINT), created_at (TIMESTAMP)
PROVIDER_BETS	Each transaction attempt and casino response.	id (INT/UUID PK), transaction_id (VARCHAR UNIQUE), round_id (FK), bet_type (VARCHAR), amount (BIGINT), casino_balance_after (BIGINT), status (VARCHAR), response_cache (JSON), created_at (TIMESTAMP)



Required constraints: external transaction IDs must be unique per system, and balance updates must be performed atomically with appropriate locking.

Security Model Between Casino and Provider

The Casino and the Provider authenticate each other using HMAC-SHA256 signatures. Each direction of communication uses its own dedicated secret and header.

Security Headers & Secrets by Direction

Direction	Endpoint family	Header	Secret
Provider → Casino	`/casino/*`	`x-casino-signature`	`CASINO_SECRET`
Casino → Provider	`/provider/*`	`x-provider-signature`	`PROVIDER_SECRET`

Rule: The header name always identifies the receiving system.

When calling the Casino, use **x-casino-signature**.
When calling the Provider, use **x-provider-signature**.

Code example (Node.js, minimal and acceptable)

```JavaScript
import crypto from "crypto";

export function signBody(body, secret) {
      const payload = JSON.stringify(body);
```

```
      return crypto.createHmac("sha256",
      secret).update(payload).digest("hex");
}

export function verifySignature(providedSig, body, secret) {
      if (!providedSig) return false;
      const expectedSig = signBody(body, secret);

      // Constant-time compare (prevents timing attacks)
      try {
            const a = Buffer.from(providedSig, "hex");
            const b = Buffer.from(expectedSig, "hex");
            if (a.length !== b.length) return false;
            return crypto.timingSafeEqual(a, b);
      } catch {
            return false;
      }
}

// Provider → Casino (callbacks)
// Header: x-casino-signature// Secret: CASINO_SECRET
// const ok = verifySignature(req.header("x-casino-signature"),req.body,
process.env.CASINO_SECRET);
// Casino → Provider (launch/simulation)
// Header: x-provider-signature
// Secret: PROVIDER_SECRET
// const sig = signBody(payload, process.env.PROVIDER_SECRET);
```

Trust Boundaries & Responsibilities

- The Casino is the source of truth for wallet balances.
- The Provider must never assume or calculate balances independently.
- The Provider changes balances only by calling Casino APIs.
- The Casino must assume provider requests may be duplicated or reordered.

Deliverables

• Source code for both casino and provider services (Rollback logic is optional for extra points)
• Database schema and migrations.
• README with setup and execution instructions.
• Clear instructions to run and validate a full game round.
• A single command or documented sequence to run /casino/simulateRound end-to-end.

Evaluation Criteria

• Correctness and completeness of the integration flow (including the simulation path).
• Database design and data integrity.
• Security implementation.
• Code quality and structure.
• Clarity of documentation.

This test reflects real challenges encountered in online gaming, fintech, and transaction-heavy systems. Focus on correctness, clarity, and robustness rather than over-engineering.