Version: 3.0 FINAL
Date: 14 November 2025
Author: UE NOBLE IMPEX EOOD, Ruse, Bulgarian
Backend URL: https://backend.jackpotx.net
API Documentation: https://backend.jackpotx.net/api-docs.json (Password: qwer1234)

STAGE 1
INTRODUCTION & GENERAL OVERVIEW

1. INTRODUCTION

JackpotX is a modular, high-performance **iGaming backend platform** engineered to support modern online casino operations at enterprise scale.
The system implements a fully decoupled service architecture, clean separation of concerns, strict data consistency rules, and robust security enforcement across all layers.
The backend exposes **over 150 REST API endpoints**, built with **Node.js, TypeScript, Express 5, PostgreSQL, MongoDB, Redis**, and integrates seamlessly with multiple external game providers.
JackpotX is designed to operate in regulated, high-volume environments and supports:

- Multi-currency deployments
- Multi-language environments
- Full compliance workflows (KYC, AML, Player Safeguards)
- Real-time WebSocket communication
- Horizontal scaling
- Provider callback processing at high throughput
- Extensive analytics

This document describes the **entire backend system**, including architecture, database models, API structure, authentication flows, security standards, integrations, and operational guidelines.

1.1 Core Purpose of the JackpotX Backend

The JackpotX backend functions as the **central authority** for:

- Player authentication & authorization
- Wallet & balance management
- Game launch & provider communication
- Bets, wins, RTP, and GGR processing
- Payment processor operations
- Bonus engines, tournaments & jackpots
- CRM, segmentation & analytics pipelines
- Admin panel operations (RBAC, dashboards, KYC, reports)
- System monitoring & error recovery

It represents the authoritative source of truth for **all transactional, analytical, and compliance-critical data** within the platform.

1.2 Primary Features

💠 **Enterprise-Grade API System**

- Over **150 RESTful API endpoints**
- Fully documented via **OpenAPI 3.0**
- Uniform response structures
- Strict validation via **Zod runtime schemas**

💠 **Complete Casino Integration**

- Integration with **1000+ casino titles**
- Multi-provider support: Evolution, Pragmatic Play, NetEnt, Innova, etc.
- Real-time session management
- Game proxy for IP masking and anti-fraud routing

💠 **Advanced Security Layer**

- JWT access/refresh tokens
- Role-Based Access Control (RBAC)
- 2FA via Google Authenticator (TOTP)
- IP tracking, device fingerprinting, rate limits
- Enforced password policies (bcrypt hashing)

💠 **Multi-Database Architecture**

- PostgreSQL for critical transactional data
- MongoDB for session/volatile storage
- Redis for caching, queues & real-time messaging

💠 **Real-Time Communication**

- WebSocket (Socket.io)
- Balance updates
- Notification system
- Live chat & support
- Event streaming for jackpots / tournaments

💠 **Financial & Regulatory Compliance**

- Responsible gaming toolkit
- Self-exclusion
- Deposit limits
- KYC automation
- AML-ready activity logs
- Fraud prevention services

💠 **Enterprise Admin System**

- Full player management suite
- Payment oversight
- Provider-level reporting
- CRM segmentation
- User activity timeline
- System health diagnostics

1.3 Technology Stack Overview

Languages & Environment

- Node.js v20+ LTS
- TypeScript 5.8+
- Express.js 5

Databases

Database	Purpose
PostgreSQL 16+	Primary relational database, strict ACID compliance
MongoDB 7+	Session storage, temporary states, enterprise features
Redis 5+	Caching, pub/sub, rate limiting, WebSocket scaling

Security, Auth & Validation

- **jsonwebtoken** for JWT
- **bcrypt** for password hashing
- **helmet** for HTTP security headers
- **express-rate-limit** for circuit breakers
- **Zod** for input validation
- **Google Authenticator TOTP** for 2FA

API Documentation

- **swagger-jsdoc**
- **swagger-ui-express**

Real-Time Layer

- **Socket.io**
- Redis pub/sub for cluster mode scaling

Utilities

- **axios** for external provider communication
- **Puppeteer** for game proxy / integration handling
- **node-cron** for scheduled jobs

1.4 Platform Metrics & Statistics

📊 **Project Size**

- **217+ TypeScript files**
- **43 backend services**
- **40+ route definition files**
- **50+ PostgreSQL tables**
- **9 structured SQL migrations**
- **30+ pages of internal documentation**

⚡ **Performance**

- Supports **10,000+ concurrent WebSocket connections**
- **500 PostgreSQL pooled connections**
- Query timeout: **60 seconds**
- Rate limiting configurable per endpoint
- Automatic retry and fallback using circuit breaker patterns

1.5 Backend Architecture Summary

JackpotX follows a strict **layered architecture**:

```
Presentation Layer (Frontend)
        ↓
API Layer (Routes & Controllers)
        ↓
Middleware (Auth, RBAC, Validation, Rate Limiting)
        ↓
Service Layer (Business Logic)
        ↓
Data Access Layer (PostgreSQL, MongoDB, Redis)
        ↓

External Integrations (Game Providers, Payment Gateways,
Analytics, 2FA Services)
```

1.6 Supported Deployment Environments

Environment	Purpose	Characteristics
Development	Local development	Hot-reload, verbose logging
Staging	Pre-production	Full API, sandbox providers
Production	Live operations	Hardened security, rate limits, load balancing

1.7 Backend Characteristics Required for iGaming

The platform has been engineered to satisfy requirements specific to regulated iGaming markets:

- Deterministic financial calculations
- Atomic wallet operations (ACID transactions)
- Real-time reconciliation with providers
- Fraud detection hooks
- Player session tracking
- Responsible gaming enforcement
- Full audit trail for every critical operation
- Support for multi-region deployments

📘 STAGE 2
FULL BACKEND ARCHITECTURE

2. BACKEND ARCHITECTURE

The JackpotX backend is structured around a **clean, modular, and fully type-safe architecture**, optimized for scalability, maintainability, and high-volume casino traffic.
The architecture is designed to isolate responsibilities clearly, maximize performance, and support horizontal scaling with minimal coupling between modules.
The system follows a strict **Layered Architecture**, enriched with a **Modular Domain Structure**, service encapsulation, and a robust data access strategy.

2.1 Architectural Principles

The platform adheres to the following engineering principles:

2.1.1 Separation of Concerns

Each architectural layer has a single, clear responsibility:

- **Routes** → Handle HTTP routing & input
- **Controllers** → Interpret request and shape responses
- **Middlewares** → Security, validation, authorization
- **Services** → Business logic
- **Database Layer** → Persistent data management
- **Integrations** → External communication (providers, payments)

2.1.2 Strong Typing & Schema Validation

- End-to-end TypeScript coverage
- Zod validation for all request inputs
- Typed provider responses
- Strict null & undefined checks

2.1.3 Stateless API Layer

The API is stateless, except for:

- Refresh tokens
- WebSocket authenticated sessions
- Optional Redis caching

2.1.4 Consistency & ACID Integrity

All financial operations (bets, wins, transactions) use:

- Database transactions
- Row-level locks
- Atomic updates
- Strict sequence ordering

This ensures data correctness across wallet updates, game callbacks, and provider settlement.

2.1.5 Horizontal Scalability

Every architectural component supports scaling:

- Multiple API instances
- Redis-based event propagation
- PostgreSQL connection pooling
- Stateless services
- WebSocket load balancing

2.2 Directory Structure (Fully Documented)

Below is the **complete and accurate directory structure**, professionally rewritten to reflect enterprise documentation standards.

```
/src
├── api/                                # Domain-driven API modules
│   ├── auth/
│   ├── user/
│   ├── game/
│   ├── admin/
│   ├── payment/
│   ├── affiliate/
│   ├── promotion/
│   ├── support/
│   ├── crm/
│   ├── analytics/
│   ├── tournament/
│   ├── jackpot/
│   ├── kyc/
│   └── responsible-gaming/
│
├── routes/                             # 40+ route entrypoints
│   ├── auth.routes.ts
│   ├── user.routes.ts
│   ├── game.routes.ts
│   ├── admin.routes.ts
│   ├── promotion.routes.ts
│   ├── affiliate.routes.ts
│   ├── payment.routes.ts
│   ├── tournament.routes.ts
│   ├── jackpot.routes.ts
│   ├── kyc.routes.ts
│   ├── support.routes.ts
│   ├── provider-callback.routes.ts
│   └── index.enterprise.ts
│
├── services/                           # Core business logic (43 modules)
│   ├── auth.service.ts
│   ├── user.service.ts
│   ├── balance.service.ts
│   ├── game.service.ts
│   ├── innova-api.service.ts
│   ├── provider-callback.service.ts
│   ├── payment.service.ts
│   ├── withdrawal.service.ts
│   ├── affiliate.service.ts
│   ├── promotion.service.ts
│   ├── crm.service.ts
│   ├── analytics.service.ts
│   ├── reports.service.ts
│   ├── jackpot.service.ts
│   ├── tournament.service.ts
│   ├── risk-management.service.ts
│   ├── kyc.service.ts
│   ├── support-ticket.service.ts
│   ├── http.service.ts
│   ├── metadata.service.ts
│   ├── multilanguage.service.ts
│   ├── cron-manager.service.ts
│   └── enterprise-cron.service.ts
│
├── middlewares/                        # Authentication, RBAC, Validation
│   ├── authenticate.ts
│   ├── authorize.ts
│   ├── rate-limiter.middleware.ts
│   ├── validate.ts
│   ├── errorHandler.ts
│   ├── activity-logger.ts
│   ├── ip-tracking.middleware.ts
│   └── swaggerAuth.middleware.ts
│
├── db/                                  # Database access layer
│   ├── pool.ts                          # PostgreSQL pool
│   ├── mongo.ts                         # MongoDB connection
│   └── migrations/                      # SQL migrations (9 files)
│
├── configs/                             # Centralized configuration
│   ├── config.ts
│   └── env.ts
│
├── types/                               # TypeScript Definitions
│   ├── express.d.ts
│   ├── user.types.ts
│   ├── game.types.ts
│   └── global.types.ts
│
├── utils/                               # Helpers & system utilities
│   ├── logger.ts
│   ├── helpers.ts
│   ├── crypto.ts
│   └── formatter.ts
│
├── app.ts                               # Express app initialization
└── index.ts                             # Application entry point
```

2.3 Layered Architecture In Depth

Below is the expanded view of the JackpotX backend stack, rewritten in technical enterprise style.

<u>2.3.1 API Layer (Routes & Controllers)</u>

Responsibilities:

- Map HTTP routes to controllers
- Input parsing
- Pass-through to middlewares
- No business logic inside routes

Example structure:

/api/auth/login → auth.controller.login() ; /api/user/profile → user.controller.getProfile()

/api/games/play → game.controller.launch()

Controllers:

- Validate request shape
- Delegate to a service
- Build the final response

No controller contains domain logic.

2.3.2 Middleware Layer

Middlewares enforce:

Security

- JWT verification
- 2FA checks
- Rate limiting
- IP blocking
- Geo restrictions

Validation

Zod schemas ensure strict request formats.

Access Control

RBAC middleware validates:

```
role → permissions → resource
```

Logging

All player actions and sensitive events are logged:

- Admin updates
- Payment changes
- Support interactions
- Bet/win callbacks

2.3.3 Service Layer (Business Logic)

This is the **heart of the backend**.
Each service:

- Implements domain rules
- Interacts with the database
- Calls external providers
- Applies security policies
- Handles multi-step business logic
- Uses dependency-free, test-friendly code

Examples:

auth.service

- Login / register
- 2FA generation
- Token issuance
- Password hashing

balance.service

- Atomic wallet updates
- Transaction logging
- Bonus wallet logic
- Provider settlement

game.service

- Game launch routing
- Session creation
- Provider round handling

provider-callback.service

Handles:
- Bets
- Wins
- Refunds
- Rollbacks
- RTP/GGR rules

analytics.service

- Player segmentation
- Revenue reporting
- Conversion tracking

2.3.4 Data Access Layer

JackpotX uses a multi-database strategy:

PostgreSQL (Primary — ACID)

Used for:
- Balances
- Games
- Bets
- Payments
- KYC
- Jackpots
- CRM

- Reports

Queries are executed via:

- Parameterized SQL
- Connection pooling
- Optimised indexes

MongoDB (Secondary — Non-critical data)

Used for:

- Sessions
- Temporary storage
- Enterprise features

Redis (Cache & Pub/Sub)

Used for:

- Session caching
- Rate limiting keys
- WebSocket scaling
- Activity counters

2.4 External Integration Layer

Only the service layer interacts with external systems.

Supported Integrations:

Casino Providers

- Innova Gaming API
- IGPX Sportsbook
- Pragmatic Launcher
- JxOriginals (internal games)

Payment Providers

- Multiple processors
- Callback validation
- Fraud/risk hooks

Messaging

- Email (SMTP)
- SMS
- Push Notifications

Compliance Services

- TOTP (2FA)
- Document verification

2.5 Data Flow Diagram: Full Request Lifecycle

```
Client
  │
  ├── 1. HTTP Request
  │      (with JWT token)
  │
  ▼
API Layer (Route → Controller)
  │
  ├── 2. Middlewares
  │     - JWT decode
  │     - RBAC
  │     - Rate limit
  │     - Validation
  │
  ▼
Service Layer
  │
  ├── 3. Business Logic
  │     - compute
  │     - validate
  │     - call providers
  │
  ▼
Database Layer
  │
  ├── 4. PostgreSQL / MongoDB / Redis
  │
  ▼
Service Layer (Output)
  │
  ├── 5. Format response
  │
  ▼
Controller
  │
  ├── 6. Send JSON Response
  │
  ▼


Client UI
```

2.6 Scalability Model

Horizontal Scaling:

- Multiple Node.js instances under PM2 or Docker
- Redis pub/sub for WebSocket scaling
- Nginx load balancer

Database Scaling:

- PostgreSQL connection pooling
- Read replicas (optional)
- Partitioning for large tables (bets, rounds, transactions)

High Availability:

- Zero-downtime reloads
- Graceful shutdown
- Circuit breaker & retry logic
- Health check endpoints

2.7 Summary

The backend architecture of JackpotX is engineered around:

- High throughput
- ACID financial safety
- Modular domain-driven design
- Strong security
- Real-time event systems
- Clean separation of concerns
- Flexibility for future extensions

This architecture ensures stable, compliant, high-performance casino operations at enterprise scale.

📘 STAGE 3
DEVELOPMENT SETUP & ENVIRONMENT CONFIGURATION

3. DEVELOPMENT SETUP & CONFIGURATION

This section provides a full, production-grade guide for setting up the JackpotX backend development environment.
It covers system requirements, database installation, environment variable configuration, dependency installation, project startup, migrations, seeding, and troubleshooting.
The setup process is designed to ensure **deterministic development**, consistent environments, and secure handling of secrets.

3.1 System Requirements

To ensure stable performance and compatibility, the following system specifications are required.

3.1.1 Minimum Requirements (Local Development)

Component	Required
Node.js	v18+ (LTS)
npm	v8+
PostgreSQL	v14+
RAM	4 GB
Storage	10 GB
OS	macOS, Linux, Windows (WSL2 recommended)

These minimum specs are suitable for local development without external providers or heavy load.

3.1.2 Recommended (Staging / Production-like dev environment)

Component	Recommended
Node.js	v20 LTS
PostgreSQL	v16+
MongoDB	v7+
Redis	v6+
RAM	8–16 GB
Storage	50 GB SSD
OS	Ubuntu 22.04 LTS or Debian 12

3.2 Development Environment Setup

The backend can be installed on macOS, Linux, or Windows using WSL2.
Below you will find detailed installation steps.

3.2.1 Install Node.js & npm

macOS (Homebrew)

```
brew install node@20
node -v && npm -v
```

Ubuntu / Debian

```
sudo apt update
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt install -y nodejs
node -v
npm -v
```

3.2.2 Install PostgreSQL

macOS

```
brew install postgresql@16
brew services start postgresql@16
```

Ubuntu / Debian

```
sudo apt install postgresql postgresql-contrib
sudo systemctl enable postgresql
sudo systemctl start postgresql
psql --version
```

3.2.3 Install MongoDB
(Optional but recommended)

Ubuntu / Debian

```
curl -fsSL https://www.mongodb.org/static/pgp/server-7.0.asc \
| sudo gpg -o /usr/share/keyrings/mongodb-server-7.0.gpg --dearmor

echo "deb [signed-by=/usr/share/keyrings/mongodb-server-7.0.gpg] \
https://repo.mongodb.org/apt/ubuntu jammy/mongodb-org/7.0
multiverse" \
| sudo tee /etc/apt/sources.list.d/mongodb-org-7.0.list

sudo apt update
sudo apt install -y mongodb-org
sudo systemctl start mongod
sudo systemctl enable mongod
```

3.2.4 Install Redis (Optional but recommended)

```
sudo apt install redis-server
sudo systemctl start redis-server
redis-cli ping   # Expected: PONG
```

3.3 Project Setup

3.3.1 Clone the repository

```
git clone <repository-url> jackpotx-backend
cd jackpotx-backend
```

If installed on a server:

```
cd /var/www/html/backend.jackpotx.net
```

3.3.2 Install dependencies

```
npm install
```

This installs:

- Express 5
- PostgreSQL clients
- TypeScript
- Swagger tools
- JWT libraries
- Zod validation
- WebSocket server
- External provider clients

3.4 Database Configuration

3.4.1 PostgreSQL Setup

Open the PostgreSQL shell:

```
sudo -u postgres psql
```

Create a database user:

```
CREATE USER jackpotx_user WITH PASSWORD 'your_secure_password';
```

Create a development database:

```
CREATE DATABASE jackpotx_dev;
```

Assign privileges:

```
GRANT ALL PRIVILEGES ON DATABASE jackpotx_dev TO jackpotx_user;
```

Exit:

```
\q
```

3.4.2 MongoDB Setup
(Optional)

```
mongosh
use admin
db.createUser({
  user: "admin",
  pwd: "your_secure_password",
  roles: [{ role: "userAdminAnyDatabase", db: "admin" }]
})

use jackpotx
db.createUser({
  user: "jackpotx_user",
  pwd: "your_secure_password",
  roles: [{ role: "readWrite", db: "jackpotx" }]
})
exit
```

3.5 Environment Configuration (.env)

Create your environment file:

```
cp .env.example .env
nano .env
```

A fully rewritten, enterprise-style .env template:

```
# ======================================================================
# JACKPOTX BACKEND — DEVELOPMENT ENVIRONMENT CONFIGURATION
# ======================================================================

# --- App Core ---
PORT=3004
HOST=localhost
NODE_ENV=development

# --- PostgreSQL ---
DB_HOST=localhost
DB_PORT=5432
DB_USER=jackpotx_user
DB_PASS=your_secure_password_here
DB_NAME=jackpotx_dev
DB_POOL_MIN=2
DB_POOL_MAX=500
DB_CONNECTION_TIMEOUT=30000
DB_IDLE_TIMEOUT=60000
DB_QUERY_TIMEOUT=60000

# --- MongoDB (Optional) ---
MONGO_URI=mongodb://jackpotx_user:your_secure_password@localhost:27017/jackpotx?authSource=admin

# --- JWT Auth ---
JWT_ACCESS_SECRET=REPLACE_ME
JWT_REFRESH_SECRET=REPLACE_ME
JWT_ACCESS_TOKEN_EXPIRES=24h
JWT_REFRESH_TOKEN_EXPIRES=30d

# --- Provider Integrations ---
SUPPLIER_API_KEY=thinkcode
SUPPLIER_SECRET_KEY=2aZWQ93V8aT1sKrA
SUPPLIER_CALLBACK_URL=http://localhost:3004/api/innova/
SUPPLIER_OPERATOR_ID=thinkcode

# --- RTP & GGR ---
GGR_FILTER_PERCENT=0.5
```

```
GGR_TOLERANCE=0.05

# --- Swagger Docs ---
SWAGGER_PASSWORD=qwer1234

# --- Rate Limiting ---
RATE_LIMIT_STANDARD_MAX=999999
RATE_LIMIT_STANDARD_WINDOW_MS=900000
RATE_LIMIT_STRICT_MAX=999999
RATE_LIMIT_STRICT_WINDOW_MS=60000

# --- WebSocket ---
JXORIGINALS_WS_URL=ws://localhost:8443

# --- Redis (Optional) ---
REDIS_HOST=localhost
REDIS_PORT=6379

# --- SMTP (Optional) ---
SMTP_HOST=smtp.gmail.com
SMTP_PORT=587
SMTP_USER=your_email@gmail.com
SMTP_PASS=your_app_password
```

3.6 Generate Secure JWT Secrets

```
openssl rand -hex 32
```

Example output:

```
0582ece5b4eea773a75a77198ac1ab0e6afd695d139b59e562689f832d59ea2f
```

Add both secrets in `.env`.

3.7 Running Migrations

```
npm run migrate
```

This creates all PostgreSQL tables:

- Users
- Wallets
- Bets
- Games
- Transactions
- RTP settings
- Metadata
- Responsible gaming limits
- …over 50 relational tables

3.8 Seeding (Optional)

```
npm run seed
```

This generates:

- Test players
- Admin accounts
- Demo games
- Providers
- Role mappings

3.9 Start the Development Server

```
npm run dev
```

Expected startup log:

```
[INFO] JackpotX Backend Initializing...
[INFO] PostgreSQL Connected
[INFO] MongoDB Connected
[INFO] Routes Loaded: 40+
[INFO] WebSocket Server Running
[INFO] Server Listening at http://localhost:3004
```

3.10 Verifying the Installation

Health Check

```
GET http://localhost:3004/health
```

Expected output:

```
{
  "status": "healthy",
  "environment": "development",
  "database": {
    "postgresql": "connected",
    "mongodb": "connected"
  }
}
```

API Docs :

Open in browser:

```
http://localhost:3004/api-docs.json
Password: qwer1234
```

3.11 Troubleshooting Guide

Port already in use

```
lsof -i :3004
kill -9 <PID>
```

PostgreSQL connection error

```
sudo systemctl restart postgresql
```

Module not found

```
rm -rf node_modules package-lock.json
npm install
```

Migration failure

Run migrations one-by-one:

```
psql -h localhost -U jackpotx_user -d jackpotx_dev -f
src/db/migrations/001.sql
```

3.12 NPM Scripts Overview

Script	Purpose
`npm run dev`	Start dev server with hot reload
`npm start`	Start production server
`npm run build`	Compile TypeScript
`npm run migrate`	Apply database migrations
`npm run seed`	Seed test data
`npm run test`	Run full test suite
`npm run lint`	Lint code
`npm run format`	Format with Prettier

📘 STAGE 4
AUTHENTICATION & SECURITY

4. AUTHENTICATION & SECURITY

The authentication and security framework of JackpotX is designed to meet the strict requirements of regulated iGaming environments.
It protects user accounts, administrative operations, financial data, and game session integrity.
This chapter covers the entire authentication lifecycle, token management, RBAC, 2FA, rate-limiting, IP controls, CAPTCHA flows, and security best practices.

4.1 Authentication Overview

The platform uses a **multi-layered security architecture**, combining:

1. **JWT Authentication (Access + Refresh Tokens)**
2. **CAPTCHA Protection** for anti-bot defenses
3. **Password Hashing (bcrypt)**
4. **Two-Factor Authentication (Google Authenticator / TOTP)**
5. **Role-Based Access Control (RBAC)**
6. **Rate Limiting per Route Category**
7. **IP Tracking & Geo Restrictions**
8. **Activity Logging & Full Audit Trail**
9. **Provider-level Signature Validation**
10. **Strict Zod Validation for Input Payloads**

Each subsystem is designed to protect against high-impact threats:
Brute force, account takeover, replay attacks, token theft, fraudulent admin activity, and automated abuse.

4.2 Authentication Flow — Complete Lifecycle

Below is a fully rewritten, enterprise-grade explanation of the authentication pipeline.

4.2.1 Registration Flow

Steps

1. **Client requests a CAPTCHA:**
   `GET /api/auth/captcha`
2. **User submits registration form**, including:
   - username
   - email
   - password
   - CAPTCHA id + text
   - optional: referral/agent code
3. **Server validates CAPTCHA**
   → Blocks all automated account creation attempts.
4. **Password hashing using bcrypt**
   → Salt + secure one-way hashing.
5. **Account created in PostgreSQL**, default role: **Player**.
6. Optional: **2FA provisioning**
   - QR code returned
   - Secret key generated

Successful Response Example

```
{
  "success": true,
  "message": "Registered Successfully",
  "data": {
    "qr_code": "<svg>...</svg>",
    "auth_secret": "ABCD1234TOTPSECRET"
  }
}
```

4.2.2 Login Flow (With and Without 2FA)

Case 1: Standard Login

1. Validate username/email
2. Check password (bcrypt)
3. Generate Access Token (24h)
4. Generate Refresh Token (30 days)

Case 2: Login With 2FA Enabled

Additional steps:

- User must submit `auth_code` (TOTP)
- Backend verifies code via TOTP validator
- If valid → token issuance
- If invalid → reject with 401

Login Response Example

```
{
  "success": true,
  "message": "Logged in successfully",
  "token": {
    "access_token": "JWT_HERE",
    "refresh_token": "JWT_HERE",
    "role": {
      "id": 2,
      "name": "Player",
      "description": "Regular player account"
    }
  }
}
```

4.2.3 Token Refresh Flow

The Access Token expires after **24 hours**.
When expired:

1. API returns **401: Token expired**
2. Client calls:
   `POST /api/auth/refresh`
3. Backend validates Refresh Token
4. New Access Token is issued
5. No need to re-login

Refresh Token Response

```
{
  "success": true,
  "access_token": "NEW_ACCESS_TOKEN"
}
```

4.3 JWT Token Specification

4.3.1 Access Token

- Lifetime: **24 hours**
- Used for **all authenticated API calls**

4.3.2 Refresh Token

- Lifetime: **30 days**
- Used only to obtain new access tokens

4.3.3 Token Payload Structure

```
{
  "userId": 123,
  "username": "player123",
  "email": "player@example.com",
  "role": "Player",
  "role_id": 2,
  "iat": 1699876543,
  "exp": 1699962943
}
```

Includes:

- Identity
- Role
- JWT timestamps
- Security claims

4.4 Role-Based Access Control (RBAC)

The RBAC system ensures that each route is accessible only to the appropriate role.

4.4.1 Available Roles

ID	Role	Description
1	**Admin**	Full platform control
2	**Player**	Standard casino user
3	**Agent**	Affiliate / sub-affiliate operations
4	**Manager**	Operational oversight (games, finances, CRM)
5	**Support**	Tickets, KYC, user support

4.4.2 Example Access Rules

Feature	Player	Support	Manager	Admin
Play Games	✓	✓	✓	✓
Approve Withdrawals	✗	✗	✓	✓
Manage Players	✗	✓ (limited)	✓	✓
Edit Settings	✗	✗	✗	✓
View Reports	✗	✗	✓	✓
Manage Games	✗	✗	✓	✓

4.4.3 RBAC Middleware Flow

```
JWT Valid?
       ↓ yes
Role allowed for this route?
       ↓ yes
Continue request
```

If not allowed → `403 Forbidden`.

4.5 CAPTCHA Security Layer

Used for:

- Registration
- Login (if brute-force detected)
- Password resets
- High-risk IP ranges

Built using:

- Random text generator
- SVG distortion
- Expiration time
- 1-use tokens

4.6 Two-Factor Authentication (2FA)

4.6.1 TOTP-based

- Google Authenticator
- Authy
- Microsoft Authenticator

4.6.2 2FA Setup Flow

1. Backend generates:
   - TOTP secret
   - QR code in SVG
2. User scans QR
3. User confirms with the 6-digit OTP
4. 2FA is activated

4.6.3 When Required

- Admin roles (mandatory)
- Manager roles (recommended)
- Optional for players
- Triggered on suspicious IP/device

4.7 Password Policies

Enforced on registration:

- Minimum 8 characters
- At least 1 uppercase letter
- At least 1 number
- At least 1 special character
- Only hashed via **bcrypt** with dynamic salt

4.8 Rate Limiting Architecture

Every route category has its own limiter:

Category	Window	Max Requests
Authentication	15 min	5 attempts
Provider Callbacks	1 min	999999
Standard API	15 min	configurable
Strict API	1 min	configurable

Purpose:

- Prevent brute force
- Prevent DDoS
- Prevent provider callback abuse
- Protect admin endpoints

4.9 IP Tracking & Geo Restrictions

Features:

- IP-based rate limiting
- Blacklist/whitelist system
- IP stored in activity logs
- Device fingerprinting (optional)
- Geo-blocking of restricted regions

4.10 Audit Logging System

Every sensitive action is logged:

- Login / logout
- Password change
- Admin actions
- Payment changes
- Support ticket updates
- KYC approvals
- Game callback anomalies

Stored in PostgreSQL + optional MongoDB mirror.

4.11 Error Model for Authentication

Example error responses:

Invalid Credentials

```
{
  "success": false,
  "status": 401,
  "message": "Invalid username or password"
}
```

Invalid 2FA

```
{
  "success": false,
  "status": 401,
  "message": "Invalid authentication code"
}
```

Token Expired

```
{
  "success": false,
  "status": 401,
  "message": "Token expired"
}
```

Forbidden

```
{
  "success": false,
  "status": 403,
  "message": "Access denied"
}
```

4.12 Security Best Practices *(MANDATORY)*

- Never store tokens in URL
- Do not log access or refresh tokens

- Only store Access Token in localStorage
- Store Refresh Token in backend-only httpOnly cookie (recommended in production)
- Rotate JWT secrets regularly
- Use HTTPS everywhere
- Enforce 2FA on all admin accounts
- Limit failed login attempts
- Log all financial and admin actions

4.13 Summary

The JackpotX authentication system combines:

- Strong cryptography
- Multi-factor identity verification
- Role-based access enforcement
- Anti-fraud IP controls
- CAPTCHA and rate limiting
- Full audit trail
- Strict data validation

This ensures protection of all accounts, administrative functions, transactions, and gameplay flows across the platform.

📘 STAGE 5
PLAYER FRONTEND INTEGRATION

5. PLAYER FRONTEND INTEGRATION

This chapter documents the full API surface available to the **Player Frontend**—the public-facing application used by casino players.
The Player API is optimized for:

- Low latency
- High concurrency
- Real-time updates (WebSocket)
- Predictable response structures
- Strict validation
- Secure authentication flows

This section describes every category used in the player experience: authentication, profile, games, payments, promotions, responsible gaming, tournaments, jackpots, support, notifications, and dashboard features.

5.1 Player API Overview

The Player API is fully RESTful and grouped into functional domains:

Domain	Purpose
Auth	Registration, login, refresh, 2FA
User	Profile, balances, activity
Games	Game listing, filtering, launch
Promotions	Bonuses & promotional offers
Payments	Deposits, withdrawals, methods
Tournaments	Competition system
Jackpots	Real-time jackpot info

Responsible Gaming	Limits & self-exclusion
Support	Ticket system
Notifications	Real-time alerts
Dashboard	Home data & player activity

*All endpoints require **JWT Access Token** unless marked as public.*

5.2 Unified Response Format

All responses returned by the backend adhere to the following structure:

Success Response

```
{
  "success": true,
  "data": { ... }
}
```

Error Response

```
{
  "success": false,
  "status": 400,
  "message": "Validation error",
  "errors": [
    { "field": "email", "message": "Invalid email format" }
  ]
}
```

Consistency is critical for frontend integration and error handling.

5.3 AUTH MODULE

5.3.1 GET /api/auth/captcha

Purpose: Retrieve a CAPTCHA challenge for security.
Auth: Public
Returns:

- CAPTCHA ID
- CAPTCHA SVG (inline)

Response Example

```
{
  "success": true,
  "data": {
    "id": "captcha_12345",
    "svg": "<svg>...</svg>"
  }
}
```

5.3.2 POST /api/auth/register

Register a new user.
Auth: Public
Body:

```
{
  "username": "newplayer",
  "email": "player@mail.com",
  "password": "StrongPass123!",
  "captcha_id": "captcha_12345",
  "captcha_text": "ABCD"
}
```

Response

```
{
  "success": true,
  "message": "Registered Successfully",
  "data": {
    "qr_code": "<svg>...</svg>",
    "auth_secret": "AUTH_SECRET"
  }
}
```

5.3.3 POST /api/auth/login

Login with optional 2FA.
Auth: Public
Body:

```
{
  "username": "player123",
  "password": "Secret123!",
  "auth_code": "123456" // optional
}
```

5.3.4 POST /api/auth/refresh

Refreshing access token.

```
{
  "refresh_token": "JWT_REFRESH"
}
```

5.4 USER MODULE

5.4.1 GET /api/user/profile

Retrieve the authenticated user's profile.
Auth: Access Token
Example Response:

```
{
  "success": true,
  "data": {
    "id": 123,
    "username": "player123",
    "email": "player@mail.com",
    "phone": "+40123456789",
    "country": "RO",
    "status": "Active",
    "created_at": "2025-01-01T00:00:00Z"
  }
}
```

5.4.2 PUT /api/user/profile

Update user profile.
Allowed fields:

- first_name
- last_name
- phone
- language
- preferences

5.4.3 GET /api/user/balance

Returns wallet + bonus balance.

```
{
  "success": true,
  "data": {
    "balance": 152.40,
    "bonus_balance": 30.00
  }
}
```

(frontend uses WebSocket for real-time updates)

5.4.4 GET /api/user/transactions

Retrieve deposit/withdrawal history.
Supports filters:

```
?type=deposit
?type=withdrawal
?limit=50
```

5.4.5 GET /api/user/bets

Retrieve betting history.

5.5 GAME MODULE

The Game API allows browsing and launching games.

5.5.1 GET /api/games

Fetch filtered game list.
Query parameters:

```
?provider=pragmatic
?category=slots
?featured=true
?search=wolf
?limit=50
```

Response example:

```
{
  "success": true,
  "data": [
    {
      "id": "WOLF_GOLD",
      "name": "Wolf Gold",
      "provider": "Pragmatic Play",
      "type": "slot",
      "thumbnail": "https://..."
    }
  ]
}
```

5.5.2 GET /api/games/providers

List all providers.

5.5.3 GET /api/games/categories

List all categories.

5.5.4 POST /api/games/play/:id

Launch a game session.
Body:

```
{
  "demo": false
}
```

Backend returns provider launch URL:

```
{
  "success": true,
  "data": {
    "launch_url": "https://provider-url.com/?session=abc"
  }
}
```

5.5.5 Favorites

- **GET /api/games/favorites**
- **POST /api/games/favorite**
- **DELETE /api/games/favorite/:id**

5.6 PROMOTIONS MODULE

5.6.1 GET /api/promotions/active

Retrieve active promotions.

5.6.2 GET /api/promotions/:id

Promotion details.

5.6.3 POST /api/promotions/claim/:id

Claim a bonus.

5.7 PAYMENTS MODULE

5.7.1 GET /api/payment/methods

Available deposit/withdrawal methods.

5.7.2 POST /api/payment/deposit

Initiate a deposit.

Body:

```
{
  "amount": 50,
  "method": "card"
}
```

5.7.3 POST /api/withdrawals/request

Withdraw funds.
Body:

```
{
  "amount": 100,
  "method": "bank"
}
```

5.7.4 GET /api/withdrawals/history

History of withdrawals.

5.8 RESPONSIBLE GAMING MODULE

5.8.1 GET /api/responsible-gaming/limits

Retrieve limits.

5.8.2 POST /api/responsible-gaming/set-limit

Set deposit/time/amount limits.

```
{
  "type": "deposit",
  "amount": 300,
  "period": "daily"
}
```

5.8.3 POST /api/responsible-gaming/self-exclude

Self-exclusion rules.

5.9 SUPPORT MODULE

5.9.1 GET /api/support/tickets

List user tickets.

5.9.2 POST /api/support/tickets

Open a ticket.

5.9.3 POST /api/support/tickets/:id/reply

Reply to a ticket.

5.10 TOURNAMENT & JACKPOT MODULES

Tournaments

- `/api/tournaments/active`
- `/api/tournaments/:id/join`
- `/api/tournaments/:id/leaderboard`

Jackpots

- `/api/jackpots/active`
- `/api/jackpots/:id`

These endpoints integrate with the real-time engine for dynamic updates.

5.11 DASHBOARD MODULE

5.11.1 GET /api/home

Fetches homepage content:

- Featured games
- Active promotions
- Player activity
- Recent wins
- Recommended games
- Balance preview

5.12 Frontend Integration Guidelines

5.12.1 Use Axios with Auto-Refresh Interceptor

- Automatically retries requests when access token expires
- Ensures smooth UX
- Minimizes login prompts

5.12.2 Always Store Wallet State via WebSocket

Avoid fetching balance repeatedly.

5.13 Summary

The Player API provides a complete, secure, and scalable interface for building a casino frontend experience:

- Full authentication lifecycle
- Real-time balance & gameplay
- Game catalog with smart filtering
- Payment flows
- Bonus & promotion claims
- Support & ticketing
- Responsible gaming compliance

This API layer guarantees fast response times, stable integration, and predictable structure for frontend developers.

📘 STAGE 6
ADMIN PANEL INTEGRATION

6. ADMIN PANEL API INTEGRATION

The Admin Panel is the **operational backbone** of the JackpotX platform.
It provides administrators, managers, agents, and support staff with full visibility and control over:

- Player accounts
- Financial operations
- Game configurations
- Provider performance
- Reports & analytics
- Risk management
- KYC workflows
- Promotional systems
- Support ticketing
- CRM segmentation
- Enterprise-level settings

This chapter documents the complete API used by the **Admin Panel**, including RBAC rules and recommended usage patterns.

Every admin endpoint is protected by:

- **JWT Access Token**
- **Role-Based Access Control (RBAC)**
- **2FA Enforcement** (mandatory for Admin and Manager roles)
- **Rate Limiting**
- **Audit Logging**

6.1 Admin API Overview

The Admin API is grouped into the following domain modules:

Module	Description
Auth	Login, 2FA, token management
Users	Player lookup, editing, banning, limits
Games	Provider games, categories, settings
Payments	Deposits, withdrawals, financial operations
KYC	Verification workflows
Promotions	Bonuses, free spins, campaigns
Affiliate	Agent accounts, commission settings
Reports	Revenue, player activity, providers
Dashboard	Overview statistics
CRM	Segmentation & targeting
Support	Ticket management
Settings	Platform-level configuration
Tournaments	Administer competitions
Jackpots	Oversight of jackpot pools

All endpoints return structured JSON responses.

6.2 Admin Authentication Module

6.2.1 POST /admin/auth/login

Admin login with 2FA requirement.
Body:

```
{
  "username": "admin",
  "password": "AdminPass123!",
  "auth_code": "123456"
}
```

Response:

```
{
  "success": true,
  "token": {
    "access_token": "JWT_HERE",
    "refresh_token": "JWT_REFRESH_HERE",
    "role": "Admin"
  }
}
```

6.2.2 POST /admin/auth/refresh

Refresh access token.

6.2.3 GET /admin/auth/me

Returns admin session profile.

6.3 Dashboard Module

The dashboard provides real-time business metrics.

6.3.1 GET /admin/dashboard/overview

Returns key performance indicators (KPIs):

```
{
  "success": true,
  "data": {
    "total_players": 10234,
    "active_players_24h": 2301,
    "ggr_24h": 5321.40,
    "bets_24h": 48210,
    "withdrawal_pending": 42,
    "deposits_today": 123,
    "provider_performance": [...],
    "recent_big_wins": [...]
  }
}
```

6.4 User Management Module

This is one of the largest modules in the Admin Panel.

6.4.1 GET /admin/users

Search players with filters:

```
?username=
?email=
?status=
?country=
?sort=created_at:desc
?limit=100
```

6.4.2 GET /admin/users/:id

Full player profile:

- account info
- balances
- KYC status
- activity
- responsible gaming limits
- login history

6.4.3 PUT /admin/users/:id

Update player metadata:

```
{
  "status": "Suspended",
  "notes": "Suspicious IP activity."
}
```

Actions (protected by RBAC)

- **Ban user** — `/admin/users/:id/ban`
- **Suspend account** — `/admin/users/:id/suspend`
- **Reset password** — `/admin/users/:id/reset-password`
- **Force logout** — `/admin/users/:id/logout`
- **Adjust balance** (Manager+ only) — `/admin/users/:id/adjust-balance`
- **Assign agent** — `/admin/users/:id/assign-agent`

6.5 Game Management Module

6.5.1 GET /admin/games

Returns all providers & games.
Filters:

```
?provider=
?status=active/inactive
?featured=true
```

6.5.2 PUT /admin/games/:id

Update game properties:

```
{
  "status": "inactive",
  "featured": false
}
```

6.5.3 GET /admin/games/providers

Provider-level status.

6.5.4 Provider Callback Logs

Monitor callback performance and failures.

6.6 Payment Management Module

6.6.1 GET /admin/payments/deposits

Search & filter deposits.

6.6.2 GET /admin/payments/withdrawals

View pending, approved, rejected withdrawals.

6.6.3 POST /admin/payments/withdrawals/:id/approve

(Manager or Admin only)

6.6.4 POST /admin/payments/withdrawals/:id/reject

Requires reason:

```
{
  "reason": "KYC not completed."
}
```

6.6.5 POST /admin/payments/manual-adjustment

Manual credit/debit (with audit entry).

6.7 KYC Management Module

6.7.1 GET /admin/kyc/pending

List users awaiting verification.

6.7.2 GET /admin/kyc/:id

Full KYC submission details:

- documents
- metadata
- timestamps
- IP address
- history

6.7.3 POST /admin/kyc/:id/approve

Approves KYC.

6.7.4 POST /admin/kyc/:id/reject

Rejects document(s).
Body Example:

```
{
  "reason": "Document blurry. Upload clearer photo."
}
```

6.8 Promotion & Bonus Management Module

6.8.1 GET /admin/promotions

List all promotions.

6.8.2 POST /admin/promotions

Create a promotion.

```
{
  "title": "Welcome Bonus 100%",
  "type": "deposit_bonus",
  "amount": 100,
  "wager": 35
}
```

6.8.3 PUT /admin/promotions/:id

Update promotion properties.

6.8.4 POST /admin/promotions/:id/activate

Activate campaign.

6.8.5 POST /admin/promotions/:id/deactivate

Deactivate campaign.

6.9 Affiliate & Agent Management

Endpoints:

- GET /admin/affiliate/agents
- POST /admin/affiliate/agents
- GET /admin/affiliate/stats
- PUT /admin/affiliate/commission
- GET /admin/affiliate/tree

Supports:

- Multi-level commission structures
- Sub-agent trees
- Conversion reporting

6.10 Reports & Analytics Module

6.10.1 GET /admin/reports/financial

Daily / weekly / monthly:

- GGR
- NGR
- Bets
- Wins
- Deposits
- Withdrawals

6.10.2 GET /admin/reports/users

User retention reports:

- R1, R7, R30 retention
- Depositor segments
- Country breakdown

6.10.3 GET /admin/reports/providers

Provider-level performance:

- RTP
- Rounds
- Revenue
- Errors

6.11 CRM Module

6.11.1 GET /admin/crm/segments

List all segments.

6.11.2 POST /admin/crm/segments

Create segment:

```
{
  "name": "VIP Highrollers",
  "condition": "balance > 1000 AND bets_30d > 200"
}
```

6.11.3 POST /admin/crm/broadcast

Send targeted notifications.

6.12 Support Ticketing Module

6.12.1 GET /admin/support/tickets

List all tickets.

6.12.2 GET /admin/support/tickets/:id

Ticket details.

6.12.3 POST /admin/support/tickets/:id/reply

Admin replies to ticket.

6.13 Settings Module

Platform-level settings:

- Maintenance mode
- Payment windows
- Provider configurations
- RTP settings
- Limits
- Languages & localization
- Email templates
- Risk management flags

6.13.1 GET /admin/settings

Retrieves all settings.

6.13.2 PUT /admin/settings

Updates system settings.

6.14 Tournaments & Jackpots Module

Tournaments

- `/admin/tournaments`
- `/admin/tournaments/create`
- `/admin/tournaments/:id/update`
- `/admin/tournaments/:id/prizes`
- `/admin/tournaments/:id/delete`

Jackpots

- `/admin/jackpots`
- `/admin/jackpots/:id/update`
- `/admin/jackpots/:id/payout`

6.15 Admin Security Controls

Mandatory:

- 2FA for all Admin & Manager accounts
- IP logging & device fingerprinting
- Full audit trail
- Access attempts stored

- Rate limits on sensitive routes
- Permission-based visibility

6.16 Summary

The Admin API provides complete operational control of the JackpotX platform, including:

- Player management
- Financial workflows
- Game configuration
- Provider monitoring
- Promotions & CRM
- KYC compliance
- Risk controls
- Support processes
- Reporting & analytics
- Enterprise system configuration

Every endpoint is carefully secured using RBAC, 2FA, logging, and strict validation to ensure operational integrity in a regulated industry.

📘 STAGE 7
WEBSOCKET & REAL-TIME COMMUNICATION

7. REAL-TIME ENGINE (WEBSOCKET)

The JackpotX real-time communication layer is powered by **Socket.io** with **Redis Pub/Sub** for full horizontal scalability.
This system enables instant, low-latency updates required for modern iGaming platforms, including:

- Real-time balance updates
- In-game events
- Jackpot feeds
- Tournament leaderboards
- Notifications
- Live support chat
- Backend → frontend broadcast messages

The Real-Time Engine is optimized for **10,000+ concurrent active connections** and operates independently of the REST API layer.

7.1 Architecture Overview

The WebSocket system is implemented as a separate service inside the backend, with the following topology:

```
+--------------+      +-----------------+      +-------------+
|  Frontend    | <--->|  API + WS Node  |<---->|   Redis     |
+--------------+      +-----------------+      +-------------+
      ↑                     ↑   ↑                  ↑
      |                     |   |                  |
   Real-Time Events         |   |               Pub/Sub
      |                     |   |
      +---------------------+---+
                Horizontal Scaling
```

Components:

- **Socket.io server** – handles connections and channels
- **Redis Pub/Sub** – synchronizes events across multiple Node.js instances
- **Auth middleware** – validates tokens before joining rooms
- **Event dispatcher** – pushes updates to frontend clients

7.2 Connection Requirements

To establish a WebSocket connection:

7.2.1 Client Connection URL

```
wss://backend.jackpotx.net
```

7.2.2 Required Query Parameter

The Access Token must be sent during connection:

```
wss://backend.jackpotx.net?token=ACCESS_TOKEN
```

If missing or invalid → connection refused.

7.3 Authentication Flow for WebSocket

1. Client connects with Access Token
2. Server verifies token via JWT
3. Token is checked for:
   - expiration
   - role
   - user status (active/suspended/banned)
4. If valid → connection approved
5. Client is placed into:
   - `user:<id>` personal room
   - `global` broadcast room

7.4 WebSocket Events (Player Side)

Below is the complete list of **player-facing** real-time events.

7.4.1 balance.update

Triggered whenever:
- bet placed
- win received
- bonus applied
- manual adjustment
- deposit or withdrawal

Payload:

```
{
  "balance": 120.50,
  "bonus_balance": 30.00
}
```

7.4.2 notification

System notifications pushed directly to the player.
Payload Example:

```
{
  "type": "promotion",
  "title": "New Bonus!",
  "message": "You received 20 Free Spins on Book of Dead.",
  "timestamp": "2025-11-14T12:00:00Z"
}
```

7.4.3 promotion.update

Sent when a promotion or bonus affecting the user changes status.
Payload:

```
{
  "promotion_id": 101,
  "status": "activated"
}
```

7.4.4 Jackpot Update

Global or personal jackpot updates.
Types:

- Progressive jackpot update
- Jackpot win broadcast
- Jackpot reset

Payload Example:

```
{
  "jackpot_id": 5,
  "amount": 12345.67,
  "event": "increment"
}
```

7.4.5 tournament.update

Real-time leaderboard or tournament state change.

```
{
  "tournament_id": 22,
  "position": 3,
  "score": 12800
}
```

7.4.6 chat.message

Used for live support or in-game chat.

```
{
  "from": "Support",
  "message": "How can we help you today?",
  "timestamp": "2025-11-14T12:45:00Z"
}
```

7.4.7 session.logout

Triggered when:

- account is suspended
- admin force-logout
- token compromised
- multiple device login conflict

Payload:

```
{
  "reason": "Session invalidated"
}
```

Frontend must:

1. Clear local tokens
2. Redirect to login screen

7.5 WebSocket Events (Admin Panel)

Admin users also receive real-time updates.

7.5.1 admin.notification

```
{
  "type": "withdrawal_request",
  "user_id": 123,
  "amount": 250
}
```

7.5.2 kyc.update

Triggered when a user submits new KYC documents.

7.5.3 support.ticket.new

New ticket from a player.

7.6 Server → Server Real-Time Communication

Redis Pub/Sub events:

Channel	Purpose
`balance_update`	Sync balance across WS clusters
`jackpot_channel`	Global jackpot increments
`provider_events`	Game provider callbacks
`admin_alerts`	Backend system alerts
`tournament_updates`	Cluster-wide tournament sync

Every event is serialized as JSON to maintain compatibility.

7.7 Room Structure

7.7.1 Personal Room

`user:<id>`

- individual balance updates
- personal notifications
- support replies

7.7.2 Global Room

`global`

- jackpot updates
- system announcements
- maintenance alerts

7.7.3 Group Rooms

Optional for:

- tournaments
- VIP groups
- segmented marketing alerts

7.8 WebSocket Error Handling

Standardized error payload:

```
{
  "success": false,
  "event": "error",
  "message": "Unauthorized"
}
```

Common errors:

- `unauthorized`
- `token_expired`
- `invalid_payload`
- `too_many_connections`
- `rate_limited`

7.9 Recommended Frontend Integration Pattern

7.9.1 Auto-Reconnect Enabled

Socket.io handles reconnection automatically.

7.9.2 Ping Interval Monitoring

If the client does not receive server pings → reconnect.

7.9.3 Throttle UI Updates

Do not re-render UI more than 10 times per second.

7.9.4 Local Cache

Cache last-known balance to reduce flickering.

7.10 Security Model for Real-Time Layer

7.10.1 Per-Connection Rate Limits

Prevents spamming and fake events.

7.10.2 Token Validation

Every connection is revalidated periodically (configurable).

7.10.3 IP Binding

Optional: restrict session to single IP.

7.10.4 Event Signatures (optional)

Cryptographic signing of sensitive events.

7.11 Summary

The JackpotX Real-Time Engine supports:
- Immediate balance updates
- Live jackpot monitoring
- Tournament synchronization
- Support chat
- Personalized notifications
- Admin alerts
- Multi-node horizontal scaling

Engineered for high volume and low latency, it ensures a responsive and engaging player experience across the entire platform.

📘 STAGE 8
COMPLETE API REFERENCE

8. API REFERENCE — FULL ENDPOINT CATALOG

This chapter provides the **complete, structured list of all REST API endpoints** available in the JackpotX backend.
Each module includes:
- Endpoint description
- Method & URL
- Request body, params & query
- Required authentication
- RBAC role restrictions
- Standard response examples

All endpoints follow:

- **RESTful design**
- **OpenAPI 3.0 specification**
- **JSON request/response format**
- **Zod validation schemas**
- **Consistent error model**

8.1 API Structure Overview

All endpoints follow the scheme:

```
https://backend.jackpotx.net/api/<module>/<endpoint>
```

Modules covered:

1. **Auth**
2. **User**
3. **Games**
4. **Payment**
5. **Withdrawals**
6. **Promotions**
7. **KYC**
8. **Tournaments**
9. **Jackpots**
10. **Responsible Gaming**
11. **Support**
12. **Affiliate**
13. **Admin Panel (15 sub-modules)**

8.2 AUTH API

POST /api/auth/captcha

Generate CAPTCHA.
Auth: Public
Response:

```
{
  "success": true,
  "data": {
    "id": "CAPTCHA_ID",
    "svg": "<svg>...</svg>"
  }
}
```

POST /api/auth/register

Create player account.
Auth: Public
Body:

```
{
  "username": "newuser",
  "email": "mail@mail.com",
  "password": "StrongPass123!",
  "captcha_id": "123",
  "captcha_text": "ABCD",
  "ref_code": "AGENT001"
}
```

Response: Success + QR code for TOTP.

POST /api/auth/login

Login with or without 2FA.

```
{
  "username": "player1",
  "password": "Secret12!",
  "auth_code": "123456"
}
```

Returns:

```
{
  "success": true,
  "token": {
    "access_token": "JWT",
    "refresh_token": "JWT",
    "role": { "id": 2, "name": "Player" }
  }
}
```

POST /api/auth/refresh

Renew access token.

GET /api/auth/me

Profile from access token.

8.3 USER API

GET /api/user/profile

Retrieve personal profile.

PUT /api/user/profile

Update profile:

```
{
  "first_name": "John",
  "last_name": "Doe",
  "language": "en"
}
```

GET /api/user/balance

Wallet & bonus balance.

GET /api/user/transactions

Transaction history
(Deposits, withdrawals, bonuses, adjustments)

GET /api/user/bets

Bet history with filters.

GET /api/user/activity

Login history, last IPs, device info.

8.4 GAMES API

GET /api/games

Fetch entire catalog with filtering:

```
?provider=pragmatic
?category=classic
?search=gold
?limit=100
```

GET /api/games/providers

Provider listing.

GET /api/games/categories

Game categories.

POST /api/games/play/:game_id

Launch a game session.
Body:

```
{ "demo": false }
```

Response:

```
{
  "success": true,
  "data": {
    "launch_url": "https://provider/start?session=XYZ"
  }
}
```

Favorites

- `GET /api/games/favorites`
- `POST /api/games/favorite`
- `DELETE /api/games/favorite/:id`

8.5 PAYMENT API

GET /api/payment/methods

List deposit methods.

POST /api/payment/deposit

Initiate deposit.

```
{
  "amount": 50,
  "method": "visa"
}
```

GET /api/payment/deposit/history

Deposit history.

<u>8.6 WITHDRAWAL API</u>

POST /api/withdraw/request

Player withdrawal request.

```
{
  "amount": 200,
  "method": "bank_transfer",
  "iban": "RO49AAAA..."
}
```

GET /api/withdraw/history

Withdrawal logs.

<u>8.7 PROMOTION API</u>

GET /api/promotions/active

Active promotions.

GET /api/promotions/:id

Promotion details.

POST /api/promotions/claim/:id

Claim bonus.

GET /api/promotions/history

Bonus history.

8.8 KYC VERIFICATION API

GET /api/kyc/status

KYC state.

POST /api/kyc/upload

Upload documents.

```
{
  "type": "id_front",
  "file": "base64..."
}
```

GET /api/kyc/documents

List submitted documents.

8.9 TOURNAMENT API

GET /api/tournaments/active

List active events.

GET /api/tournaments/:id

Tournament details.

POST /api/tournaments/:id/join

Join event.

GET /api/tournaments/:id/leaderboard

Leaderboard.

8.10 JACKPOT API

GET /api/jackpots/active

Active jackpots.

GET /api/jackpots/:id

Jackpot status.

8.11 RESPONSIBLE GAMING API

GET /api/responsible-gaming/limits

Existing limits.

POST /api/responsible-gaming/set-limit

Set:

- Deposit limits
- Time limits
- Loss limits

POST /api/responsible-gaming/self-exclude

Self-exclusion:

```
{
  "duration": "6_months"
}
```

8.12 SUPPORT API

GET /api/support/tickets

Player's tickets.

POST /api/support/tickets

Open ticket:

```
{
  "subject": "Deposit not received",
  "message": "I deposited 50 EUR and it did not appear."
}
```

POST /api/support/tickets/:id/reply

Player reply.

8.13 AFFILIATE API

POST /api/affiliate/apply

Player applies as agent.

GET /api/affiliate/stats

Agent stats:

- Referrals
- Qualified players
- Commissions

GET /api/affiliate/referrals

List recruited players.

8.14 ADMIN PANEL API

(Full Admin API was covered in detail in Stage 6, here is the endpoint index.)

Admin Auth

- POST `/admin/auth/login`
- POST `/admin/auth/refresh`
- GET `/admin/auth/me`

Dashboard

- GET `/admin/dashboard/overview`

User Management

- GET `/admin/users`
- GET `/admin/users/:id`
- PUT `/admin/users/:id`
- POST `/admin/users/:id/ban`
- POST `/admin/users/:id/suspend`
- POST `/admin/users/:id/reset-password`
- POST `/admin/users/:id/adjust-balance`

KYC

- GET `/admin/kyc/pending`
- GET `/admin/kyc/:id`
- POST `/admin/kyc/:id/approve`
- POST `/admin/kyc/:id/reject`

Payments

- GET `/admin/payments/deposits`
- GET `/admin/payments/withdrawals`
- POST `/admin/payments/withdrawals/:id/approve`
- POST `/admin/payments/withdrawals/:id/reject`

Promotions

- GET `/admin/promotions`
- POST `/admin/promotions`
- PUT `/admin/promotions/:id`

Affiliate

- GET `/admin/affiliate/agents`
- POST `/admin/affiliate/agents`
- GET `/admin/affiliate/stats`

Reports

- GET `/admin/reports/financial`
- GET `/admin/reports/providers`
- GET `/admin/reports/users`

Settings

- GET `/admin/settings`
- PUT `/admin/settings`

Tournaments

- POST `/admin/tournaments`
- PUT `/admin/tournaments/:id`
- DELETE `/admin/tournaments/:id`

Jackpots

- GET `/admin/jackpots`
- PUT `/admin/jackpots/:id`
- POST `/admin/jackpots/:id/payout`

8.15 Error Reference

All endpoints return a standard error format:

```
{
  "success": false,
  "status": 401,
  "message": "Unauthorized"
}
```

Common status codes:

Code	Meaning
400	Bad request / validation
401	Token missing / expired
403	Forbidden by RBAC
404	Resource not found
409	Conflict (duplicate)
500	Internal error

8.16 Summary

This complete API reference provides:

- Full route catalog
- Structured domain grouping
- Unified authentication model
- Professional request/response structure
- Integration guidelines for frontend & admin systems

It serves as the authoritative source for all developers integrating with JackpotX.

📘 STAGE 9
ERROR HANDLING & EXCEPTION ARCHITECTURE

9. ERROR HANDLING ARCHITECTURE

The JackpotX backend implements a **centralized, consistent, and fault-tolerant error-handling system**, engineered to ensure:

- predictable responses for frontend and admin consumers
- clean separation between operational and programming errors
- secure masking of internal error details in production

- full integrity of financial and transactional operations
- comprehensive logging for audit, compliance, and debugging
- controlled failover mechanisms and fallback strategies

This chapter documents the complete error architecture, including classification, structure, examples, propagation, logging, and circuit-breaker integration.

9.1 Error Categories

Errors in the platform are grouped into **five official categories**:

9.1.1. Validation Errors

Returned when input does not match the required schema.
Generated by **Zod** and controller-level checks.

9.1.2. Authentication Errors

Triggered by login, JWT issues, missing tokens, or 2FA failures.

9.1.3. Authorization / RBAC Errors

Access denied due to insufficient permissions or invalid role.

9.1.4. Business Logic Errors

Errors thrown inside services:
- insufficient balance
- invalid withdrawal request
- game not available
- promotion not applicable

9.1.5. System & Internal Errors

Unexpected failures:
- database connection issues
- Redis timeouts
- provider API failure
- unhandled exceptions

These are logged in the audit/error logs but never exposed to the client.

9.2 Unified Error Response Format

Every error in the system returns a standardized JSON response:

Error Structure

```
{
  "success": false,
  "status": 400,
  "message": "Validation error",
  "errors": [
    {
      "field": "email",
      "message": "Invalid email format"
    }
  ]
}
```

Fields Explained

Field	Meaning
`success`	Always `false` for errors
`status`	HTTP error status code
`message`	Human-readable error
`errors`	Optional array of field-level issues

9.3 HTTP Status Codes Used

The platform uses a strict, deterministic mapping:

Code	Category	Meaning
400	Validation	Invalid input / format
401	Authentication	Invalid or missing token
403	Authorization	RBAC or restricted access
404	Resource	Entity not found
409	Conflict	Duplicate or invalid state
429	Rate Limit	Too many requests
500	System	Server or internal error

9.4 Validation Error Examples

Missing required field

```
{
  "success": false,
  "status": 400,
  "message": "Validation error",
  "errors": [
    { "field": "password", "message": "Password is required" }
  ]
}
```

Invalid format

```
{
  "success": false,
  "status": 400,
  "message": "Validation error",
  "errors": [
    { "field": "email", "message": "Invalid email format" }
  ]
}
```

9.5 Authentication Error Examples

Invalid credentials

```
{
  "success": false,
  "status": 401,
  "message": "Invalid username or password"
}
```

Token expired

```
{
  "success": false,
  "status": 401,
  "message": "Token expired"
}
```

2FA validation failed

```
{
  "success": false,
  "status": 401,
  "message": "Invalid authentication code"
}
```

9.6 Authorization (RBAC) Error Examples

```
{
  "success": false,
  "status": 403,
  "message": "Access denied"
}
```

Triggers:

- insufficient role
- restricted module
- admin-only endpoint

9.7 Business Logic Error Examples

Insufficient balance

```
{
  "success": false,
  "status": 400,
  "message": "Insufficient balance"
}
```

Withdrawal locked due to KYC

```
{
  "success": false,
  "status": 400,
  "message": "KYC verification required before withdrawal"
}
```

Promotion not eligible

```
{
  "success": false,
  "status": 400,
  "message": "Promotion is not available for this user"
}
```

9.8 Provider Callback Error Examples

Provider callbacks use a special response format, identical to Innova/Pragmatic standards.

Invalid signature

```
{
  "status": "error",
  "message": "Invalid callback signature"
}
```

Game round not found

```
{
  "status": "error",
  "message": "Round not found"
}
```

Rollback not allowed

```
{
  "status": "error",
  "message": "Rollback not permitted"
}
```

9.9 System Error Examples

Below errors are masked to avoid exposing internals.

Internal error

```
{
  "success": false,
  "status": 500,
  "message": "Internal server error"
}
```

Database timeout

```
{
  "success": false,
  "status": 500,
  "message": "Database unavailable. Please try again later."
}
```

9.10 Centralized Error Handler

All errors propagate through:

```
middlewares/errorHandler.ts
```

Responsibilities:

- normalize error format
- detect error type
- mask sensitive fields
- log error details
- attach correlation ID
- return safe JSON structure

9.11 Correlation ID System

Every request receives a unique trace ID:

```
X-Correlation-ID: 7f029bd4-c1f8-4ce5-b307-a1c23150d835
```

Used to:

- track errors across microservices
- align API logs with provider callbacks
- debug wallet/transaction issues

Frontend teams should include this ID in bug reports

9.12 Logging & Audit Integration

All errors are logged to:

9.12.1. PostgreSQL Audit Logs

For:

- admin actions
- financial errors
- security breaches

9.12.2. File-Based Logs (Winston)

Daily rotating logs:

- `error.log`
- `combined.log`

9.12.3. Optional External Log Aggregators

- Grafana Loki
- Elastic Stack
- Graylog
- Datadog

9.13 Circuit Breaker System

High-risk operations (provider callbacks, payment integrations) use circuit breakers to:

- detect high failure rate
- stop sending traffic to the failing module
- return fallback response
- auto-recover after cooldown

Example fallback:

```
{
  "success": false,
  "status": 503,
  "message": "Service temporarily unavailable"
}
```

9.14 Retry & Backoff Strategy

Used for:

- provider API calls
- email sending
- payment gateways

Pattern:

- first retry after 1s
- second retry after 2s
- exponential backoff

9.15 Developer Debug Mode

When running in development:

```
NODE_ENV=development
```

Responses include:

- stack trace
- error code
- raw error message

Never exposed in production !

9.16 Summary

The JackpotX error-handling architecture ensures:

- Predictable, consistent API behavior
- Clear separation between error types
- Safe masking of internal errors
- Full compliance with regulated iGaming requirements
- Comprehensive auditability
- Seamless debugging and traceability
- Stable fallback during provider or system failures

This robust system plays a critical role in platform stability and operational safety.

📘 STAGE 10
BEST PRACTICES & SECURITY GUIDELINES

10. BEST PRACTICES & SECURITY

This chapter outlines every critical principle, standard, and procedure required to ensure **safe, compliant, and high-performing operations** across the entire JackpotX backend.
It includes coding guidelines, architectural practices, database design rules, OWASP controls, anti-fraud systems, and iGaming-specific protections.
These best practices are mandatory for all developers and operators integrating or extending the JackpotX platform.

10.1 Security Principles (Core Pillars)

The platform follows six fundamental security guidelines:

10.1.1. Zero Trust Architecture

Every request must be authenticated and validated, even internal ones.

10.1.2. Least Privilege

Each user, role, and service receives only the access it strictly needs.

10.1.3. Defense in Depth

Multiple security layers: JWT, RBAC, 2FA, CAPTCHA, rate limiting, audit logs.

10.1.4. Secure by Default

Features must be built safe from the beginning, not patched later.

10.1.5. Separation of Duties

High-risk workflows require layered approval (e.g., withdrawals).

10.1.6. Fail Secure

If a system fails, it must fail closed*, not* open*.*

10.2 Coding Best Practices

10.2.1 Use Strict TypeScript Everywhere

- Enable strict mode
- No implicit anys
- No untyped responses
- Interfaces for every payload

10.2.2 Avoid Business Logic in Controllers

Controllers:

- Validate input
- Forward to service
- Format output

All logic must be inside service layers.

10.2.3 Use Dependency Injection for External Clients

Makes testing, mocking, and swapping providers easy.

10.2.4 Always Wrap DB Operations in Try/Catch

This ensures:

- clean rollback
- proper logging
- correct error formatting

10.2.5 Use Parameterized SQL Queries Only

Never concatenate SQL strings.

10.2.6 No Circular Dependencies

Strict domain isolation rules.

10.3 Database Best Practices

10.3.1 Use ACID Transactions for Financial Operations

Balance updates must always be:

- atomic
- consistent
- isolated
- durable

10.3.2 Implement Row-Level Locks for Wallet Updates

Prevents:
- double debit
- race conditions
- duplicate bets

10.3.3 Use Proper Indexing

Index:
- frequently queried columns
- foreign keys
- timestamps used for filtering

10.3.4 Maintain Audit Tables

Every sensitive event must be logged:
- admin actions
- financial updates
- KYC changes
- risk flags

10.3.5 Separate OLTP and Analytics Workloads

Avoid heavy analytical queries on transactional DB.
Use replicas or dedicated analytics DB if needed.

10.4 API Security Best Practices

10.4.1 Always Use HTTPS

TLS 1.2+ required.

10.4.2 JWT Access/Refresh Tokens

- Use strong secrets
- Rotate regularly
- Store refresh tokens in httpOnly cookies (recommended)

10.4.3 Implement Request Rate Limits

Each module has dedicated rate limits.

10.4.4 Enforce RBAC on All Sensitive Routes

Admin-only endpoints must be locked behind role checks.

10.4.5 Validate All Inputs with Zod

Prevents:
- injection
- malformed data + missing fields

10.4.6 Prevent Overexposing Data

Never expose:
- internal IDs
- database errors
- stack traces (except in development)

10.5 OWASP Compliance

The JackpotX backend mitigates all ***OWASP Top 10*** *vulnerabilities.*

A1: Broken Access Control

- RBAC
- Permissions per endpoint
- Admin override checks

A2: Cryptographic Failures

- bcrypt for passwords
- TLS enforced
- Token secrets rotated

A3: Injection

- Parameterized SQL
- Input validation
- Safe ORM abstractions

A4: Insecure Design

- Threat modeling
- Defense in depth
- Strong architectural review

A5: Security Misconfiguration

- Strict CORS
- Disabled x-powered-by
- Hardened headers (Helmet)

A6: Vulnerable Components

- Regular deps audit
- Zero tolerance for outdated libs

A7: Identification Failures

- JWT
- 2FA
- Device/IP tracking

A8: Software Integrity

- Locked dependency versions
- No auto-update in production

A9: Logging & Monitoring

- Structured logs
- Real-time alerting
- Audit trails

A10: SSRF Protection

- Whitelisted external hosts for providers

10.6 Anti-Fraud & Risk Mitigation (iGaming Specific)

iGaming platforms face sophisticated fraud patterns. JackpotX includes built-in protections.

10.6.1 Multi-Account Detection

- IP matching
- Device fingerprinting
- Login pattern analysis

10.6.2 Bonus Abuse Prevention

- bonus eligibility check
- wagering requirements
- deposit history validation

10.6.3 Payment Fraud Detection

- suspicious deposit patterns
- velocity checks
- sudden balance transfers

10.6.4 Chargeback Protection

Monitoring:

- failed payments
- recurring disputes
- inconsistent deposit/withdraw cycles

10.6.5 KYC & AML Enforcement

- document verification
- PEP/sanction checks (optional)
- transaction monitoring
- red flag triggers

10.7 Operational Best Practices

10.7.1 Graceful Shutdowns

Server should:

- reject new requests
- complete ongoing operations
- then close connections

10.7.2 Health Check Endpoints

For load balancers & uptime monitors.

10.7.3 Use PM2 or Docker for Process Management

Cluster mode recommended.

10.7.4 Use Environment Files for Secrets

Never commit secrets to Git.

10.7.5 Blue/Green Deployments

Zero downtime production updates.

10.8 Logging & Monitoring

10.8.1 Structured Logging

Using Winston/JSON logs.

10.8.2 Distributed Tracing

Correlation IDs for all requests.

10.8.3 Log Aggregation Systems

(Optional integrations):

- Grafana Loki
- ELK Stack
- Datadog
- Graylog

10.8.4 Alerting

Alerts for:

- failed provider callbacks
- high error rates
- DB latency increases
- suspicious admin actions

10.9 Performance Best Practices

10.9.1 Cache Hot Data with Redis

Avoid unnecessary DB queries.

10.9.2 Use Pagination Everywhere

Never return full lists.

10.9.3 Offload Heavy Jobs to Cron/Queue

Examples:

- daily reports
- expired bonus cleanup
- tournament settlement

10.9.4 Optimize Queries

- limit, offset
- indexes
- partitioning (bets, transactions)

10.10 Deployment Security

10.10.1 Harden Linux Servers

- disable password authentication
- use SSH keys
- install security patches
- firewall allowed ports only

10.10.2 Enforce SSL Termination

At NGINX or load balancer.

10.10.3 Use Fail2Ban

Protects from brute-force attempts.

10.10.4 Disable Unused Services

Lean production environment.

10.11 Compliance Best Practices (iGaming Regulations)

10.11.1 KYC Requirements

Full verification before:

- high withdrawals
- large deposits
- suspicious activity

10.11.2 Responsible Gaming Policies

Enforced limits:

- deposit
- loss
- session time

10.11.3 Data Retention Rules

Typically:

- 5–7 years for transaction records
- 10 years for KYC documents (depending on jurisdiction)

10.11.4 GDPR Compliance

- right to access
- right to erasure
- encrypted storage
- data minimization

10.12 Summary

The JackpotX Best Practices & Security guidelines provide a comprehensive foundation for:

- secure code
- reliable infrastructure
- regulatory compliance
- fraud prevention
- consistent data integrity
- stable deployments
- professional software engineering standards

Following these practices ensures safety, performance, and long-term scalability for any operator running the JackpotX platform.

📘 STAGE 11
TESTING & DEBUGGING

11. TESTING & DEBUGGING FRAMEWORK

The JackpotX platform includes a fully structured **testing and debugging ecosystem** designed to guarantee reliability, correctness, and stability across all backend components.
This chapter outlines:

- Test hierarchy (unit, integration, e2e)
- Provider callback simulators
- Payment sandbox strategies
- Logging & debugging tools
- Recommended methodologies
- Load testing & high-concurrency validation
- Error reproduction procedures
- CI/CD test pipeline

Testing follows **AAA (Arrange-Act-Assert)** methodology, strict isolation, and enforcement of deterministic results — essential for financial and regulatory environments like iGaming.

11.1 Test Types

JackpotX implements ***six categories of tests****, each covering a different surface area of the platform.*

11.1.1 Unit Tests

Purpose: Validate isolated logic inside services and utilities.
Characteristics:

- No external services
- Mock database queries
- Mock provider calls
- Fast execution (< 20ms/test)
- Pure TypeScript units

Examples:

- wallet calculations
- bet/win settlement formulas
- bonus eligibility functions
- JWT generation/validation

11.1.2 Integration Tests

Purpose: Validate communication between modules.
Includes:

- controllers → services → DB
- DB transactions
- callback workflows
- multi-step business logic

Executed on a temporary PostgreSQL test database.

11.1.3 End-to-End (E2E) Tests

Simulate a real player flow:

1. register
2. login
3. deposit
4. launch game
5. place bet
6. receive win (via provider callback)
7. withdraw funds

These tests ensure that **the entire system functions correctly from start to finish**.

11.1.4 Provider Simulation Tests

Critical for iGaming.
Simulated providers:

- Innova Gaming
- IGPX Sportsbook
- JxOriginals Internal Games

Tests include:

- bet callbacks
- win callbacks
- rollback

- round closures
- duplicate transaction prevention

Simulators mimic:

- incorrect signatures
- high-latency
- failure responses
- packet loss

11.1.5 Load & Stress Tests

Used to validate platform behavior under:

- 10,000+ simultaneous WebSocket connections
- 500+ RPS on standard endpoints
- provider callback bursts
- jackpot increment spikes

Tools:

- k6
- Locust
- Artillery

KPIs:

- 99th percentile latency
- database throughput
- memory stability
- CPU spikes
- WebSocket packet loss

11.1.6 Security Tests

Covers:

- brute force
- SQL injection
- replay attacks
- JWT manipulation
- race conditions
- bonus exploit attempts

11.2 Test Folder Structure

```
/tests
├── unit/
│   ├── user.service.test.ts
│   ├── balance.service.test.ts
│   ├── bonus.logic.test.ts
│   └── utils/
├── integration/
│   ├── auth.integration.test.ts
│   ├── payment.integration.test.ts
│   ├── kyc.integration.test.ts
├── e2e/
│   ├── player-flow.test.ts
│   ├── game-callback-flow.test.ts
│   └── admin-flow.test.ts
├── providers/
│   ├── innova-callback.test.ts
│   ├── sportsbook-callback.test.ts
└── mocks/
    ├── db.mock.ts
    ├── redis.mock.ts
    ├── provider.mock.ts
```

11.3 Mocks & Fakes

Mocks ensure tests run deterministically.

Database Mock

Simulates PostgreSQL queries for unit tests.

Redis Mock

Simulates pub/sub without external Redis.

Provider Mock

Simulates bet/win callbacks with variations:

- valid signature
- invalid signature
- duplicate callbacks
- rollback scenarios

11.4 Running Tests

Run Unit Tests

```
npm run test:unit
```

Run Integration Tests

(Requires test DB)

```
npm run test:integration
```

Run E2E Tests

```
npm run test:e2e
```

Run All Tests

```
npm test
```

11.5 Debugging Tools

11.5.1 Logger (Winston)

Every request & error is logged with:

- timestamp
- route
- correlation ID
- user ID
- IP address
- error code

Example entry:

```
[2025-11-14T12:22:35Z] ERROR /api/payment/deposit
correlation: 82ff-234a-ab12
user: 123
message: Provider timeout
```

11.5.2 Debug Mode

Enable detailed logs:

```
NODE_ENV=development
DEBUG=jackpotx:*
```

Shows:

- request payloads
- SQL queries
- provider retries
- WebSocket events

11.5.3 Postman Collections

Exported collection includes:

- all endpoints
- request samples
- environment variables
- test scripts

11.5.4 API Docs (Swagger)

```
https://backend.jackpotx.net/api-docs.json
Password: qwer1234
```

Interactive testing available.

11.6 Debugging Provider Callbacks

Provider callbacks can be complex. JackpotX includes special tools.

11.6.1 Callback Replay System

Replays a provider callback based on ID — used for debugging failed rounds.

11.6.2 Transaction Trace Log

Shows:

- callback payload
- processed transaction
- before/after wallet
- signature validation
- rollback state

11.6.3 Duplicate Callback Detection

If the provider sends the same callback twice:

- hash is checked
- duplicate is ignored
- result returned as success

11.7 Debugging Payment Integrations

Sandbox modes supported:

- Card processor sandbox
- Bank transfer mock
- Crypto mock (if enabled)

Test flows:

- insufficient funds
- double deposit attempt
- currency mismatch
- KYC-blocked withdrawal

11.8 Debugging WebSocket Layer

Tools:

- connection viewer
- user → room mapping
- event log per user
- real-time packet monitor

Issues detected:

- dropped connections
- token expiration mid-session
- concurrency spikes

11.9 Continuous Integration (CI)

CI pipeline (GitHub Actions or GitLab CI):

1. Install dependencies
2. Lint code
3. Run unit tests
4. Spin up test DB via Docker
5. Run integration tests
6. Run e2e tests
7. Build project
8. Produce artifacts

Build fails if:

- test fails
- code coverage drops below threshold
- linting errors exist

11.10 Debugging Workflow (Standard Procedure)

When an issue appears:

1. Identify error code
2. Retrieve correlation ID
3. Check Winston logs
4. Check PostgreSQL audit logs
5. Replay request/callback if applicable
6. Reproduce in development
7. Patch & test
8. Deploy through CI

9. Monitor logs post-fix

Documented in the internal Jira workflow.

11.11 Performance Testing

Using **k6** or **Locust**:

Scenarios tested:

- 1000 concurrent logins
- 500 rounds/minute provider callbacks
- 10k WebSocket clients
- 100 deposits/second

Metrics collected:

- latency (p50, p95, p99)
- throughput
- CPU usage
- memory stability
- DB locks
- rollback rates

11.12 Summary

The JackpotX testing & debugging framework ensures:

- deterministic correctness
- stable financial operations
- safe provider integrations
- consistent behavior under load
- smooth ops for admins & players
- rapid isolation of issues
- regulatory compliance

The testing suite is designed to handle **complex iGaming flows**, minimizing downtime and ensuring maximum reliability.

📘 STAGE 12
PRODUCTION DEPLOYMENT & INFRASTRUCTURE

12. PRODUCTION DEPLOYMENT

The JackpotX backend follows a **hardened, scalable, and fully container-ready deployment model**, designed for high availability, fault tolerance, and low-latency iGaming operations.

This chapter documents the complete process of:

- Infrastructure preparation
- Environment configuration
- Server and process management
- NGINX reverse proxy setup
- TLS/SSL termination
- Docker deployment model
- Load balancing

- Horizontal and vertical scaling
- Monitoring & alerting
- Backup & disaster recovery procedures

The deployment pipeline ensures stable, 24/7 production uptime with zero downtime during updates.

12.1 Deployment Models Supported

JackpotX can be deployed using one of three models:

Model 1
Bare-Metal / VPS Deployment

- Ubuntu 22.04 LTS or Debian 12
- PM2 process manager
- NGINX reverse proxy
- PostgreSQL + MongoDB + Redis installed on server(s)

Model 2
Docker-Based Deployment

- Docker Compose for simple environments
- Optional Kubernetes for large operators
- Ideal for multi-node scaling

Model 3
Hybrid Deployment

- Node.js API in Docker
- PostgreSQL on dedicated machine
- Redis on separate server
- Load balancer in front

12.2 Hardware Requirements

Minimum (Small Operator)

- 4-core CPU
- 8 GB RAM
- 80 GB SSD
- 500 Mbps bandwidth

Recommended (Medium Operator)

- 8-core CPU
- 16–32 GB RAM
- 200+ GB NVMe SSD
- Multi-node Redis + DB replication

High-Volume (Enterprise Operator)

- 16+ core CPU
- 64–128 GB RAM
- Dedicated DB node
- Redis cluster
- Load balancer cluster
- Kubernetes deployment

12.3 System Preparation

12.3.1 Required Packages

```
sudo apt update
sudo apt install -y build-essential git curl nginx ufw
```

12.3.2 Install Node.js (LTS)

```
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt install -y nodejs
```

12.3.3 Install PM2 Globally

```
sudo npm install -g pm2
```

12.4 Environment Configuration

Create environment file:

```
nano /var/www/jackpotx/.env
```

Use production-level settings:

```
NODE_ENV=production
PORT=3004
HOST=0.0.0.0

DB_HOST=xxx.xxx.xxx.xxx
DB_USER=jackpotx_user
DB_PASS=strong_password
DB_NAME=jackpotx_production

JWT_ACCESS_SECRET=...
JWT_REFRESH_SECRET=...

REDIS_HOST=localhost
REDIS_PORT=6379

SWAGGER_PASSWORD=xxxx

ENABLE_PROVIDER_CALLBACKS=true
```

12.5 Building & Deploying Backend

Clone Repository

```
git clone <repo-url> backend.jackpotx.net
cd backend.jackpotx.net
```

Install Dependencies

```
npm install
```

Build TypeScript

```
npm run build
```

Start with PM2

```
pm2 start dist/index.js --name jackpotx && pm2 save
```

Auto-Start on Boot

```
pm2 startup
```

12.6 NGINX Reverse Proxy Configuration

Create site configuration

```
nano /etc/nginx/sites-available/jackpotx
```

Paste:

```
server {
    listen 80;
    server_name backend.jackpotx.net;

    location / {
        proxy_pass http://127.0.0.1:3004;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

Enable site

```
ln -s /etc/nginx/sites-available/jackpotx /etc/nginx/sites-enabled/
nginx -t
systemctl restart nginx
```

12.7 SSL / TLS Configuration

Install Certbot

```
apt install certbot python3-certbot-nginx
```

Enable HTTPS

```
certbot --nginx -d backend.jackpotx.net
```

Cert renews automatically.

12.8 File Permissions & Hardening

Set proper file ownership

```
chown -R www-data:www-data /var/www/jackpotx
chmod -R 755 /var/www/jackpotx
```

Firewall Setup

```
ufw allow OpenSSH
ufw allow 'Nginx Full'
ufw enable
```

12.9 Provider Callback Routing

Most providers require:

```
https://backend.jackpotx.net/api/provider-callback/<provider>
```

Requirements:

- allowlist IPs from provider
- disable caching
- log every callback

- never require JWT
- verify signature field

12.10 Cron Jobs & Scheduled Tasks

JackpotX uses **node-cron** for:

- RTP calculation
- Jackpot increment
- Bonus expiration cleanup
- Daily reports
- Tournament settlement

PM2 handles cron tasks with separate processes:

```
pm2 start dist/cron.js --name jackpotx-cron
pm2 save
```

12.11 Horizontal Scaling (Multi-Node)

If traffic grows, scale using PM2 cluster mode:

```
pm2 start dist/index.js -i max
```

Or with specific count:

```
pm2 start dist/index.js -i 4
```

Redis Pub/Sub ensures WebSocket sync across instances.

12.12 Load Balancing

Layer 7 LB Options

- NGINX
- HAProxy
- Traefik
- AWS ALB

Used for:

- distributing traffic
- health checks
- SSL termination
- rate limiting
- advanced routing

Optional sticky sessions for WebSocket.

12.13 Docker Deployment (Recommended)

docker-compose.yml example:

```
version: '3.8'

services:
  backend:
    build: .
    ports:
      - "3004:3004"
    environment:
      NODE_ENV: production
      DB_HOST: db
      REDIS_HOST: redis
    depends_on:
      - db
      - redis

  db:
    image: postgres:16
    restart: always
    environment:
      POSTGRES_DB: jackpotx
      POSTGRES_USER: jackpotx_user
      POSTGRES_PASSWORD: strongpass
    volumes:
      - dbdata:/var/lib/postgresql/data

  redis:
    image: redis:6
    restart: always

volumes:
  dbdata:
```

Start:

```
docker-compose up -d
```

12.14 Monitoring & Observability

Critical for iGaming operations.

Tools Supported

- **Grafana + Prometheus** (recommended)
- **Elastic Stack (ELK)**
- **Graylog**
- **Datadog**
- **Sentry for error monitoring**

Metrics Monitored

- CPU, RAM
- DB connections
- Query duration
- API latency
- WebSocket traffic
- Provider callback failures
- Deposit / withdrawal errors
- Jackpot increments
- Active users

12.15 Logging

Logs stored in:

```
/var/log/jackpotx/backend.log
/var/log/jackpotx/error.log
```

Format:

```
[TIMESTAMP] [LEVEL] [ROUTE] [USER] [CORRELATION-ID] message
```

12.16 Backups

Daily backups are mandatory.

PostgreSQL

```
pg_dump jackpotx_production > backup_$(date +%F).sql
```

MongoDB

```
mongodump --out /backups/$(date +%F)
```

Redis

```
RDB snapshot.
```

Offsite Storage

- S3
- Google Cloud Storage
- Azure Blob

12.17 Disaster Recovery (DR)

Failover plan:

1. Restore latest DB backup
2. Rebuild Docker or PM2 process
3. Resync Redis cache (optional)
4. Re-enable provider callbacks
5. Run integrity check scripts
6. Trigger internal alerts to risk, finance, and support teams

DR time: **< 30 minutes** for medium-size operators.

12.18 Zero-Downtime Deployment

Method: Blue/Green

- Deploy new instance
- Validate health
- Switch traffic gradually

Method: Rolling Restart (PM2)

```
pm2 reload jackpotx
```

Method: Hot Reload WebSocket

Redis pub/sub keeps clients alive during reload.

12.19 Compliance (iGaming Infrastructure)

12.19.1 Data Centers

Must have:

- ISO 27001
- SOC2 Type II
- GDPR-compliant storage

12.19.2 Auditability

Logs kept:

- admin actions
- financial adjustments
- callback errors
- unusual withdrawal patterns

12.19.3 Separation of Environments

Strict separation:

- dev
- staging
- production

12.20 Summary

The JackpotX deployment architecture ensures:

- high availability
- secure infrastructure
- fast, zero-downtime updates
- scalable design
- hardened environment
- complete observability
- reliable provider integrations
- regulatory compliance

This chapter provides everything required to deploy, scale, monitor, and maintain the JackpotX platform in a professional production environment.