# Marketplace Booking – Part 1: Design & Architecture (NestJS + Stripe)

Goal: real‑world marketplace where clients book services with professionals; implementable in NestJS. This brief focuses on clear assumptions, high‑level architecture, database design, critical flows, and key considerations.

## 1) Assumptions

- Actors: Client (customer), Pro (service provider), Admin.

- Services: Each Pro offers one or more services with fixed duration and price; optional add‑ons.

- Time model: Pros expose availability windows; bookings occupy a time slot (start time + duration). Timezone stored per Pro; all persisted as UTC.

- Payments: Stripe Payment Intents with card by default; marketplace takes platform fee via Connect (destination charges/transfer). Refunds/voids via Stripe.

- Booking policy: A booking is tentatively PENDING until payment succeeds; then becomes CONFIRMED. Holds expire if payment not completed within X minutes.

- Concurrency: High contention on popular slots; must prevent double booking across instances.

- Scale: v1 single region, stateless app behind LB, Postgres primary + read replica; Redis for caching/locks; S3-compatible storage for assets.

- Notifications: Email + push/webhooks for confirmation/updates.

## 2) High‑Level Architecture (NestJS‑centric)

Pattern: Modular monolith with clear bounded contexts (Users, Pros, Catalog, Availability, Booking, Payments, Notifications). Event‑driven inside app via domain events; external via webhooks/queues.

[Client Apps]  
 | REST/GraphQL + Webhooks + WebSockets (for status updates)  
[API Gateway / Ingress]  
 |--> NestJS App (Stateless)  
 Modules:  
 - Auth (JWT, RBAC)  
 - Users & Pros  
 - Catalog (Services, Pricing)  
 - Availability (Calendars, Timezones)  
 - Booking (Slots, Reservations)  
 - Payments (Stripe)  
 - Notifications (Email/SMS/Push)  
 - Admin (reports, refunds)  
 Infra Adapters:  
 - PostgreSQL (TypeORM/Prisma)  
 - Redis (cache + locks)  
 - Stripe SDK + Webhook listener  
 - Message Bus (BullMQ/RabbitMQ optional)  
 - Observability (OpenTelemetry)  
  
[PostgreSQL] [Redis] [Stripe] [Object Storage]

Why this architecture: Keeps deployment simple (single codebase) while enforcing module boundaries. Evented design decouples booking/payment concerns. Redis gives fast reads and safe distributed locks; Postgres ensures transactional integrity.

## 3) Database Design (PostgreSQL)

Prefer Prisma or TypeORM with explicit unique constraints + partial indexes. Times are UTC; timestamptz types.

- **Users**: id (pk), email (uniq), phone (index), role ENUM('client','pro','admin'), created\_at

- **Pros**: id (pk, fk users.id), display\_name, timezone (IANA), stripe\_account\_id (uniq, nullable)

- **Services**: id (pk), pro\_id (fk), title, description, duration\_min, price\_cents, currency, active BOOL index

- **Availability\_windows**: id (pk), pro\_id (fk), weekday, start\_min, end\_min

- **Time\_off**: id (pk), pro\_id (fk), start\_ts, end\_ts

- **Bookings**: id (pk), client\_id, pro\_id, service\_id, start\_ts, end\_ts, status ENUM, price\_cents, payment\_intent\_id (uniq nullable), idempotency\_key (uniq), created\_at, updated\_at

- **Payments**: id (pk), booking\_id (uniq fk), payment\_intent\_id (uniq), charge\_id (index), status ENUM, amount\_cents, application\_fee\_cents, currency, raw JSONB

Indexes/constraints (critical): UNIQUE (pro\_id, start\_ts, end\_ts) WHERE status IN ('PENDING','CONFIRMED'); exclusion constraint on tstzrange; idempotency key unique; services(pro\_id, active) index; bookings(pro\_id, start\_ts) index.

## 4) Critical Flows

### A) Creating a Booking

1. Validate slot against availability.

2. Acquire slot lock in Redis or via DB transaction with exclusion constraint.

3. Create booking with PENDING status and idempotency key.

4. Create Stripe PaymentIntent with amount, destination charge, metadata.

5. Return client secret; client confirms payment.

6. Auto-expire pending bookings after timeout.

### B) Confirming a Stripe Payment

1. Stripe webhook received and verified.

2. Idempotent upsert on payment\_intent\_id.

3. If booking PENDING and slot valid → set CONFIRMED, persist payment record.

4. Handle canceled/failed intents by expiring booking.

## 5) Key Considerations

- Double-booking prevention: exclusion constraint + Redis lock + availability checks.

- Idempotency: client-provided idempotency key, unique constraint, Stripe idempotency header.

- Monitoring: metrics, tracing, structured logging, alerts.

- Security: JWT auth, input validation, secret management, webhook signature verification, PCI compliance, rate limiting, data privacy.