**SmartRentals**

**1) Download the starter backend (FastAPI + SQLite)**

I’ve packaged a clean, production-minded scaffold:

**What’s inside**

* Products, InventoryItems, Customers, Orders, Reservations, Payments
* Availability conflict checks (prevents double booking)
* Basic order lifecycle: pending → confirmed → returned
* Local payment records (manual verification flow)
* Auto-generated API docs at /docs

**Run locally**

unzip smartrentals\_mvp.zip

cd smartrentals\_mvp

pip install -r requirements.txt

uvicorn app.main:app --reload

# Docs: http://127.0.0.1:8000/docs

**2) Target Architecture (scales from MVP to production)**

**Frontend**

* Next.js (React), Tailwind, shadcn/ui
* Admin portal + Customer storefront
* Components: Product catalog, availability calendar, cart, checkout

**Backend**

* FastAPI (Python), Pydantic, SQLAlchemy
* Postgres (prod), SQLite (dev)
* Redis (caching, background jobs)
* Celery / RQ (notifications, report generation, availability recompute)

**Core Services**

* Catalog Service (products, categories, specs)
* Inventory Service (serialised items, locations, maintenance)
* Booking Engine (availability search, conflict resolution)
* Orders & Payments (invoices, deposits, refunds, ledger)
* Users & Roles (RBAC for admin, staff, finance)
* Reporting & Analytics (utilization, revenue, asset ROI)

**Storage & Infra**

* Object storage (S3/Spaces) for images/docs
* Deployment: Docker + Fly.io/Render/AWS
* Observability: Prometheus + Grafana, Sentry
* Security: JWT auth, role-based permissions, audit logs, rate limiting, per-tenant isolation

**3) Data Model (initial tables)**

* products(id, name, description, daily\_rate, sku)
* inventory\_items(id, product\_id, label, location, active)
* customers(id, name, email, phone)
* orders(id, customer\_id, status, created\_at)
* reservations(id, order\_id, inventory\_item\_id, start\_date, end\_date)
* payments(id, order\_id, method, amount, reference, received\_at)

MVP enforces **no overlapping reservations** per inventory item.

**4) Key API endpoints (included in the ZIP)**

**Products**

* POST /products (create)
* GET /products (list)

**Inventory**

* POST /inventory (add item)
* GET /inventory (list items)

**Customers**

* POST /customers (create)
* GET /customers (list)

**Orders**

* POST /orders
  + includes reservation lines with inventory\_item\_id, start\_date, end\_date
  + conflicts return **409** if item unavailable
* POST /orders/{order\_id}/status/{new\_status}
* GET /orders

**Payments**

* POST /payments (record local payments)

**5) Immediate roadmap (4–6 weeks)**

**Week 1: MVP polish**

* Add **JWT auth**, password reset flow, RBAC (admin, staff, viewer)
* Switch to **Postgres**, add Dockerfile/docker-compose
* Seed scripts for products & inventory

**Week 2: Pricing & deposits**

* Price rules: hourly/daily/weekly rates, weekend rules, coupons
* Deposits & refunds; Finance reports (CSV export)

**Week 3: Availability & search**

* Calendar availability view (per item & per product)
* Hold windows / grace periods for unpaid orders
* Overbooking prevention finalized via DB constraints + transactions

**Week 4: Payments**

* Local payments module (Orange Money, AfriMoney, bank transfer) with **admin verification flow**
* Webhook-ready interface to drop in processors later (Vult, Stripe, etc.)

**Week 5: Management**

* Maintenance schedules, service logs, downtime
* Attachments: upload insurance/inspection docs

**Week 6: Multi-tenant prep**

* Tenant-aware schema (org\_id on all tables)
* White-label storefront & theming
* Audit logs, soft delete, backups & restore

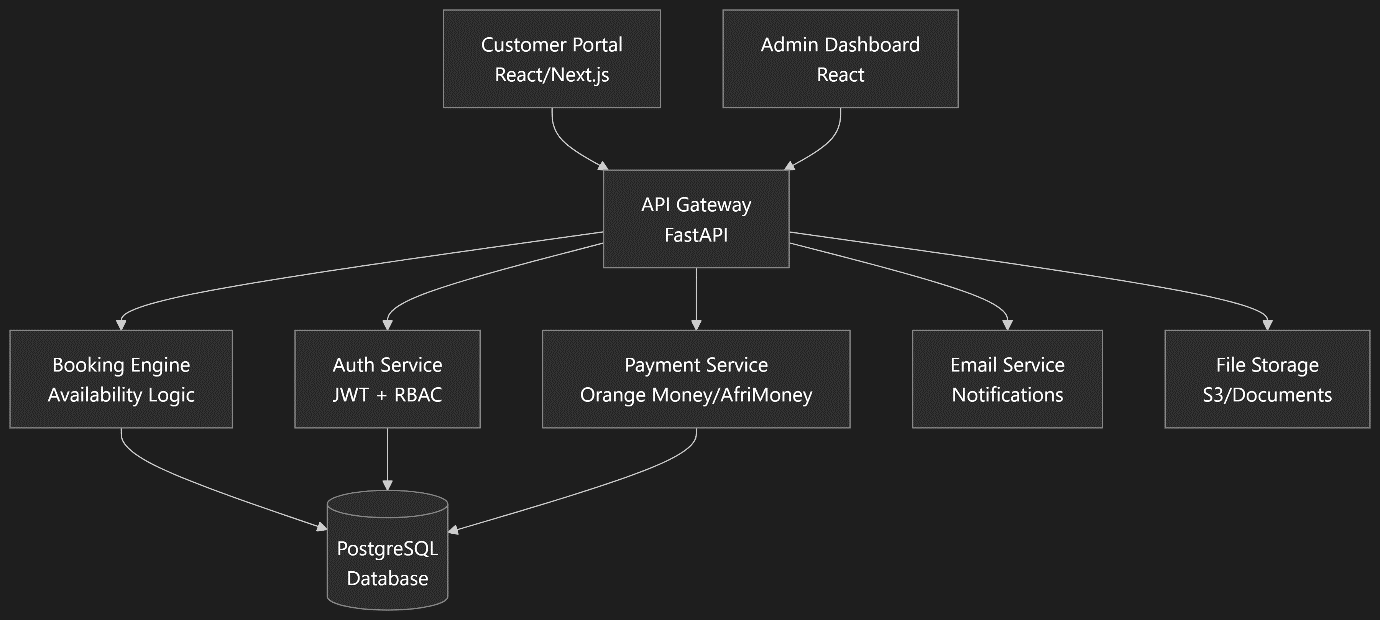
**6) Migration plan (from Booqable to SmartRentals)**

1. **Export** from Booqable: products, inventory, customers, current orders
2. **Transform** CSVs to SmartRentals format (mapping sheet)
3. **Import** via admin scripts / endpoints
4. Run in **parallel** for 1 week (shadow mode)
5. Switch DNS to SmartRentals storefront; archive Booqable for 30 days

**7) Security & Compliance essentials**

* Enforce **RBAC**, hashed passwords, MFA for admins
* Per-tenant data isolation, secrets vault (env vars)
* Backups + PITR for Postgres; S3 versioning for uploads
* PII minimization, audit trails, access logs

Mermaid of Rental System



Breakdown of Each Component & Its Connections

**1. Customer Portal (React/Next.js)**

* **What it is:** The public-facing website. This is what your customers use to browse your rental items, check availability, make bookings, and manage their accounts.
* **Connections:** It connects primarily to the **API Gateway**. Every time a user loads a page, searches for items, or submits a booking form, it sends an HTTP request (e.g., GET, POST) to the Gateway.

**2. Admin Dashboard (React)**

* **What it is:** A private, secure web application for your internal team. Admins use it to manage inventory, view all bookings, update order statuses, process refunds, and generate reports.
* **Connections:** Just like the Customer Portal, it connects to the **API Gateway**. Its requests will include special admin-only endpoints that the Customer Portal cannot access (enforced by the Auth Service).

**3. API Gateway (FastAPI)**

* **What it is:** The front door and traffic cop for your entire backend. It acts as a single entry point for all requests from both the Customer Portal and Admin Dashboard.
* **Key Responsibilities:**
  + **Routing:** It receives a request (e.g., /api/bookings) and routes it to the correct underlying service (e.g., the Booking Engine).
  + **Authentication:** It delegates the task of checking JWT tokens to the **Auth Service**.
  + **Aggregation:** It might fetch data from multiple services to compose a single response for the frontend.
* **Connections:** It is connected to **every other service**. It calls the **Auth Service** first to validate every request. Then, based on the request, it calls the **Booking Engine**, **Payment Service**, etc.

**4. Auth Service (JWT + RBAC)**

* **What it is:** The central security system for your application.
* **Key Responsibilities:**
  + **Authentication (JWT):** Verifies usernames and passwords. Issues and validates JSON Web Tokens (JWT) that prove a user's identity.
  + **Authorization (RBAC):** Uses Role-Based Access Control (e.g., 'Customer', 'Admin', 'Manager') to check if an authenticated user has permission to perform a specific action (e.g., an Admin can delete a product, a Customer cannot).
* **Connections:** The **API Gateway** calls it for every request. It connects to the **PostgreSQL Database** to validate user credentials and check user roles.

**5. Booking Engine (Availability Logic)**

* **What it is:** The brain of your rental business. This service contains the core logic that determines if an item is available for a given date range.
* **Key Responsibilities:** It prevents double-booking by checking existing reservations in the database before confirming a new booking.
* **Connections:** It is called by the **API Gateway**. It reads from and writes to the **PostgreSQL Database** constantly to check and create reservations.

**6. Payment Service (Orange Money/AfriMoney)**

* **What it is:** This service handles all financial transactions.
* **Key Responsibilities:** It integrates with third-party payment providers (Orange Money, AfriMoney) to securely process payments, refunds, and potentially handle payouts.
* **Connections:** It is called by the **API Gateway** when a user proceeds to checkout. It connects to the external payment gateways' APIs. It records transaction success/failure details in the **PostgreSQL Database**.

**7. Email Service (Notifications)**

* **What it is:** A dedicated service for sending emails.
* **Key Responsibilities:** It sends transactional emails like booking confirmations, payment receipts, password resets, and availability notifications.
* **Connections:** It is called by other services (via the **API Gateway** or a message queue). For example, after the **Payment Service** confirms a successful payment, it would ask the **Email Service** to send a confirmation email. It connects to an external email delivery provider (e.g., SendGrid, Mailgun, SMTP server).

**8. File Storage (S3/Documents)**

* **What it is:** A service for managing uploads and downloads of files.
* **Key Responsibilities:** It handles storing product images, user signed contracts (PDFs), insurance documents, etc. It likely uses Amazon S3 or a similar cloud storage service.
* **Connections:** It is called by the **API Gateway** when a user or admin uploads a file. It stores the actual file in S3 and saves the file's metadata (e.g., URL, filename) in the **PostgreSQL Database**.

**9. PostgreSQL Database**

* **What it is:** The single source of truth for all structured data.
* **What it stores:** Everything! Users, products, bookings, orders, payments, categories, roles, file metadata.
* **Connections:** It is the central data hub. Nearly every other service (**Auth, Booking Engine, Payment Service**, etc.) connects to it to persist and retrieve data. The **API Gateway** typically does *not* connect to the database directly; it goes through the other services.

Summary of Data Flow (A User Makes a Booking):

1. **Customer Portal** sends a POST /api/bookings request to the **API Gateway**.
2. **API Gateway** asks the **Auth Service**: "Is this user's JWT token valid?"
3. The **Booking Engine** checks **PostgreSQL** for availability and creates a reservation.
4. The **API Gateway** calls the **Payment Service** to process the payment.
5. The **Payment Service** talks to Orange Money and records the transaction in **PostgreSQL**.
6. The **Payment Service** triggers the **Email Service** to send a confirmation email.
7. The **API Gateway** returns a "Booking Confirmed!" response to the **Customer Porta**