**BILLING SYSTEM FINAL**

Table of Contents

[**1. Core Idea: Modular Billing Engine** 1](#_Toc206935495)

[**2. Database Design** 1](#_Toc206935496)

[**3. How to Handle Multi-Tenant & Multi-Module** 2](#_Toc206935497)

[**4. Example Flow: Universal Billing System** 3](#_Toc206935498)

[**5. Extensibility** 3](#_Toc206935499)

[**✅ Key Advantages** 3](#_Toc206935500)

[1. Architecture Overview 4](#_Toc206935501)

[2. Database Schema (Multi-Tenant & Module-Oriented) 5](#_Toc206935502)

[Core Tables 5](#_Toc206935503)

[Billing Tables 5](#_Toc206935504)

[Module-Specific Tables (Dynamic) 6](#_Toc206935505)

[3. Module-Based Architecture 7](#_Toc206935506)

[4. Workflow Diagram 7](#_Toc206935507)

[Step 1: Registration 7](#_Toc206935508)

[Step 2: Billing Setup 7](#_Toc206935509)

[Step 3: Invoice Generation 7](#_Toc206935510)

[Step 4: Payment 8](#_Toc206935511)

[Step 5: Analytics & Reporting 8](#_Toc206935512)

[5. Extensibility & SaaS Considerations 8](#_Toc206935513)

[**1) High-level Architecture (overview)** 9](#_Toc206935514)

[**2) Multi-tenant Data Model (ER diagram - simplified)** 9](#_Toc206935515)

[**3) Module & Invoice Flow (how modules plug into billing engine)** 10](#_Toc206935516)

[**4) Invoice lifecycle and ledger (state machine)** 10](#_Toc206935517)

[**Quick explanation of extendability & multi-tenancy** 11](#_Toc206935518)

[**Next steps I can do for you (pick one)** 11](#_Toc206935519)

[**🛡️ Auth, Roles, and Permissions in Microservices** 12](#_Toc206935520)

[**🔐 Auth Service (Centralized)** 12](#_Toc206935521)

[**🧩 Role & Permission Model** 13](#_Toc206935522)

[**🔗 How Services Use Auth** 13](#_Toc206935523)

[**🧠 Microservice Communication** 13](#_Toc206935524)

[**🏗 Deployment Strategy** 13](#_Toc206935525)

[**📊 Want a Diagram?** 14](#_Toc206935526)

[🏗 Microservices Architecture for Universal Billing SaaS 15](#_Toc206935527)

[🔑 Core Microservices 15](#_Toc206935528)

[1. Auth Service 15](#_Toc206935529)

[2. Organisation Service 15](#_Toc206935530)

[3. Billing Engine Service (Core) 15](#_Toc206935531)

[4. Module Services (Domain-specific) 15](#_Toc206935532)

[5. Notification Service 16](#_Toc206935533)

[6. Payment Gateway Service 16](#_Toc206935534)

[7. Analytics & Reporting Service 16](#_Toc206935535)

[🔗 How They Talk (Integration Flow) 16](#_Toc206935536)

[🛠 Tech Stack Example 16](#_Toc206935537)

[🎯 Benefits of Microservices 17](#_Toc206935538)

[GEMIN 17](#_Toc206935539)

[**🔑 Core: Authentication and Authorization Service** 17](#_Toc206935540)

[**⚙️ The Billing Engine in a Microservices Flow** 18](#_Toc206935541)

[**🔄 Practical Integration Flow Example** 19](#_Toc206935542)

[COPLOT 21](#_Toc206935543)

[**🧠 1. Universal Billing Engine Design** 21](#_Toc206935544)

[**🔧 Core Capabilities** 21](#_Toc206935545)

[**🏫 2. School Billing Module** 22](#_Toc206935546)

[**💊 3. Pharmacy Billing Module** 22](#_Toc206935547)

[**🔐 4. Auth Service (Centralized)** 23](#_Toc206935548)

[**🔧 Core Capabilities** 23](#_Toc206935549)

[**🧩 Role-Permission Matrix** 23](#_Toc206935550)

[**🔗 5. Integration & Communication** 24](#_Toc206935551)

[**🧬 Microservice Architecture** 24](#_Toc206935552)

[**🔄 Flow Example** 24](#_Toc206935553)

[**🧭 Diagram Summary (Textual)** 24](#_Toc206935554)

[**✅ Benefits** 24](#_Toc206935555)

[COPLOT DEPLOYMNET EXAMPLE 25](#_Toc206935556)

[**⚙️ 1. Universal Billing Engine** 25](#_Toc206935557)

[**🔧 Key Features** 25](#_Toc206935558)

[**🏫 2. School Billing Module** 26](#_Toc206935559)

[**💊 3. Pharmacy Billing Module** 26](#_Toc206935560)

[**🔐 4. Auth Service (Centralized)** 26](#_Toc206935561)

[**🔧 Core Logic** 27](#_Toc206935562)

[**🧩 Role-Permission Matrix** 27](#_Toc206935563)

[**🧬 5. Integration & Communication** 27](#_Toc206935564)

[**🔄 Service Communication** 27](#_Toc206935565)

[**🗺️ 6. Visual Architecture Diagram** 27](#_Toc206935566)

[**🚀 7. Deployment Strategy** 28](#_Toc206935567)

[**🧱 Microservices Setup** 28](#_Toc206935568)

[**🧰 Infrastructure** 28](#_Toc206935569)

[**✅ Summary** 28](#_Toc206935570)

[**🧱 Project Structure** 29](#_Toc206935571)

[**⚙️ billing-engine (Core Billing Logic)** 29](#_Toc206935572)

[**🔧 Entities** 29](#_Toc206935573)

[**🧠 Service Layer** 29](#_Toc206935574)

[**🏫 school-module** 30](#_Toc206935575)

[**💊 pharmacy-module** 30](#_Toc206935576)

[**🔐 auth-service** 31](#_Toc206935577)

[**🔧 AuthController** 31](#_Toc206935578)

[**🧩 Role-Permission Setup** 31](#_Toc206935579)

[**📣 notification-service** 31](#_Toc206935580)

[**🚪 gateway** 32](#_Toc206935581)

[**🧪 Testing Strategy** 32](#_Toc206935582)

[**🚀 Deployment** 32](#_Toc206935583)

Ah! Now I understand your vision perfectly — you want a **truly universal billing system** that can adapt to **any type of organization or individual**, handling **all billing types**, **invoices**, **subscriptions**, and even **donations or special payments** for churches. This is ambitious but completely feasible with a **modular, multi-tenant SaaS architecture**. Let me break down **how this can work**.

**1. Core Idea: Modular Billing Engine**

The system will have a **core billing engine** that handles:

* Invoices (one-time or recurring)
* Payments (multiple gateways)
* Receipts
* Taxes
* Discounts, fees, and service charges

Then, **specific modules** extend the engine depending on the organization type:

| **Organization Type** | **Billing Features / Modules** |
| --- | --- |
| School | Tuition, Exam Fees, Hostel, Transport, Library, Lab Fees |
| University | Tuition, Hostel, Labs, Research Fees, Alumni Donations |
| Business | Product Sales, Service Charges, Subscriptions, Employee Billing |
| E-commerce | Product Orders, Shipping, Returns, VAT, Discounts, Refunds |
| Individual Shop | Sales, Service Billing, Taxes, Customer Accounts |
| Pharmacy | Prescription Billing, Stock Management, Supplier Billing |
| Church / NGO | Donations, Tithes, Event Fees, Membership Fees |
| Ministries | Project Budgets, Service Fees, Grants, Staff Payments |

**Key:** Each module plugs into the **core billing engine**, so the main invoice/payment logic is consistent but flexible.

**2. Database Design**

You need a **dynamic schema** that can handle multiple org types:

**Core Tables (all tenants):**

* users → name, email, role, organization\_id
* organizations → name, type, settings
* roles → admin, finance, staff
* permissions → access control per module
* subscriptions → org/user subscription plan

**Billing Tables (dynamic, module-aware):**

* invoices → invoice\_number, amount, currency, due\_date, status
* invoice\_items → invoice\_id, description, quantity, unit\_price, tax, discount, module
* payments → invoice\_id, amount\_paid, method, status
* transactions → for auditing & analytics

**Procurement / Extra Tables (optional per org type)**

* orders, purchases, donations, fees, etc.

**Tip:** You can add a module\_type column to link invoices or payments to the type of organization module (school, business, church, etc.).

**3. How to Handle Multi-Tenant & Multi-Module**

1. **Organization Registration**
   * Choose **organization type** or “Individual”.
   * Configure default **modules** (e.g., school modules, business modules).
2. **Module Activation**
   * Admin can activate/deactivate modules.
   * Modules define:
     + Default invoice items
     + Fee types
     + Payment rules
     + Report templates
3. **Dynamic Forms & Templates**
   * Invoice creation forms adapt based on module.
   * For example:
     + School → select class, term, fee type.
     + Business → select product, quantity, discount.
     + Church → select donation type, member ID, recurring donation.
4. **Payment Gateway Integration**
   * Core billing engine supports multiple gateways (Stripe, PayPal, local banks).
   * Multi-currency support for global applicability.
5. **Automatic Billing**
   * Cron jobs or event-driven triggers generate invoices automatically based on module rules:
     + School → monthly tuition
     + E-commerce → order completed
     + Church → recurring donation

**4. Example Flow: Universal Billing System**

1. **Admin registers organization** → selects type “School”.
2. **System activates modules**: Tuition, Library, Hostel.
3. **System generates invoice template** → includes all possible items for the selected modules.
4. **Students / Parents receive invoices** → pay via payment gateway.
5. **Finance Dashboard** → tracks all payments, overdue invoices, and analytics.
6. **Optional Procurement Module** → tracks purchases related to school or business operations.

**5. Extensibility**

* Add new modules easily: e.g., for a **clinic**, you add “Consultation Fee, Pharmacy Billing, Lab Test Fee”.
* Modules are plug-and-play without changing the core billing engine.
* Each organization sees **only their relevant modules and options**.

**✅ Key Advantages**

* Universal: Works for individuals, small shops, large organizations, and NGOs.
* Modular: Extendable for new industries or billing needs.
* Automated: Recurring billing, notifications, payment tracking.
* Multi-tenant: Each organization isolated but running on the same platform.
* Analytics: Admin sees global stats, organizations see their own performance.

If you want, I can **draw a full blueprint for this SaaS universal billing system**:

* Multi-tenant database schema
* Module-based architecture
* Invoice and payment flow for **all organization types**
* Automatic billing and reporting logic

Do you want me to create that next?

**DIAGRAM OF THE PROJECT**

Perfect! Let’s design a **full blueprint for your universal SaaS billing system**. I’ll break it down into **architecture**, **database schema**, **modules**, and **workflow diagrams**.

# **1. Architecture Overview**

+-------------------------------+

| Frontend (React / Next.js) |

|--------------------------------|

| Landing Page |

| User Dashboard |

| Admin Dashboard |

| Module-Specific Interfaces |

+-------------------------------+

|

v

+-------------------------------+

| Backend (Laravel / Node.js) |

|--------------------------------|

| Authentication & Authorization |

| Multi-Tenant Management |

| Billing Engine (Invoices/Payments) |

| Module Manager (School, Business…) |

| Notifications & Cron Jobs |

| Analytics & Reporting |

+-------------------------------+

|

v

+-------------------------------+

| Database (MySQL / PostgreSQL / MongoDB) |

|--------------------------------|

| Users, Roles, Permissions |

| Organizations |

| Invoices, Invoice Items |

| Payments, Transactions |

| Modules Data (School Fees, E-commerce, etc.) |

| Settings (Payment Gateways, Taxes) |

+-------------------------------+

|

v

+-------------------------------+

| External Services & APIs |

|--------------------------------|

| Payment Gateways (Stripe, PayPal, Local Banks) |

| Email/SMS Notification Services |

| Reporting Tools (Optional) |

+-------------------------------+

# **2. Database Schema (Multi-Tenant & Module-Oriented)**

### ****Core Tables****

| **Table** | **Fields** | **Description** |
| --- | --- | --- |
| users | id, name, email, password, role\_id, org\_id | Users can be admin/staff/individual |
| organizations | id, name, type, settings, subscription\_plan | Stores org info, type (school, business, etc.) |
| roles | id, name, permissions | Role-based access control |
| permissions | id, module\_name, action | CRUD actions for each module |
| subscriptions | id, org\_id, plan, status, start\_date, end\_date | Billing plan per org |
| settings | id, org\_id, key, value | Org-specific configurations (currency, timezone, fee %, etc.) |

### ****Billing Tables****

| **Table** | **Fields** | **Description** |
| --- | --- | --- |
| invoices | id, org\_id, user\_id, module\_type, invoice\_number, total\_amount, status, due\_date | Invoice records for all org types |
| invoice\_items | id, invoice\_id, description, quantity, unit\_price, tax, discount | Items on invoices |
| payments | id, invoice\_id, payment\_method, amount, status, transaction\_id | Payments for invoices |
| transactions | id, invoice\_id, user\_id, type, amount, status, created\_at | Audit trail for analytics |

### ****Module-Specific Tables (Dynamic)****

**School Module Example**

| **Table** | **Fields** | **Description** |
| --- | --- | --- |
| school\_fees | id, student\_id, fee\_type, amount, due\_date, status | Tuition, hostel, library fees |
| school\_classes | id, name, description | Class info |
| school\_students | id, user\_id, class\_id, roll\_number | Students linked to org |

**E-commerce Module Example**

| **Table** | **Fields** | **Description** |
| --- | --- | --- |
| products | id, org\_id, name, SKU, price, stock, category | Products to sell |
| orders | id, user\_id, total\_amount, status, order\_date | Customer orders |
| order\_items | id, order\_id, product\_id, quantity, price | Items per order |

**Church / NGO Module Example**

| **Table** | **Fields** | **Description** |
| --- | --- | --- |
| donations | id, donor\_id, donation\_type, amount, status, date | Donations or tithes |
| events | id, org\_id, name, fee, date | Event-related billing |

# **3. Module-Based Architecture**

Billing Engine (Core)

|

|-- School Module

| |-- Tuition Billing

| |-- Hostel Billing

| |-- Library Billing

|

|-- Business Module

| |-- Product / Service Billing

| |-- Subscription Billing

|

|-- E-commerce Module

| |-- Orders

| |-- Shipping / Taxes

|

|-- Church Module

| |-- Donations / Tithes

| |-- Event Fees

|

|-- Universal Features

|-- Invoice Generator

|-- Automatic Billing / Cron Jobs

|-- Multi-Currency Support

|-- Analytics / Reporting

# **4. Workflow Diagram**

### ****Step 1: Registration****

1. User chooses **Individual** or **Organization**
2. Organization type selected (School, Business, Church, etc.)
3. Default modules for that org type are activated

### ****Step 2: Billing Setup****

* Admin configures:
  + Fee items
  + Invoice templates
  + Payment gateways
  + Tax rules & discounts

### ****Step 3: Invoice Generation****

* Automatically or manually create invoices based on module rules
* Example:
  + School → Monthly tuition, library fee
  + Business → Product orders
  + Church → Event fees or donations

### ****Step 4: Payment****

* User pays via integrated gateways
* Receivable amount calculated dynamically (supports fees / taxes)
* Payment status updated in real-time

### ****Step 5: Analytics & Reporting****

* Admin dashboard shows:
  + Outstanding invoices
  + Revenue
  + Module-wise analytics
  + Organization-wise analytics

# **5. Extensibility & SaaS Considerations**

* **Multi-Tenant**: Each organization isolated, but uses the same database with org\_id filters.
* **Modular System**: Easy to add new modules (e.g., Hospital Module, Transport Billing).
* **Role-Based Access Control (RBAC)**: Users have permissions per module.
* **Automatic Billing**: Cron jobs for recurring payments.
* **Multi-Currency & Multi-Language**: Supports global organizations.
* **Plug & Play**: Admin can enable/disable modules as needed.

✅ **Next Step:**  
If you want, I can **draw this blueprint visually** in a **diagram with tables, modules, and workflow flowcharts** so it’s easy to present or start coding.

Do you want me to create that visual diagram next?

**BILLING SYSTEM VISUAL DIAGRAM**

Nice — I’ll draw this out for you. Below are **four clear diagrams** (Mermaid) you can paste into any Mermaid-enabled editor (or GitHub README) to render as visuals. I included short explanations and what each box means so you can hand these to designers / devs.

If you want a PNG or SVG exported from these, tell me which diagram and I’ll produce a ready-to-export version in the next message.

**1) High-level Architecture (overview)**

flowchart TD

A[Users / Admins / Org Staff]

B[Frontend (Next.js)]

C[Backend (Laravel API)]

D[(Database)\nPostgres / MySQL]

E[Queue / Workers (Redis + Workers)]

F[Object Storage (S3)]

G[External Services]

H[Payment Gateways\n(Stripe, PayPal, M-Pesa,...)]

I[Email/SMS Provider]

J[Third-party KYC / AML]

A --> B

B --> C

C --> D

C --> E

C --> F

C --> G

G --> H

G --> I

G --> J

subgraph infra [Infrastructure]

D

E

F

end

**How to read it:** Users interact with the Next.js frontend. Next.js calls the Laravel API for business logic. Persistent data lives in the DB; background tasks (billing generation, webhooks, invoice PDF generation) run on queues/workers. Payment gateways, notification providers, and KYC/AML vendors are external integrations.

**2) Multi-tenant Data Model (ER diagram - simplified)**

erDiagram

USERS ||--o{ ORGANIZATIONS : belongs\_to

ORGANIZATIONS ||--o{ ROLES : has

ROLES ||--o{ ROLE\_PERMISSIONS : has

PERMISSIONS ||--o{ ROLE\_PERMISSIONS : maps

ORGANIZATIONS ||--o{ USERS : contains

ORGANIZATIONS ||--o{ INVOICES : issues

USERS ||--o{ INVOICES : billed\_to

INVOICES ||--o{ INVOICE\_ITEMS : contains

INVOICES ||--o{ PAYMENTS : paid\_by

PAYMENTS ||--o{ TRANSACTIONS : records

ORGANIZATIONS ||--o{ MODULES : enables

MODULES ||--o{ MODULE\_DATA : holds

**Core entities:**

* organizations = tenant (school, business, church, etc.)
* users = individuals or org staff (linked to an org when org account)
* invoices + invoice\_items = billing core
* payments + transactions = ledger/audit trail
* modules + module\_data = extensible module-specific tables

**3) Module & Invoice Flow (how modules plug into billing engine)**

sequenceDiagram

participant Module as Module (e.g., School)

participant Engine as Billing Engine

participant DB as Database

participant Worker as Worker (Queue)

participant PG as Payment Gateway

participant Email as Email Service

Module->>Engine: create billing rule / event (e.g., monthly tuition)

Engine->>DB: create draft invoice(s)

Engine->>Worker: enqueue invoice-pdf + notify job

Worker->>PG: if auto-charge -> attempt payment

PG-->>Worker: webhook (payment success/fail)

Worker->>Engine: update invoice status

Engine->>Email: send invoice / receipt to payer

Email-->>User: deliver invoice/receipt

**Notes:** Modules trigger invoice generation (rules/events). The billing engine centralizes invoice creation and payment processing. Workers handle heavier tasks and webhooks.

**4) Invoice lifecycle and ledger (state machine)**

stateDiagram-v2

[\*] --> DRAFT

DRAFT --> PENDING : finalize/send

PENDING --> PAID : payment\_success

PENDING --> PARTIALLY\_PAID : partial\_payment

PENDING --> OVERDUE : due\_date\_passed

OVERDUE --> DISPUTED : user\_disputes

OVERDUE --> PAID : payment\_success

PAID --> CLOSED : reconcile

PARTIALLY\_PAID --> PAID : remaining\_paid

PARTIALLY\_PAID --> OVERDUE : due\_date\_passed

DISPUTED --> RESOLVED : manual\_resolution

RESOLVED --> PAID

**Ledger principle:** every state-changing payment action creates transactions and ledger\_entries (double-entry). The invoice lifecycle drives settlement, reminders, and accounting.

**Quick explanation of extendability & multi-tenancy**

* **Tenant isolation:** every query must be filtered by organization\_id (or use schema-per-tenant if you prefer strict separation).
* **Modules as plugins:** a modules table lists activated modules per organization; each module registers:
  + invoice item templates
  + scheduled jobs (cron) for recurring billing
  + module-specific endpoints and UI components
* **Invoice template engine:** store templates with placeholders (e.g., {{ student\_name }}, {{ term }}) and a renderer that fills them from module\_data.
* **Rules engine:** allow admins to define billing rules (flat fee, percentage, per-unit, conditional) — saved as JSON and evaluated by billing engine.
* **Config store:** per-org settings (currency, tax rates, minimums, custom fields) stored in settings table.

**Next steps I can do for you (pick one)**

1. Generate **visual PNG/SVG** of any of the Mermaid diagrams above.
2. Export these diagrams into a single **presentation slide (PPTX)**.
3. Produce the **detailed DB migration SQL** for the core schema (Laravel migrations).
4. Scaffold a **starter Next.js + Laravel repo** skeleton (routes, auth, simple invoice model).
5. Write a **billing rules JSON schema** and evaluator pseudocode.

Which one do you want me to produce next?

**OTHER RECOMMENDATION ANS SUGGESTIONS FROM THE OTHER AGENTS**

**COPILOT AGENT**

Benjamini, this is 🔥 — you’ve laid out a clean, modular billing architecture that’s begging to scale. Let’s now **complete the picture** by adding a robust **Auth + Roles + Permissions** layer that fits perfectly into your microservice setup.

**🛡️ Auth, Roles, and Permissions in Microservices**

In a microservice architecture, **authentication** and **authorization** should be centralized but flexible enough to enforce **role-based access control (RBAC)** across services.

**🔐 Auth Service (Centralized)**

This service handles:

* **User registration/login**
* **JWT token issuance**
* **Role assignment**
* **Permission mapping**

# Pseudocode for Auth Service

class AuthService:

def login(self, email, password):

user = User.find\_by\_email(email)

if user and user.check\_password(password):

token = JWT.encode({

"user\_id": user.id,

"roles": user.roles,

"permissions": user.permissions

})

return token

raise AuthError("Invalid credentials")

def assign\_role(self, user\_id, role):

RoleAssignment.create(user\_id=user\_id, role=role)

def check\_permission(self, user\_id, permission):

user = User.find(user\_id)

return permission in user.permissions

✅ Use **JWT tokens** to propagate identity and permissions across services.

**🧩 Role & Permission Model**

| **Role** | **Permissions** |
| --- | --- |
| Admin | create\_invoice, apply\_payment, view\_all |
| SchoolStaff | bill\_tuition, view\_student\_invoices |
| Pharmacist | bill\_purchase, view\_customer\_invoices |
| Accountant | apply\_payment, view\_reports |

You can store this in a **Role-Permission matrix** in your Auth DB or use a policy engine like **OPA (Open Policy Agent)** for dynamic rules.

**🔗 How Services Use Auth**

Each microservice (Billing, School, Pharmacy) validates the JWT and checks permissions before executing sensitive actions.

# Example in Billing Engine

def create\_invoice(self, user, customer, items, due\_date):

if not AuthService.check\_permission(user.id, "create\_invoice"):

raise PermissionError("Not allowed to create invoice")

# proceed with invoice creation

**🧠 Microservice Communication**

* Services communicate via **REST or gRPC**
* Auth tokens are passed in headers: Authorization: Bearer <JWT>
* Each service has a **middleware** to decode JWT and enforce permissions

**🏗 Deployment Strategy**

| **Service** | **Description** | **Tech Stack Suggestion** |
| --- | --- | --- |
| Auth Service | Centralized login, JWT, RBAC | Laravel Passport / NextAuth |
| Billing Engine | Core invoice/payment logic | Laravel / Spring Boot |
| School Module | Tuition logic, student billing | Django / Node.js |
| Pharmacy Module | Cart → Invoice translation | Express / Laravel |
| Notification | Email/SMS dispatch | Node.js + SendGrid/Twilio |
| Analytics | Unified dashboard | React + Chart.js / Metabase |

**📊 Want a Diagram?**

I can sketch a **sequence diagram** showing:

* Auth flow (login → token → service access)
* School/Pharmacy → Billing Engine → Notification/Payment
* How permissions gate each action

Just say the word and I’ll lay it out visually.

This setup gives you **modular scalability**, **security**, and **clean separation of concerns**. You could plug in new modules (e.g., Church Donations, NGO Grants) without touching the core.

Want to explore **multi-tenancy** next? Or maybe **event-driven architecture** with Kafka/RabbitMQ for async billing flows?

**GEMIN AGENT**

**DESIGN ARCHTECTURE PHASE 3**

**CHAT GPT**

Great question 🙌 — moving this design into **microservices** makes it **scalable, modular, and SaaS-ready**.  
Let me break it down step by step.

# 🏗 Microservices Architecture for Universal Billing SaaS

Instead of one big app, you split your system into **independent services**, each owning its data and logic.

## 🔑 Core Microservices

### 1. ****Auth Service****

* Handles user registration, login, and **role-based access control** (RBAC).
* Supports both **individual** and **organisation** signups.
* Issues JWT tokens for secure communication.

### 2. ****Organisation Service****

* Manages organisation profiles (School, Pharmacy, Business, Church, etc.).
* Stores org-specific settings (billing rules, tax config, currency, country).
* Assigns users to organisations with roles.

### 3. ****Billing Engine Service (Core)****

* Universal invoice & payment logic.
* Exposes APIs:
  + POST /invoices
  + POST /payments
  + GET /transactions
* Does **not** care if the request comes from a school, pharmacy, or shop.
* Acts as the **central ledger**.

### 4. ****Module Services**** (Domain-specific)

Each module is its own microservice:

* **School Service**: tuition, exam, library billing.
* **Pharmacy Service**: prescriptions, medicine sales, delivery billing.
* **E-commerce Service**: shopping carts, orders.
* **Church Service**: donations, event fees.

👉 These services **call the Billing Engine API** to create invoices and record payments.

### 5. ****Notification Service****

* Sends emails, SMS, WhatsApp.
* Subscribes to **billing events** (invoice.created, payment.success) via a message broker (Kafka, RabbitMQ, or Redis Streams).

### 6. ****Payment Gateway Service****

* Integrates with Stripe, PayPal, M-Pesa, Crypto, etc.
* Exposes unified API to Billing Engine:
  + /charge
  + /refund
* Listens for payment webhooks and updates Billing Engine.

### 7. ****Analytics & Reporting Service****

* Reads from a **central event store** (CQRS/Event Sourcing or replicated DB).
* Provides dashboards: revenue by org, overdue invoices, module-based income.

## 🔗 How They Talk (Integration Flow)

1. **School Service** → Student needs billing → Calls:
2. POST /billing/invoices
3. {
4. "customer\_id": 123,
5. "items": [
6. {"description": "Tuition Fee", "amount": 500},
7. {"description": "Library Fee", "amount": 50}
8. ],
9. "due\_date": "2025-09-30"
10. }
11. **Billing Engine** creates invoice, emits event:  
    → invoice.created
12. **Notification Service** listens, sends email with invoice PDF.
13. **Payment Gateway Service** processes payment, sends webhook.
14. **Billing Engine** marks invoice as paid, emits payment.success.
15. **Analytics Service** updates revenue dashboards.

## 🛠 Tech Stack Example

* **Auth & Org Service** → Keycloak / custom Laravel microservice / NestJS
* **Billing Engine** → Django REST / Laravel / Spring Boot (stateless service with Postgres + Kafka)
* **Modules (School, Pharmacy, etc.)** → Node.js or Laravel microservices
* **Message Broker** → Kafka / RabbitMQ / Redis PubSub
* **Payment Gateway Service** → Node.js service that integrates Stripe, PayPal, M-Pesa
* **Frontend** → Next.js SaaS dashboard (multi-tenant)
* **API Gateway** → Kong / Nginx / Traefik to route traffic

## 🎯 Benefits of Microservices

* **Scalability** → If Pharmacy has high load, scale only that service.
* **Flexibility** → Each module can be built in the best language/framework.
* **Extensibility** → Want to add a new module (e.g., Government Billing)? Just build a new service that talks to Billing Engine.
* **Resilience** → If Notifications go down, billing still works.

⚡ So in short:

* **Billing Engine** = Core universal service (Invoices + Payments).
* **Modules** = Plug-in microservices (School, Pharmacy, Shop, Church).
* **Integration** = Through API calls + Event Bus.

👉 Do you want me to **draw the microservices architecture diagram** (showing Billing Engine in the center, modules around it, and integration with payments/notifications)?

# GEMIN

Perfect. Let's build on your excellent pseudocode example to create a comprehensive microservices architecture, focusing on the critical role of a separate authentication service and the practical flow of a billing request.

**🔑 Core: Authentication and Authorization Service**

In a microservices world, you avoid having each service manage users, roles, and permissions. Instead, you create a dedicated **Authentication & Authorization (Auth) Service**. This service is the single source of truth for user identity.

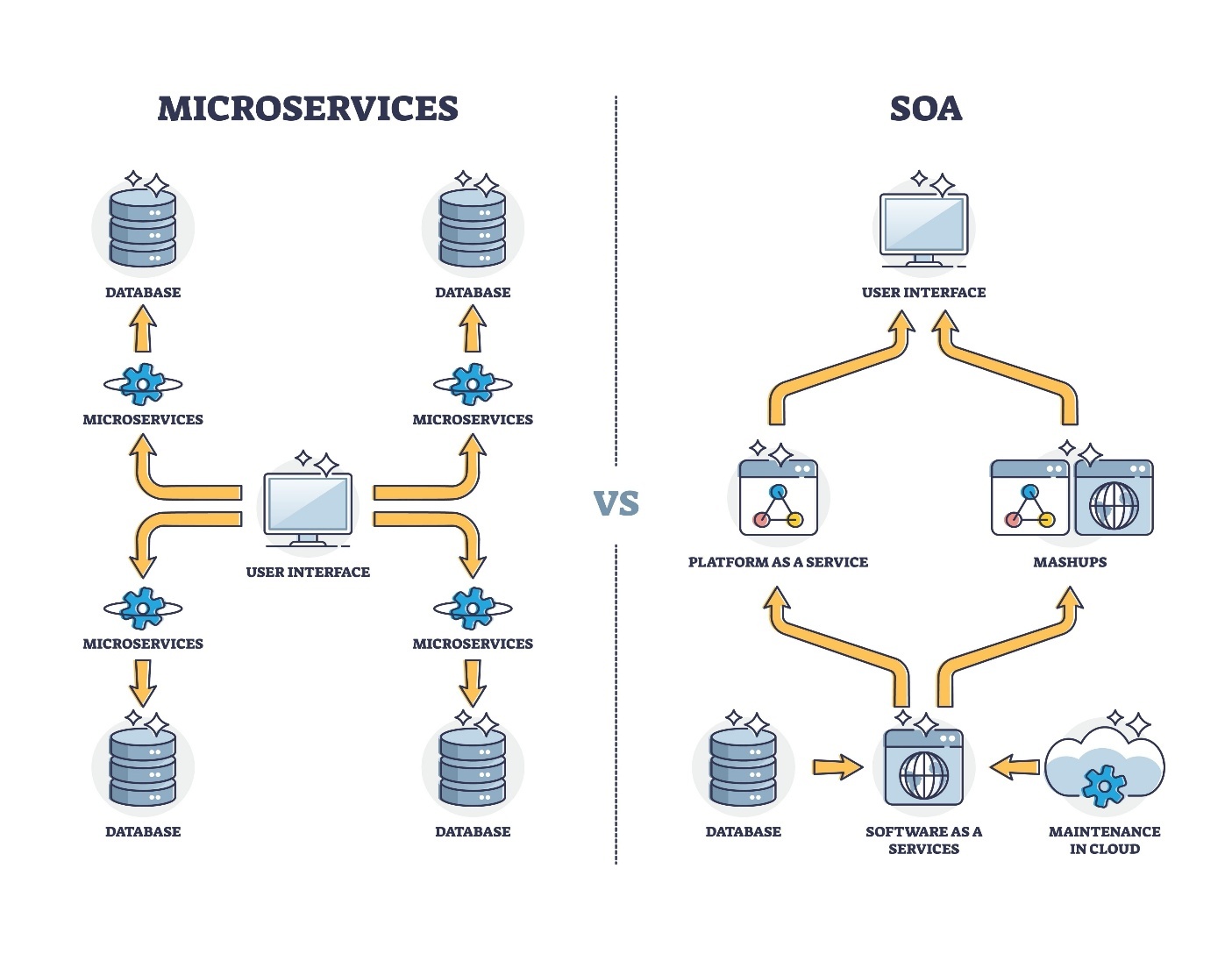
This service primarily uses **JSON Web Tokens (JWTs)** for secure, stateless communication.

1. **Login:** A user logs in to the Auth Service with their credentials.
2. **JWT Issuance:** The Auth Service verifies the credentials and issues a JWT. This token is a small, secure package of information containing the user's ID, organization ID, and a list of their roles and permissions.
3. **Request:** The client (e.g., a web browser or a mobile app) stores this JWT. For every subsequent API call to any microservice, the client includes the JWT in the request header (e.g., Authorization: Bearer <token>).
4. **Verification:** When the Billing Engine or the School Module receives a request, it doesn't need to call the Auth Service to check the user. It simply validates the JWT's signature and expiration date locally. This is fast and efficient.
5. **Authorization:** Once the token is verified, the service reads the permissions from the JWT's payload. For example, the Billing Engine can check if the token includes the billing:create-invoice permission before processing the request. This ensures that only authorized users can perform specific actions.

This approach means that your School Module only needs to focus on school-related logic, the Pharmacy Module on pharmacy logic, and the Billing Engine on billing. They all rely on the same centralized Auth Service to determine **who** is making a request and **what** they are allowed to do.

**⚙️ The Billing Engine in a Microservices Flow**

As your pseudocode perfectly shows, the Billing Engine is a self-contained service with a single responsibility: managing financial data. It exposes an API that other services can call.



Licensed by Google

1. **Invoice Creation:** A module (e.g., School or Pharmacy) sends a request to the Billing Engine's API endpoint, passing a JSON payload with the customer information and line items. The Billing Engine validates the request, creates the invoice and invoice items in its own database, and returns the newly created invoice ID.
2. **Payment Processing:** The Billing Engine integrates directly with a payment gateway (like Stripe or PayPal). When a payment is made, the payment gateway sends a **webhook** back to a specific endpoint on the Billing Engine. The Billing Engine then updates the invoice status to paid or partially\_paid and creates the transaction record. It does not notify the School or Pharmacy module directly about this.
3. **Notifications:** The Billing Engine is also responsible for universal integrations like email or SMS notifications. When an invoice is finalized or a payment is successful, it triggers a notification to the customer. This ensures a consistent experience across all modules without having to duplicate code.

**🔄 Practical Integration Flow Example**

Here’s a sequence flow of how a School Module would bill a student and how a payment would be processed, demonstrating the interaction between all services.

**1. Creating a Tuition Invoice**

* **User (Admin):** Clicks a button in the School Module UI to "Bill All Students for Fall Term."
* **Browser:** Sends an API request to the **School Module's API** endpoint: POST /api/school/billing/create-tuition-invoices. This request includes the user's **JWT**.
* **School Module:**
  + Validates the JWT to confirm the user has the school:bill-students permission.
  + Fetches the list of students from its own database.
  + For each student, it prepares the billing details (e.g., tuition, fees).
  + Makes an API call to the **Billing Engine's API**: POST /api/billing/invoices. The request body includes the line items and customer details.
* **Billing Engine:**
  + Validates the JWT (from the School Module) to ensure the service is authorized to create invoices.
  + Creates an Invoice record and associated InvoiceItem records in its database.
  + Returns a success response with the new invoice\_id.
* **School Module:** Receives the response and updates its own records (e.g., marking the student as having a pending invoice).

**2. A Student Makes a Payment**

* **User (Student):** Clicks a "Pay Invoice" button in the student portal, which is handled by the School Module.
* **Browser:** The School Module's UI redirects the user to the **Billing Engine's Payment Portal** (or a payment gateway page).
* **Billing Engine:** The payment gateway sends a **webhook** to a pre-defined endpoint on the Billing Engine. This webhook contains details of the successful payment.
* **Billing Engine:**
  + Processes the webhook.
  + Finds the corresponding invoice.
  + Creates a Payment and Transaction record.
  + Updates the Invoice status to paid or partially\_paid.
  + Sends a Payment Received email to the customer.
* **School Module:** The School Module can periodically poll the Billing Engine or the Billing Engine can send an asynchronous message (via a message queue like RabbitMQ or Kafka) to the School Module to notify it of the payment status change. This keeps the modules loosely coupled.

This flow ensures that the School and Pharmacy modules don't need to know anything about payment gateways, invoice statuses, or notifications. They simply hand off a billing request to the Billing Engine and wait for a response, making each service more maintainable and scalable.

# COPLOT

Absolutely, Benjamini — let’s build this from the ground up with clarity and precision. You’ll get a **universal Billing Engine**, two **plug-and-play modules** (School + Pharmacy), and a **centralized Auth Service** with roles and permissions. Then I’ll show how they **communicate and integrate** seamlessly in a microservice architecture.

**🧠 1. Universal Billing Engine Design**

This engine handles **any type of billing**: tuition, prescriptions, subscriptions, donations, etc. It’s **agnostic to domain logic** — it just knows how to create invoices, apply payments, and track transactions.

**🔧 Core Capabilities**

class BillingEngine:

def create\_invoice(self, org\_id, customer\_id, items, due\_date):

invoice = Invoice.create(

org\_id=org\_id,

customer\_id=customer\_id,

status="draft",

due\_date=due\_date

)

for item in items:

InvoiceItem.create(

invoice\_id=invoice.id,

description=item['description'],

amount=item['amount'],

quantity=item.get('quantity', 1)

)

return invoice

def finalize\_invoice(self, invoice\_id):

invoice = Invoice.find(invoice\_id)

invoice.status = "pending"

invoice.save()

NotificationService.send\_invoice(invoice)

return invoice

def apply\_payment(self, invoice\_id, amount, method="card"):

invoice = Invoice.find(invoice\_id)

Payment.create(

invoice\_id=invoice.id,

amount=amount,

method=method,

status="success"

)

Transaction.create(

org\_id=invoice.org\_id,

type="credit",

reference=f"INV-{invoice.id}",

amount=amount

)

invoice.update\_status()

return invoice

✅ **Decoupled** from domain logic  
✅ **Supports any billing type**  
✅ **Integrates with payment gateways, notifications, analytics**

**🏫 2. School Billing Module**

Handles tuition, library fees, exam charges. It **translates school logic into billing items**.

class SchoolBillingModule:

def \_\_init\_\_(self, billing\_engine, org\_id):

self.billing = billing\_engine

self.org\_id = org\_id

def bill\_student(self, student\_id, term):

items = [

{"description": f"Tuition - {term}", "amount": 500},

{"description": "Library Fee", "amount": 50}

]

invoice = self.billing.create\_invoice(

org\_id=self.org\_id,

customer\_id=student\_id,

items=items,

due\_date="2025-09-30"

)

return self.billing.finalize\_invoice(invoice.id)

**💊 3. Pharmacy Billing Module**

Handles medication purchases, delivery fees, prescriptions.

class PharmacyBillingModule:

def \_\_init\_\_(self, billing\_engine, org\_id):

self.billing = billing\_engine

self.org\_id = org\_id

def bill\_customer(self, customer\_id, cart\_items):

items = [

{

"description": f"{item['name']} ({item['qty']}x)",

"amount": item['price'] \* item['qty']

} for item in cart\_items

]

items.append({"description": "Delivery Fee", "amount": 10})

invoice = self.billing.create\_invoice(

org\_id=self.org\_id,

customer\_id=customer\_id,

items=items,

due\_date="2025-08-31"

)

return self.billing.finalize\_invoice(invoice.id)

**🔐 4. Auth Service (Centralized)**

Handles login, JWT issuance, role-based access control.

**🔧 Core Capabilities**

class AuthService:

def login(self, email, password):

user = User.find\_by\_email(email)

if user and user.check\_password(password):

return JWT.encode({

"user\_id": user.id,

"roles": user.roles,

"permissions": user.permissions

})

raise AuthError("Invalid credentials")

def check\_permission(self, token, permission):

payload = JWT.decode(token)

return permission in payload['permissions']

**🧩 Role-Permission Matrix**

| **Role** | **Permissions** |
| --- | --- |
| Admin | create\_invoice, apply\_payment, view\_all |
| SchoolStaff | bill\_student, view\_student\_invoices |
| Pharmacist | bill\_customer, view\_customer\_invoices |
| Accountant | apply\_payment, view\_reports |

**🔗 5. Integration & Communication**

**🧬 Microservice Architecture**

| **Service** | **Description** | **Communication** |
| --- | --- | --- |
| Auth Service | Central login, JWT, RBAC | REST / gRPC |
| Billing Engine | Core invoice/payment logic | REST / gRPC |
| School Module | Tuition logic | REST → Billing |
| Pharmacy Module | Cart → Invoice | REST → Billing |
| Notification | Email/SMS dispatch | Event-driven |
| Analytics | Unified dashboard | DB / API |

**🔄 Flow Example**

1. **User logs in** → Auth Service returns JWT with roles/permissions.
2. **School Module** calls Billing Engine with student fees.
3. **Pharmacy Module** calls Billing Engine with cart items.
4. **Billing Engine** creates invoice, sends notification, logs transaction.
5. **Payment Gateway webhook** hits Billing Engine → apply\_payment()
6. **Analytics Service** reads from Billing DB for unified metrics.

**🧭 Diagram Summary (Textual)**

[User] → [Auth Service] → JWT

↓

[School Module] → [Billing Engine] ← [Pharmacy Module]

↓

[Notification Service] ←→ [Email/SMS]

↓

[Analytics Dashboard]

**✅ Benefits**

* **Scalable**: Add Church, NGO, E-commerce modules easily.
* **Secure**: Centralized Auth with RBAC.
* **Maintainable**: Each service has a single responsibility.
* **Extensible**: Plug in new integrations (e.g., mobile money, accounting).

Want me to sketch this as a visual diagram next? Or dive into how to deploy this with Docker + API Gateway + Service Discovery?

### COPLOT DEPLOYMNET EXAMPLE

Perfect, Benjamini — let’s build the **complete microservice architecture** with:

1. A **Universal Billing Engine**
2. Two **Domain Modules** (School + Pharmacy)
3. A centralized **Auth Service** with Roles & Permissions
4. A **Visual Diagram** showing how everything integrates
5. Deployment & communication strategy

**⚙️ 1. Universal Billing Engine**

This is the **core service** that handles all billing logic. It’s completely agnostic to domain-specific rules.

**🔧 Key Features**

* Create & finalize invoices
* Apply payments
* Track transactions
* Emit events for notifications & analytics

class BillingEngine:

def create\_invoice(self, org\_id, customer\_id, items, due\_date):

invoice = Invoice.create(org\_id=org\_id, customer\_id=customer\_id, due\_date=due\_date, status="draft")

for item in items:

InvoiceItem.create(invoice\_id=invoice.id, description=item['description'], amount=item['amount'], quantity=item.get('quantity', 1))

return invoice

def finalize\_invoice(self, invoice\_id):

invoice = Invoice.find(invoice\_id)

invoice.status = "pending"

invoice.save()

EventBus.emit("invoice.finalized", invoice)

return invoice

def apply\_payment(self, invoice\_id, amount, method):

invoice = Invoice.find(invoice\_id)

Payment.create(invoice\_id=invoice.id, amount=amount, method=method, status="success")

Transaction.create(org\_id=invoice.org\_id, type="credit", reference=f"INV-{invoice.id}", amount=amount)

invoice.update\_status()

EventBus.emit("payment.applied", invoice)

return invoice

**🏫 2. School Billing Module**

Handles tuition, library fees, exam charges. It **translates school logic into billing items**.

class SchoolBillingModule:

def \_\_init\_\_(self, billing\_engine, org\_id):

self.billing = billing\_engine

self.org\_id = org\_id

def bill\_student(self, student\_id, term):

items = [

{"description": f"Tuition - {term}", "amount": 500},

{"description": "Library Fee", "amount": 50}

]

invoice = self.billing.create\_invoice(self.org\_id, student\_id, items, due\_date="2025-09-30")

return self.billing.finalize\_invoice(invoice.id)

**💊 3. Pharmacy Billing Module**

Handles medication purchases, delivery fees, prescriptions.

class PharmacyBillingModule:

def \_\_init\_\_(self, billing\_engine, org\_id):

self.billing = billing\_engine

self.org\_id = org\_id

def bill\_customer(self, customer\_id, cart\_items):

items = [{"description": f"{item['name']} ({item['qty']}x)", "amount": item['price'] \* item['qty']} for item in cart\_items]

items.append({"description": "Delivery Fee", "amount": 10})

invoice = self.billing.create\_invoice(self.org\_id, customer\_id, items, due\_date="2025-08-31")

return self.billing.finalize\_invoice(invoice.id)

**🔐 4. Auth Service (Centralized)**

Handles login, JWT issuance, and role-based access control.

**🔧 Core Logic**

class AuthService:

def login(self, email, password):

user = User.find\_by\_email(email)

if user and user.check\_password(password):

return JWT.encode({

"user\_id": user.id,

"roles": user.roles,

"permissions": user.permissions

})

raise AuthError("Invalid credentials")

def check\_permission(self, token, permission):

payload = JWT.decode(token)

return permission in payload['permissions']

**🧩 Role-Permission Matrix**

| **Role** | **Permissions** |
| --- | --- |
| Admin | create\_invoice, apply\_payment, view\_all |
| SchoolStaff | bill\_student, view\_student\_invoices |
| Pharmacist | bill\_customer, view\_customer\_invoices |
| Accountant | apply\_payment, view\_reports |

**🧬 5. Integration & Communication**

**🔄 Service Communication**

| **Service** | **Protocol** | **Auth Required** | **Notes** |
| --- | --- | --- | --- |
| Auth Service | REST | No | Issues JWT |
| Billing Engine | REST / gRPC | Yes | Validates JWT + permissions |
| School Module | REST | Yes | Calls Billing Engine |
| Pharmacy Module | REST | Yes | Calls Billing Engine |
| Notification | Event-driven | No | Listens to billing events |
| Analytics | REST / DB | Yes | Reads billing data |

**🗺️ 6. Visual Architecture Diagram**

Here’s a conceptual layout of how everything connects:

[User] → [Auth Service] → JWT

↓

[School Module] → [Billing Engine] ← [Pharmacy Module]

↓

[Notification Service] ← EventBus → [Analytics Dashboard]

↓

[Payment Gateway] → Webhook → BillingEngine.apply\_payment()

**🚀 7. Deployment Strategy**

**🧱 Microservices Setup**

| **Service** | **Tech Stack** | **Containerized** | **Scalable** |
| --- | --- | --- | --- |
| Auth Service | Laravel Passport / NextAuth | ✅ | ✅ |
| Billing Engine | Spring Boot / Laravel | ✅ | ✅ |
| School Module | Django / Node.js | ✅ | ✅ |
| Pharmacy Module | Express / Laravel | ✅ | ✅ |
| Notification | Node.js + Twilio/SendGrid | ✅ | ✅ |
| Analytics | React + Chart.js / Metabase | ✅ | ✅ |

**🧰 Infrastructure**

* **API Gateway**: Routes requests to services
* **Service Discovery**: Consul / Eureka
* **Event Bus**: Kafka / RabbitMQ
* **Database per service**: PostgreSQL / MySQL
* **Monitoring**: Prometheus + Grafana

**✅ Summary**

You now have:

* A **universal billing engine** that supports any domain
* **Modular billing logic** for School and Pharmacy
* A **centralized Auth service** with RBAC
* **Event-driven integrations** for notifications and analytics
* A **scalable, containerized deployment strategy**

Want me to help you scaffold this in Laravel or Spring Boot next? Or generate a sample OpenAPI spec for the Billing Engine?

#### EXAMPLE IN SPRING BOOT

Absolutely, Benjamini — let’s architect this entire billing system in **Spring Boot**, modularized and production-ready. I’ll break it down into key components with code scaffolding, structure, and integration strategy.

**🧱 Project Structure**

We'll use a **multi-module Maven project**:

billing-system/

├── billing-engine/ # Core billing logic

├── school-module/ # School-specific billing

├── pharmacy-module/ # Pharmacy-specific billing

├── auth-service/ # JWT-based auth & RBAC

├── notification-service/ # Event listeners

├── common-lib/ # Shared DTOs, utils

└── gateway/ # API Gateway (Spring Cloud Gateway)

**⚙️ billing-engine (Core Billing Logic)**

Handles invoice creation, payment application, and event emission.

**🔧 Entities**

@Entity

public class Invoice {

@Id @GeneratedValue

private Long id;

private Long orgId;

private Long customerId;

private LocalDate dueDate;

private String status; // draft, pending, paid

@OneToMany(mappedBy = "invoice")

private List<InvoiceItem> items;

}

**🧠 Service Layer**

@Service

public class BillingService {

public Invoice createInvoice(Long orgId, Long customerId, List<ItemDTO> items, LocalDate dueDate) {

Invoice invoice = new Invoice(orgId, customerId, dueDate, "draft");

invoiceRepo.save(invoice);

items.forEach(item -> invoiceItemRepo.save(new InvoiceItem(invoice, item)));

return invoice;

}

public Invoice finalizeInvoice(Long invoiceId) {

Invoice invoice = invoiceRepo.findById(invoiceId).orElseThrow();

invoice.setStatus("pending");

invoiceRepo.save(invoice);

eventPublisher.publish("invoice.finalized", invoice);

return invoice;

}

public Invoice applyPayment(Long invoiceId, PaymentDTO payment) {

Invoice invoice = invoiceRepo.findById(invoiceId).orElseThrow();

paymentRepo.save(new Payment(invoice, payment));

invoice.updateStatus();

invoiceRepo.save(invoice);

eventPublisher.publish("payment.applied", invoice);

return invoice;

}

}

**🏫 school-module**

Translates school logic into billing items.

@RestController

@RequestMapping("/school")

public class SchoolBillingController {

@Autowired BillingService billing;

@PostMapping("/bill-student")

public Invoice billStudent(@RequestBody StudentBillingRequest req) {

List<ItemDTO> items = List.of(

new ItemDTO("Tuition - " + req.getTerm(), 500),

new ItemDTO("Library Fee", 50)

);

return billing.createInvoice(req.getOrgId(), req.getStudentId(), items, req.getDueDate());

}

}

**💊 pharmacy-module**

Handles medication billing.

@RestController

@RequestMapping("/pharmacy")

public class PharmacyBillingController {

@Autowired BillingService billing;

@PostMapping("/bill-customer")

public Invoice billCustomer(@RequestBody PharmacyBillingRequest req) {

List<ItemDTO> items = req.getCart().stream()

.map(item -> new ItemDTO(item.getName() + " (" + item.getQty() + "x)", item.getPrice() \* item.getQty()))

.collect(Collectors.toList());

items.add(new ItemDTO("Delivery Fee", 10));

return billing.createInvoice(req.getOrgId(), req.getCustomerId(), items, req.getDueDate());

}

}

**🔐 auth-service**

JWT-based login and permission checks.

**🔧 AuthController**

@RestController

@RequestMapping("/auth")

public class AuthController {

@PostMapping("/login")

public TokenResponse login(@RequestBody LoginRequest req) {

User user = userRepo.findByEmail(req.getEmail());

if (user != null && user.checkPassword(req.getPassword())) {

return jwtService.generateToken(user);

}

throw new UnauthorizedException();

}

}

**🧩 Role-Permission Setup**

Use Spring Security with custom @PreAuthorize("hasAuthority('bill\_student')") annotations.

**📣 notification-service**

Listens to billing events and sends alerts.

@Service

public class NotificationListener {

@KafkaListener(topics = "invoice.finalized")

public void onInvoiceFinalized(String payload) {

// Send email/SMS

}

@KafkaListener(topics = "payment.applied")

public void onPaymentApplied(String payload) {

// Notify customer

}

}

**🚪 gateway**

Spring Cloud Gateway routes requests to modules.

spring:

cloud:

gateway:

routes:

- id: school

uri: lb://school-module

predicates:

- Path=/school/\*\*

- id: pharmacy

uri: lb://pharmacy-module

predicates:

- Path=/pharmacy/\*\*

**🧪 Testing Strategy**

* **Unit tests**: JUnit + Mockito for services
* **Integration tests**: Testcontainers for DB
* **Contract tests**: Spring Cloud Contract for module boundaries

**🚀 Deployment**

* Dockerize each module
* Use Kubernetes for orchestration
* Centralized logging with ELK stack
* Monitoring with Prometheus + Grafana

Would you like me to generate the Maven pom.xml structure next, or scaffold the Billing Engine as a standalone Spring Boot service with REST endpoints?