🧾 Project Scope – Node.js Microservices

🧱 Overview

Build a scalable microservices-based backend system where:

• A **default admin** can create new users.

• Each user gets a **dedicated MongoDB database**.

• Each user is assigned services with **slab-based service charges**.

• On user creation, a **wallet is auto-created**.

• Transactions are queued via **RabbitMQ**.

• Wallet rules are enforced (hold, min/max, lean).

• Dummy third-party API is called asynchronously.

• Transaction statuses update through **cron job** execution.

🧩 Microservices Architecture

1. Auth Service

• Admin login with JWT.

• Middleware for role-based access.

2. User Service

• Admin creates users.

• Auto-creation of per-user MongoDB database.

• Wallet initialization for the user.

3. Wallet Service

• Wallet schema includes:

```
{
  userId, balance, hold, minLimit, maxLimit, lean
}
```

• Wallet top-up rules:

• Deduct lean first.

• Remaining goes to balance.

4. Service Charge Service

• Service schema:

```
{
  userId, serviceId, slab: "50_1000_2.5_rupees/1001_5000_5_rupees"
}
```

• Slab logic is parsed dynamically per transaction.

5. Transaction Service

• Accepts transaction requests via queue.

• Transaction schema:

```
{
  amount, serviceCharge, gst, userId, prevBalance,
  updatedBalance, serviceId, status // initiated → awaited → success
}
```

6. Queue Service (RabbitMQ)

• **Producer** adds validated transaction requests to queue.

• **Consumer**:

• Validates wallet.

• Deducts balance.

• Creates transaction (status: initiated).

• Sends request to dummy 3rd-party API.

• Updates status to awaited.

7. Dummy Third-party API

• Simulates banking API.

• Accepts transaction, returns:

{ status: "acknowledged" }

8. Cron Service

• Runs every 1 minute.

• Updates all awaited transactions to success.

9. API Gateway

• Central entry point.

• Routes requests.

• Auth token verification.

• Rate-limiting (optional).

10. Common Module

• Shared utility functions.

• DB connection logic.

• Models, error handling, logger, etc.

📦 Folder Structure

```
microservices-project/
│
├── api-gateway/
├── auth-service/
├── user-service/
├── wallet-service/
├── transaction-service/
├── service-charge-service/
├── queue-service/
├── cron-service/
├── dummy-bank-api/
├── common/
└── docker-compose.yml
```

📁 Root Directory Structure

```
microservices-project/
├── api-gateway/
│   ├── index.js
│   └── routes/
├── auth-service/
│   ├── index.js
│   └── controllers/
├── user-service/
│   ├── index.js
│   └── controllers/
├── wallet-service/
│   ├── index.js
│   └── services/
├── service-charge-service/
│   ├── index.js
│   └── utils/
├── transaction-service/
│   ├── index.js
│   └── models/
├── queue-service/
│   ├── index.js
│   └── consumer.js
├── cron-service/
│   ├── index.js
│   └── cron.js
├── dummy-bank-api/
│   └── index.js
├── common/
│   ├── db.js
│   ├── logger.js
│   └── models/
├── docker-compose.yml
├── .env
└── README.md
```

✅ **Highlights**:

// - `user-service` creates users and auto-creates DB and wallet
// - `wallet-service` enforces hold, lean, min/max rules
// - `transaction-service` creates transaction and status (initiated > awaited)
// - `queue-service` consumes RabbitMQ jobs and hits dummy API
// - `cron-service` updates 'awaited' to 'success'
// - `dummy-bank-api` simulates third-party system
// - `common/` holds reusable DB/model code

🧪 Example Data Flow

🧍 Admin Creates User

• POST /user

```
{
  "name": "Kumar",
```

```
  "email": "123@sparkuptech.in",
  "mobile": "9876567871",
  "userId": "SP10001"
}
```


➡️ Creates DB sparkup_SP10001

➡️ Auto wallet:


```
{
  "userId": "SP10001",
  "balance": 10000,
  "hold": 100,
  "minLimit": 50,
  "maxLimit": 5000,
  "lean": 200
}
```


⚙️ **Assign Service**

• POST /service/assign


```
{
  "userId": "SP10001",
  "serviceId": "SV1001",
  "slabs": "50_1000_2.5_rupees/1001_5000_5_rupees"
}
```


💸 **Make Transaction**

• POST /transaction


```
{
  "userId": "SP10001",
  "amount": 1000,
  "serviceId": "SV1001"
}
```


➡️ Goes to RabbitMQ →

➡️ Wallet is validated →

➡️ Transaction created:


```
{
  "amount": 1000,
  "serviceCharge": 2.5,
  "gst": 0.45,
  "prevBalance": 10000,
  "updatedBalance": 8997.05,
  "status": "awaited"
}
```


➡️ Dummy API is hit →

➡️ Cron (after 1 min):

awaited → success

🔐 Business Rules to Enforce

• Balance available = balance - hold

• Cannot process if:

• amount < minLimit

• amount > maxLimit

• availableBalance < amount

• Top-up logic:

• If lean = 200 and top-up is 1000:

• lean = 0, balance += 800

⚙️ Tech Stack

Component	Technology
Language	Node.js (TypeScript/JS)
Database	MongoDB (per user DB)
Queue	RabbitMQ
API Gateway	Express.js
Auth	JWT
Cron Job	node-cron / agenda
Containerization	Docker + Docker Compose
Monitoring (optional)	Prometheus + Grafana

🎯 Deliverables

• Complete microservices project

• Postman collection

• Docker setup for local dev

• README with setup steps

• Sample data scripts

• Cron job setup

• Test transactions with dummy bank API