TRANSACTION SERVICE

Developed by- Jatin Saini

2024tm93002@wilp.bits-pilani.ac.in

Repository Link: https://github.com/jatin904/scalable-banking.git

Table of Contents

Description

The **Transaction** Service handles all money-movement operations within the Scalable Banking system. It records deposits, withdrawals, and transfers, applies business validations, and ensures reliability via idempotency and RabbitMQ-based events for downstream services (e.g., notifications).

Component	Technology
Language	Node.js (Express.js ES Modules)
Database	PostgreSQL
Messaging	RabbitMQ
Monitoring	Prometheus + Grafana
Containerization	Docker

Component	Technology
Orchestration	Kubernetes (Minikube)
Docs	Swagger / OpenAPI

Responsibilities

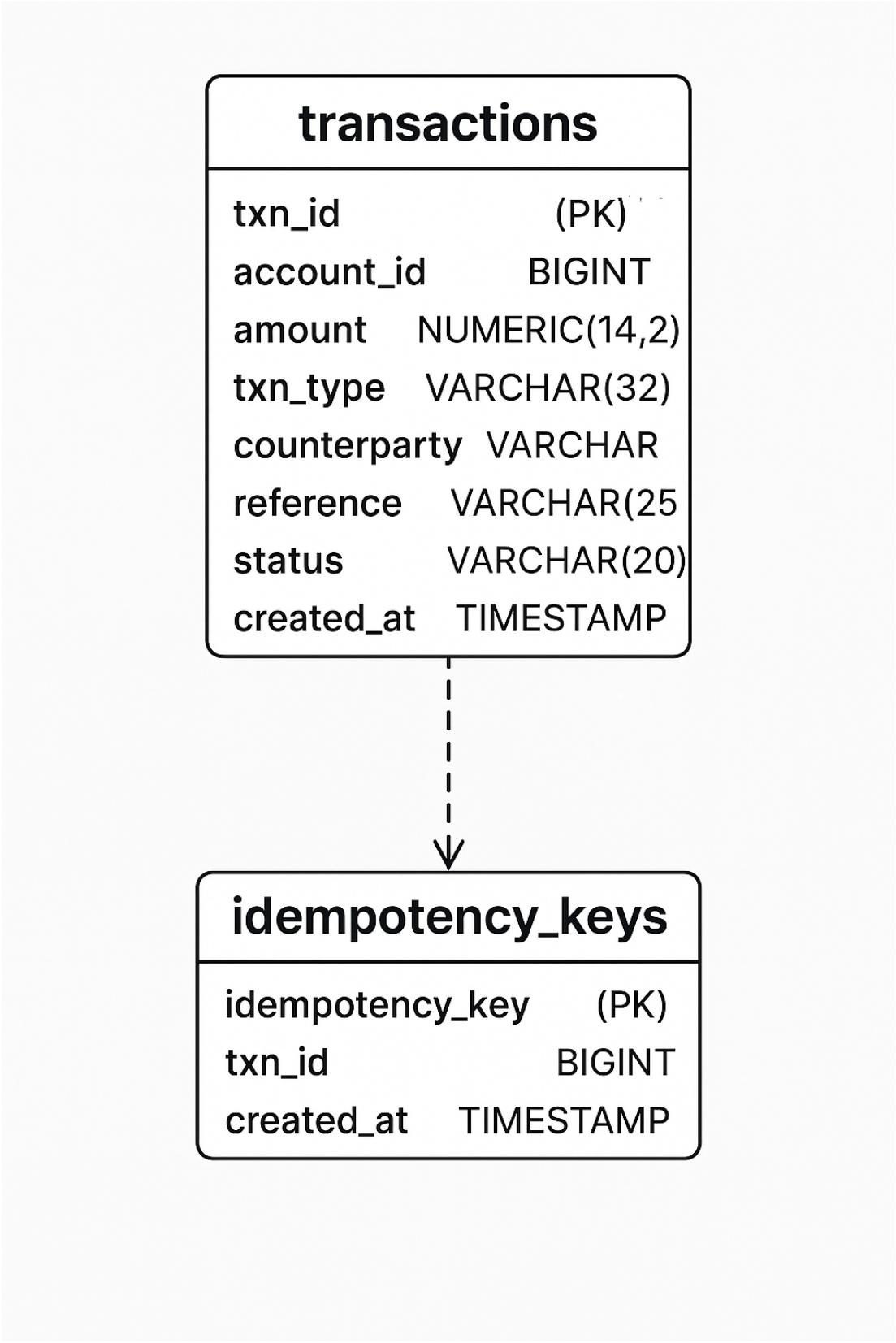
Record and validate Deposit and Transfer transactions
Enforce business rules (daily limits, balance, account status)
Maintain idempotency for safe retries
Publish transaction events (DEPOSIT_CREATED, TRANSFER_COMPLETED)
Expose REST APIs + /metrics endpoint for Prometheus
Log structured JSON with correlation IDs

Transaction Service – ER Diagram

Logical ER structure based on my PostgreSQL schema:

Database: transaction_db contains:

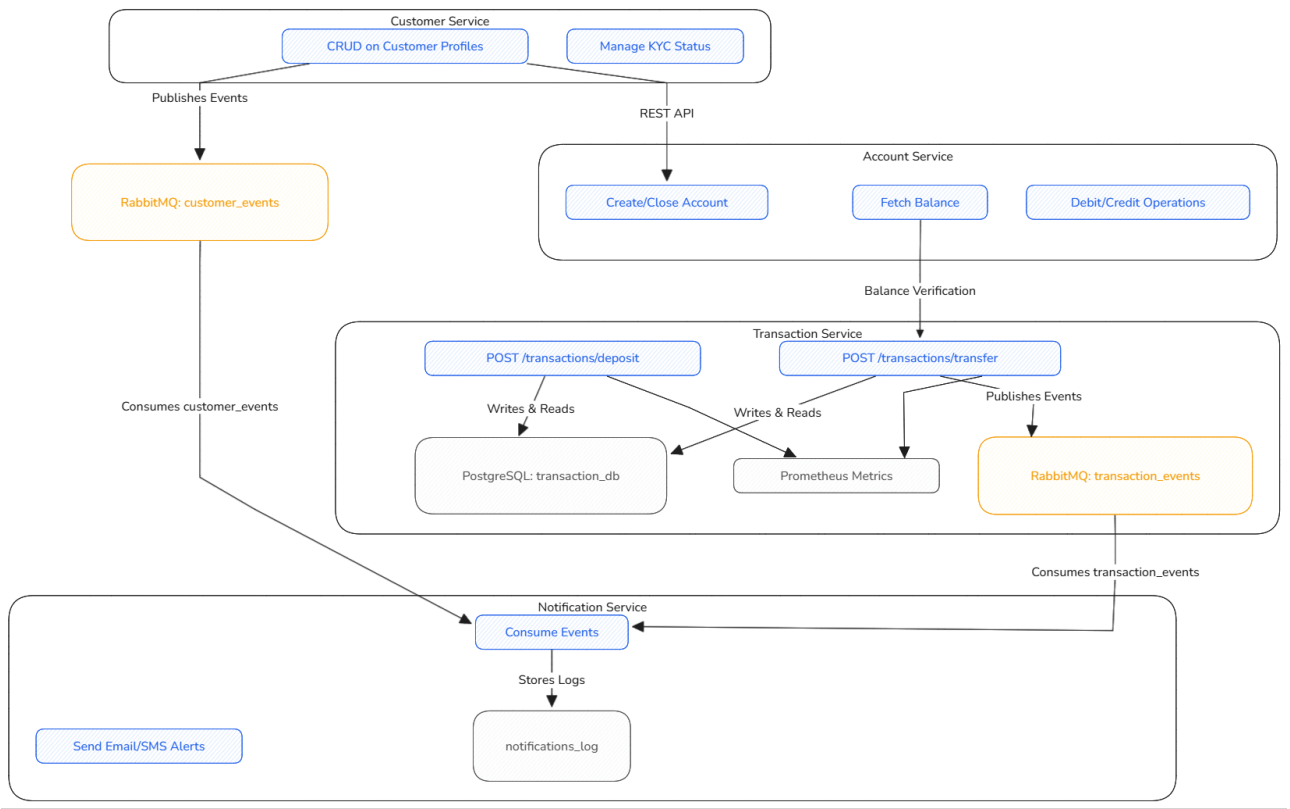
- transactions (txn_id, account_id, amount, type, ref, created_at)
- idempotency_keys (key, txn_id)



- Each transaction is uniquely identified by txn_id.
- idempotency_keys ensures API safety — no duplicate transactions.
- Logical account_id linkage to Account Service (no actual foreign key since each service owns its DB).

JATIN SAINI (2024TM93002) Repository: Github

Microservice Architecture Diagram



Patterns Used

- **Database per service:** No shared DBs.
- **Asynchronous communication:** RabbitMQ → for DEPOSIT_CREATED and TRANSFER_COMPLETED events.
- **Synchronous REST calls:** Transaction → Account (/balance, /debit, /credit).
- **Data replication pattern:** Transaction stores only account_id and counterparty info (no cross-DB joins).

API Endpoints

Method	Endpoint	Description
GET	/health	Service health check
GET	/db-check	Database connectivity
POST	/transactions/deposit	Deposit into an account
POST	/transactions/transfer	Transfer funds between accounts
GET	/transactions	List all transactions
GET	/metrics	Prometheus metrics endpoint

Business Rules & Validations

Rule	Description	Example
Idempotency	No duplicate processing if same key reused	Idempotency-Key: same-123

Rule	Description	Example
Positive Amount	Must be > 0	Reject 0 or negative
Daily Transfer Limit	₹200,000 per account per day	Reject if exceeded
Account Status	Must be ACTIVE	Reject frozen/inactive
Sufficient Balance	No overdraft	Reject if balance < amount
Self-Transfer Prevention	from ≠ to account	Reject same IDs

Containerization with Docker

Containerize each microservice and its dependent components to ensure consistent, isolated, and reproducible runtime environments. Validate connectivity among services through docker-compose.

Inter-Service Communication

- **Synchronous REST** → Account Service
    - /balance (check balance)
    - /debit / /credit (update funds)
- **Asynchronous Events** → Notification Service via RabbitMQ
    - Queue: transaction_events
    - Event types:
        - DEPOSIT_CREATED
        - TRANSFER_COMPLETED

Dockerfile Highlights

Each Node.js microservice contains a lightweight Dockerfile:

```
FROM node:20-alpine

WORKDIR /app

COPY package*.json ./

RUN npm install --production

COPY . .

EXPOSE 8082

CMD ["npm","start"]
```

This ensures a minimal footprint and quick build times.

Docker-Compose Setup

Transaction service is orchestrated via a single docker-compose.yml:

- **Networking:** Shared default network for inter-service REST/RabbitMQ communication.
- **Persistent Storage:** Postgres volume (txn_data) maintains data across container restarts.

- **Environment Variables:** Injected securely for DB credentials, ports, and RabbitMQ URLs.
- **Dependencies:** depends_on ensures DB and RabbitMQ are initialized before app startup.

Verification Steps:

Build & Start docker compose up –build

Check Containers docker ps

Health Endpoints

- http://localhost:8082/health → "Transaction Service running"
- http://localhost:8083/health → "Account Service running"
- http://localhost:15672 → RabbitMQ management console (guest/guest)
- **Database Connectivity**
- http://localhost:8082/db-check → confirms Postgres connection.
- **Metrics**
- http://localhost:8082/metrics → Prometheus-formatted metrics exported successfully.

Kubernetes Deployment (Minikube)

kubectl create namespace banking

kubectl apply -f k8s/transaction-service/ -n banking

kubectl get pods -n banking

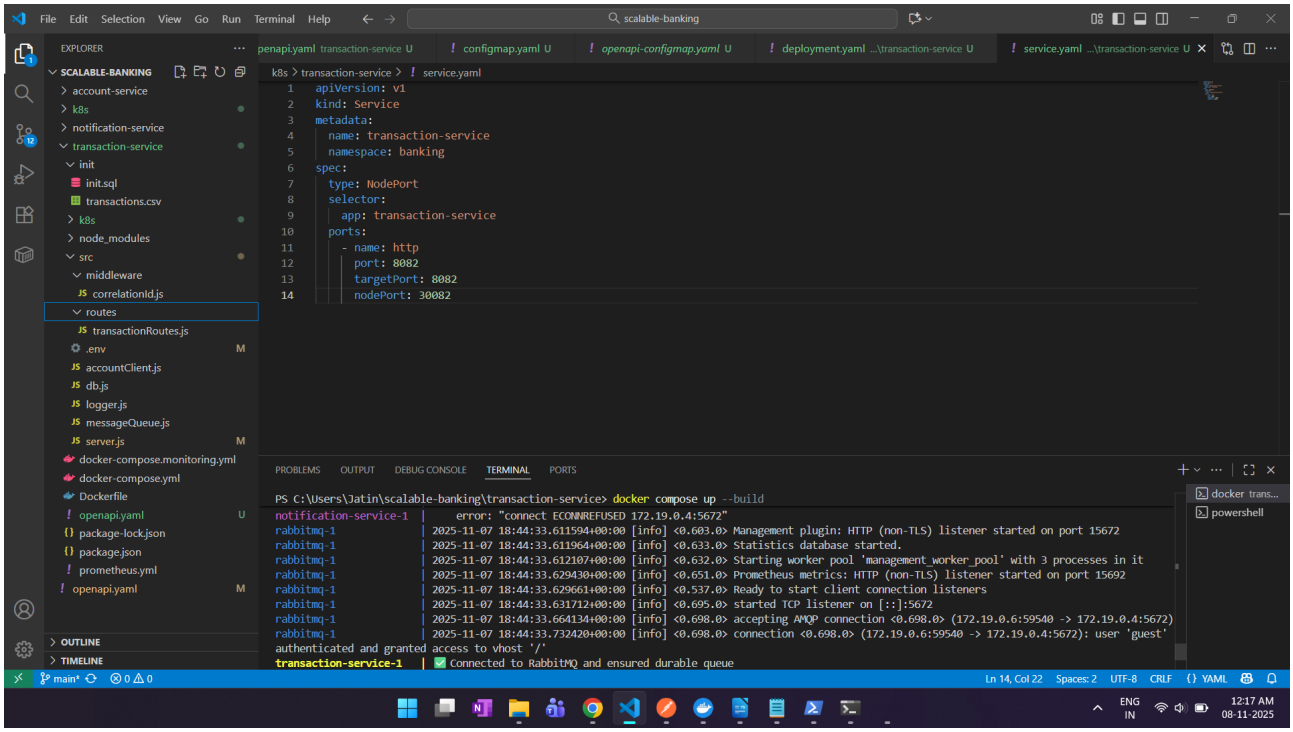
kubectl get svc -n banking

minikube service transaction-service -n banking –url

Screenshots:

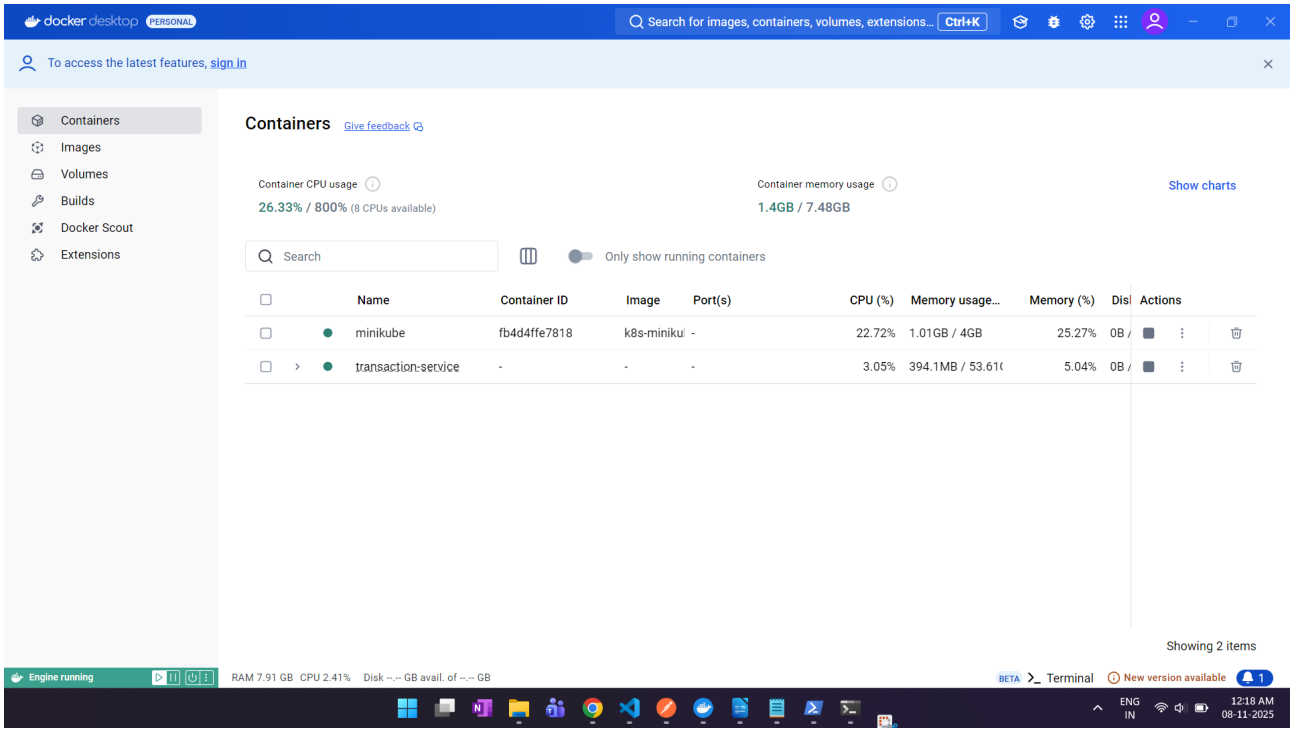
Started docker:

JATIN SAINI (2024TM93002) Repository: [Github]

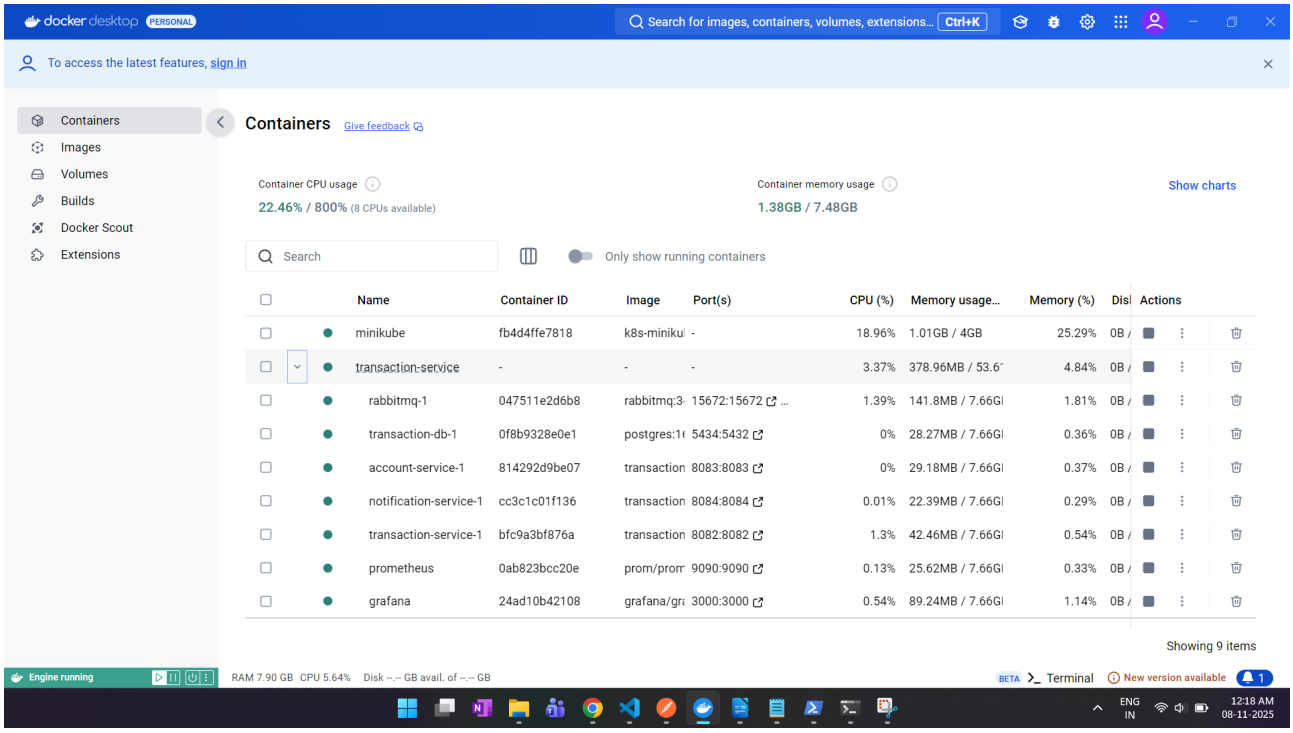


JATIN SAINI (2024TM93002) Repository: Github

Docker Desktop:



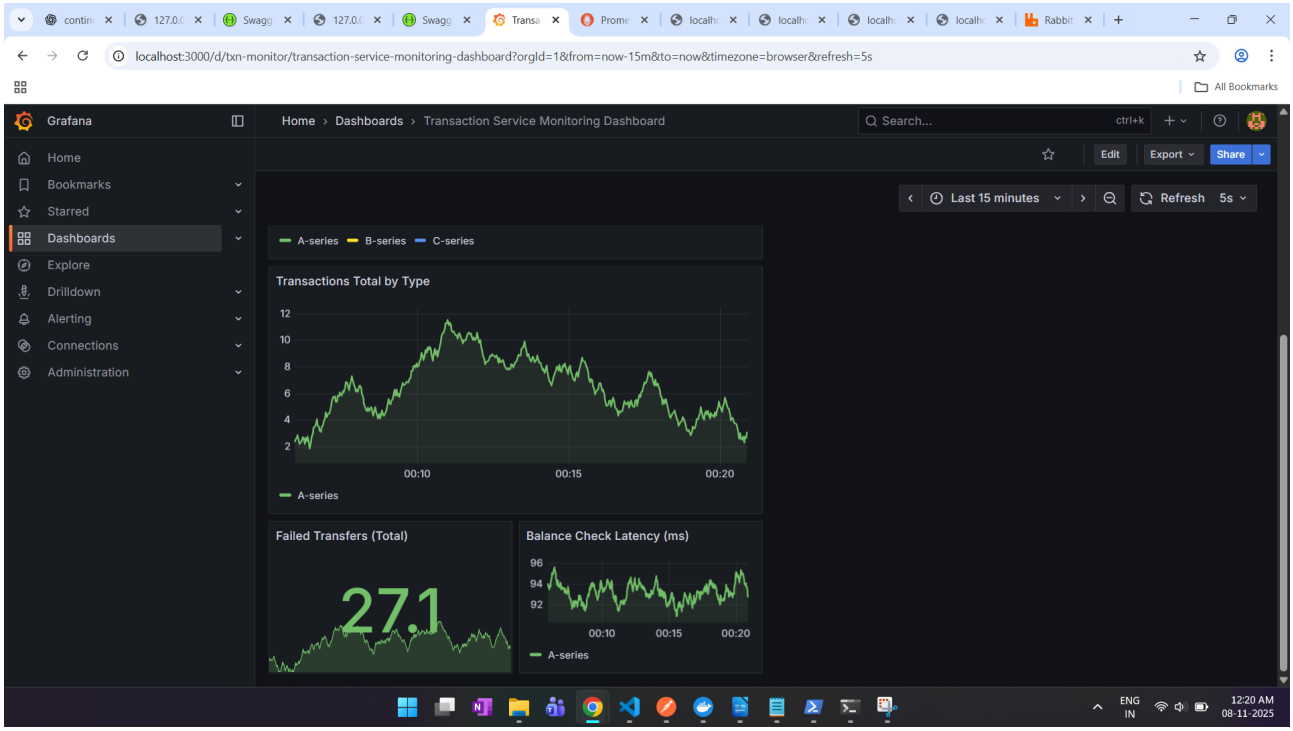
All inner service are running fine inside transaction-service container

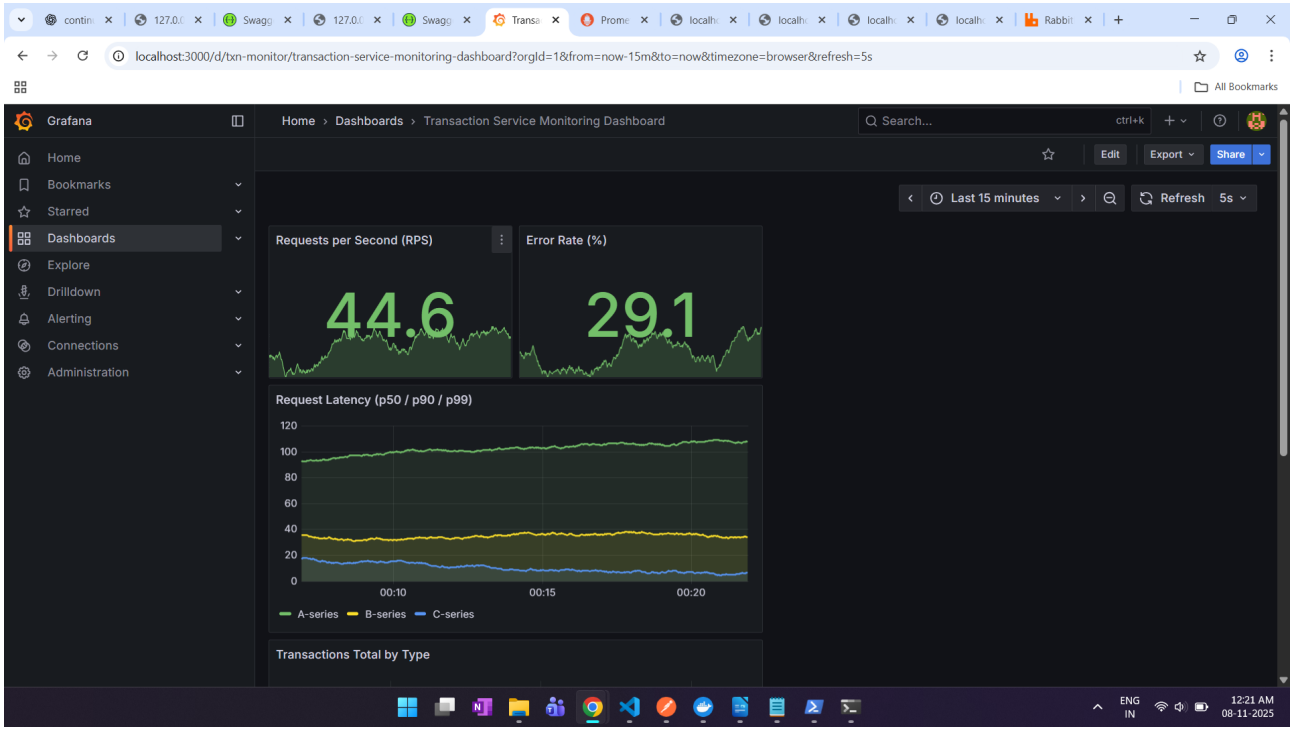


JATIN SAINI (2024TM93002) Repository: Github

Grafana dashboard:

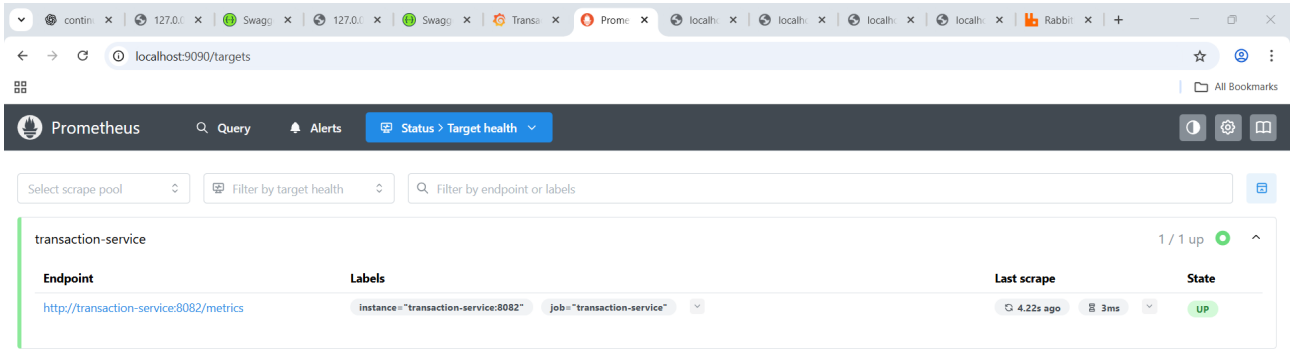
for visual metrics of transaction service



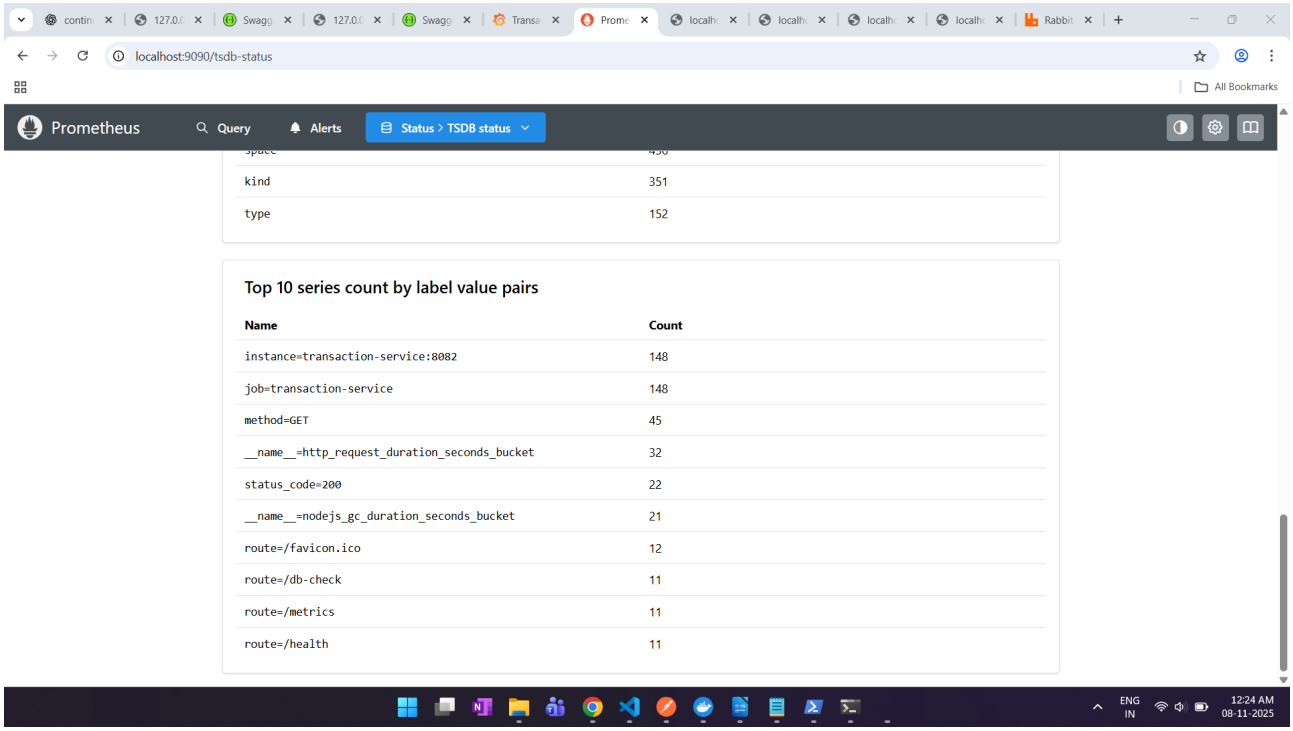


JATIN SAINI (2024TM93002) Repository: Github

Prometheus:



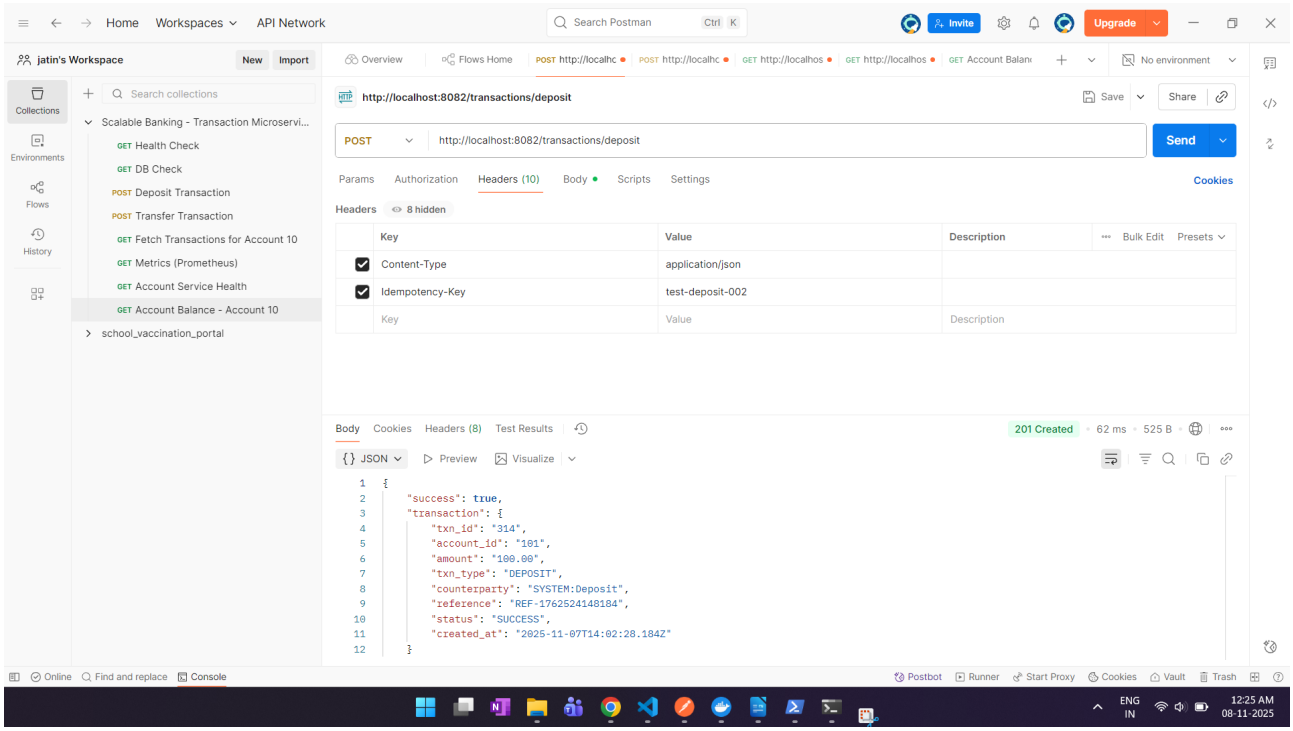
Metric in Prometheus:

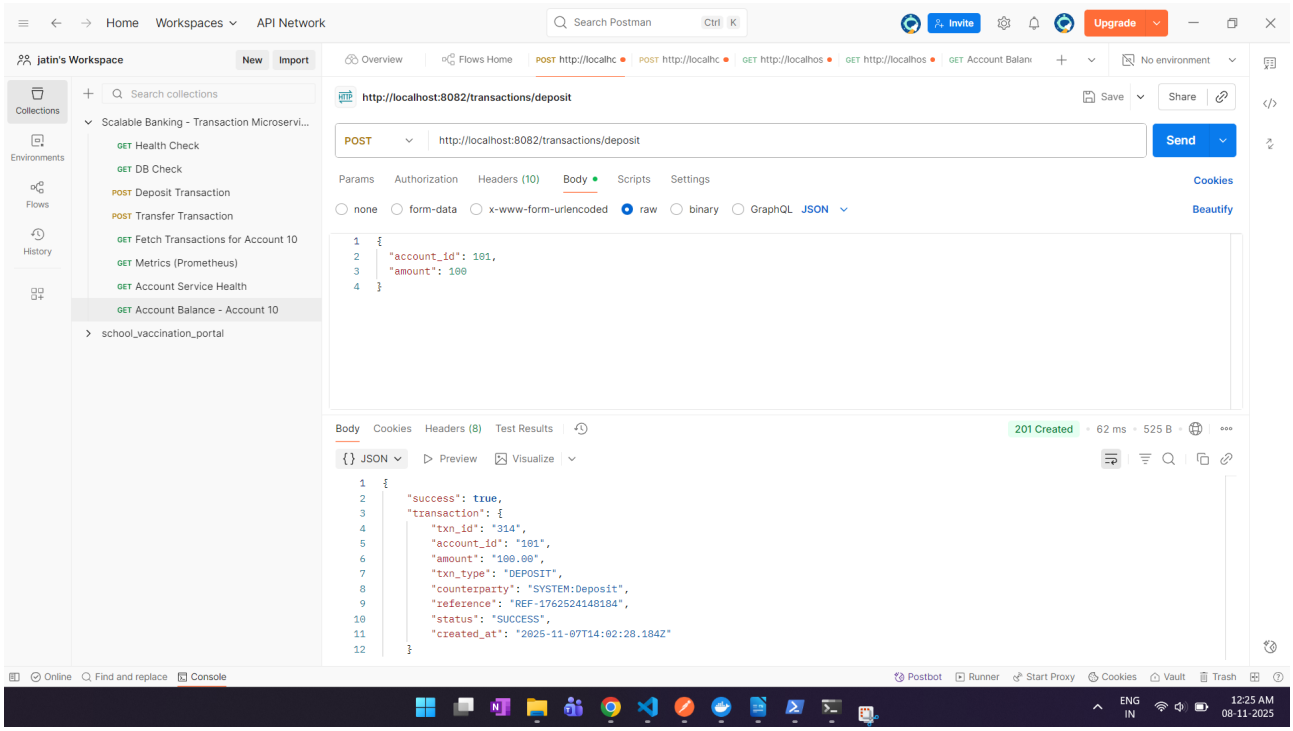


JATIN SAINI (2024TM93002) Repository: Github

API:

POST http://localhost:8082/transactions/deposit





{

"success": true,

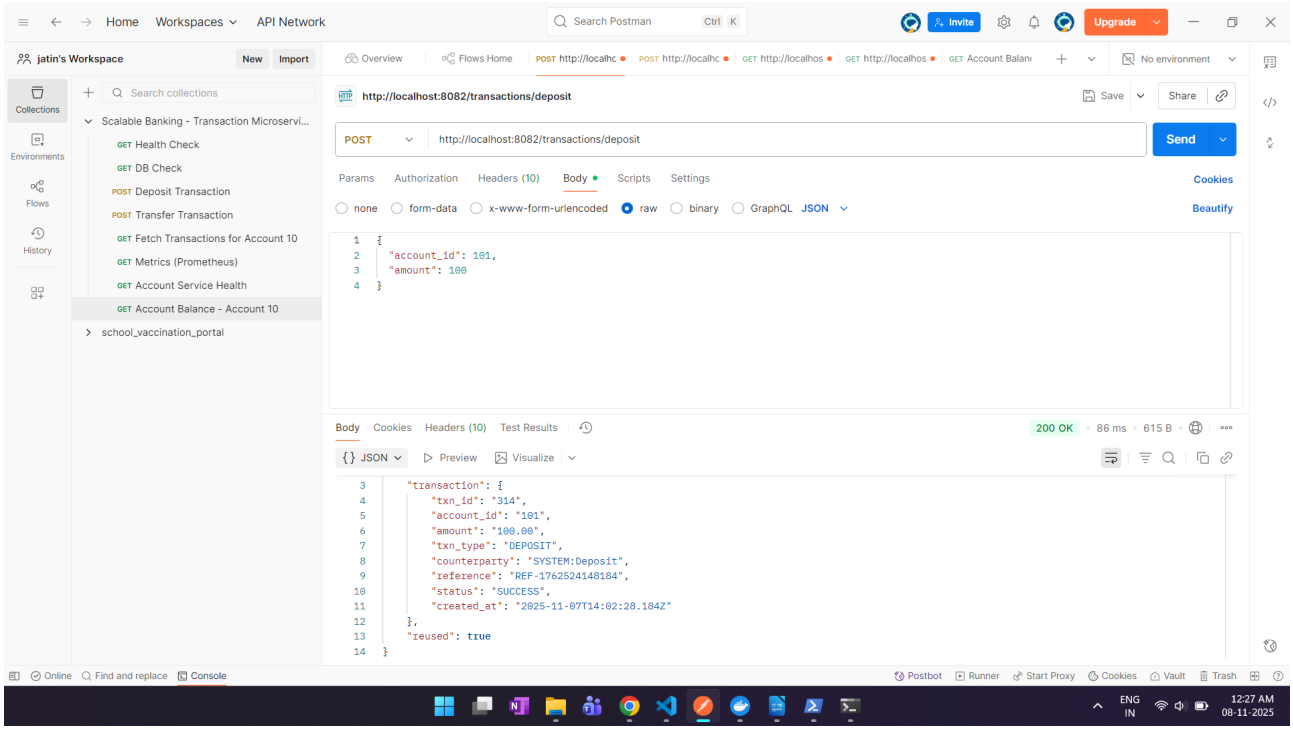
"transaction": {

"txn_id": "317",

"account_id": "101",

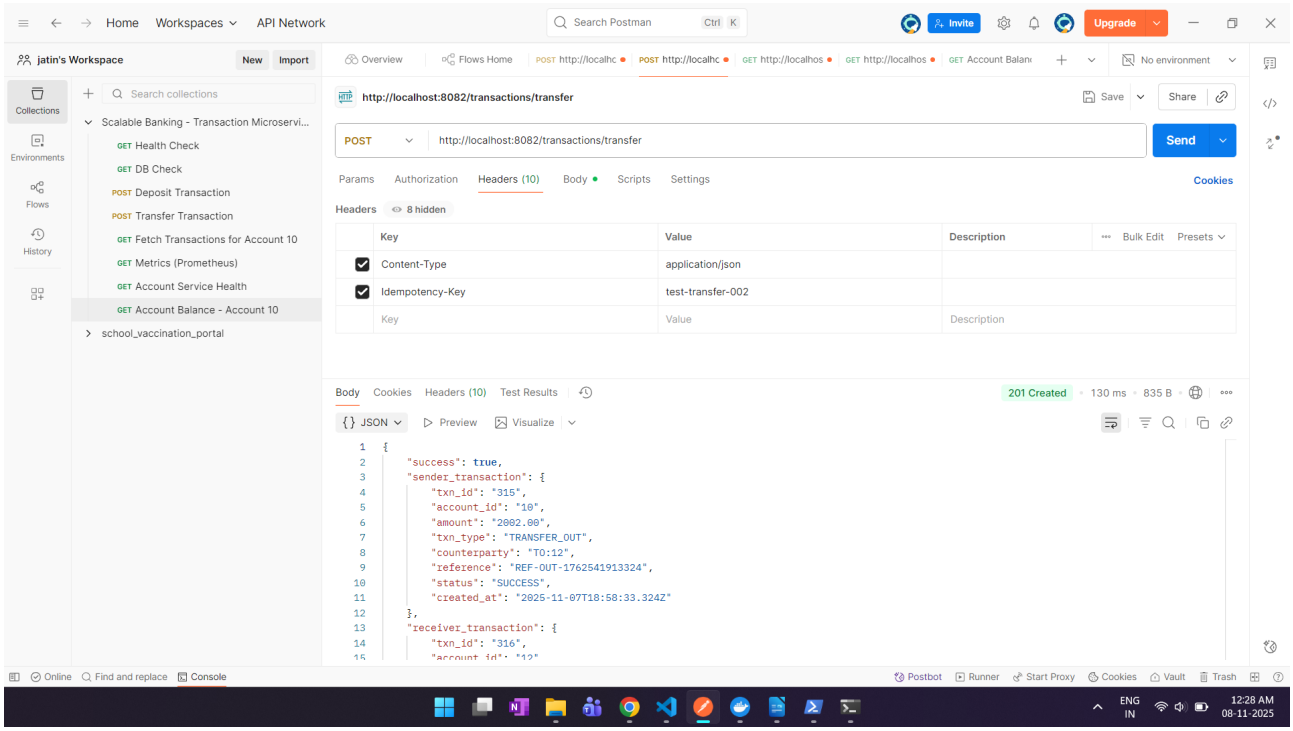
```
    "amount": "101.00",
    "txn_type": "DEPOSIT",
    "counterparty": "SYSTEM:Deposit",
    "reference": "REF-1762542603692",
    "status": "SUCCESS",
    "created_at": "2025-11-07T19:10:03.692Z"
  }
}
```

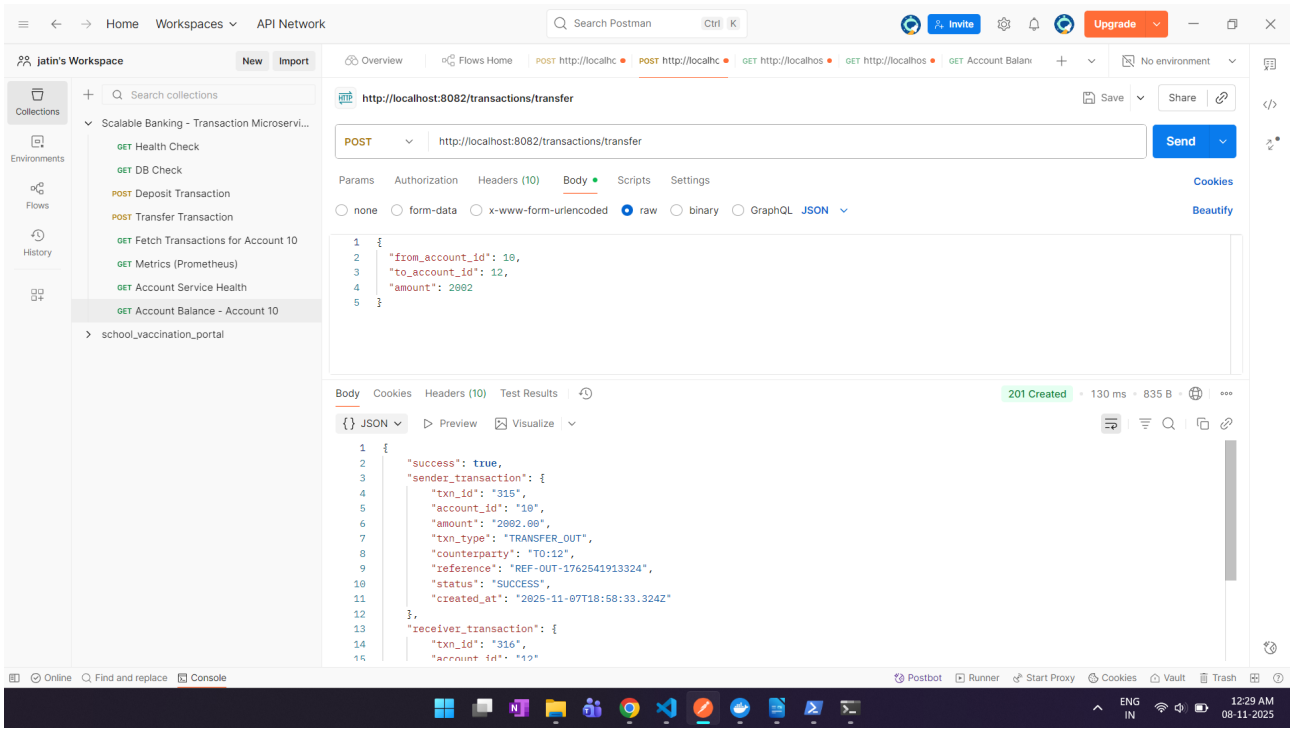

If same request sent again getting reused:true



JATIN SAINI (2024TM93002) Repository: Github

POST http://localhost:8082/transactions/transfer





{

"success": true,

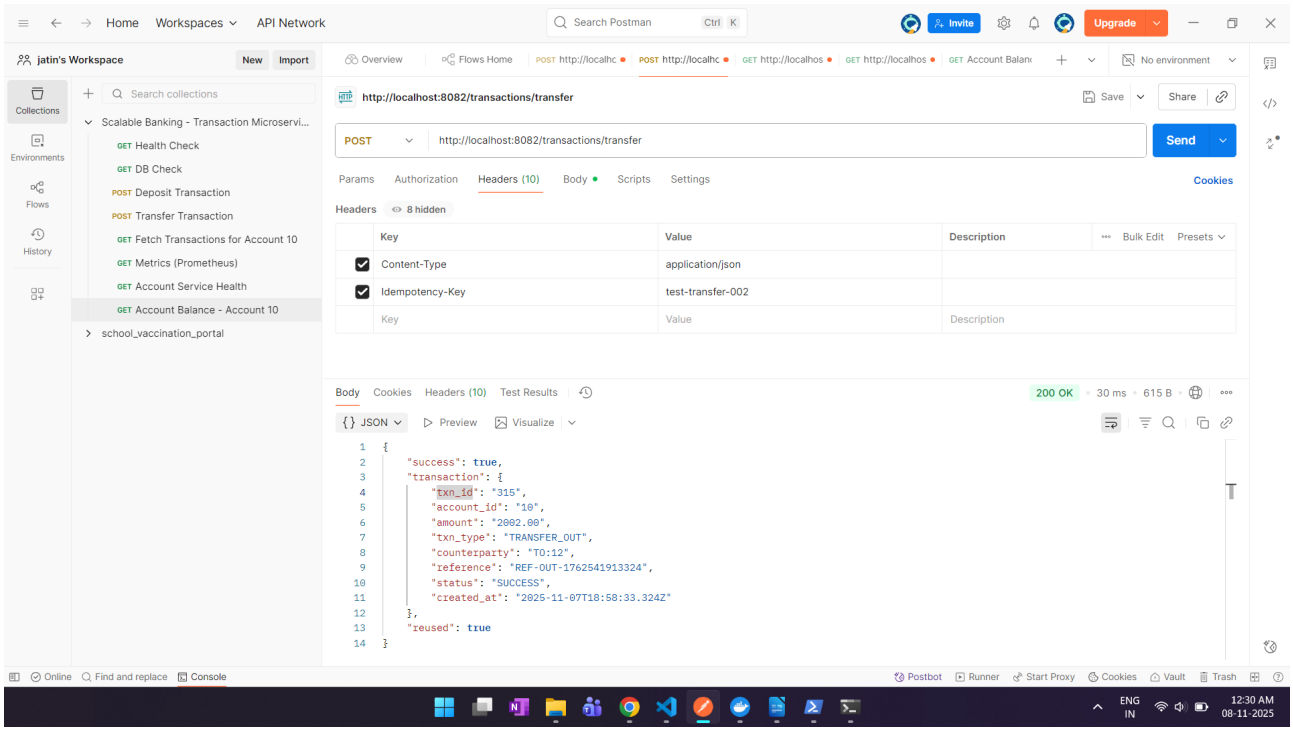
"sender_transaction": {

"txn_id": "315",

"account_id": "10",

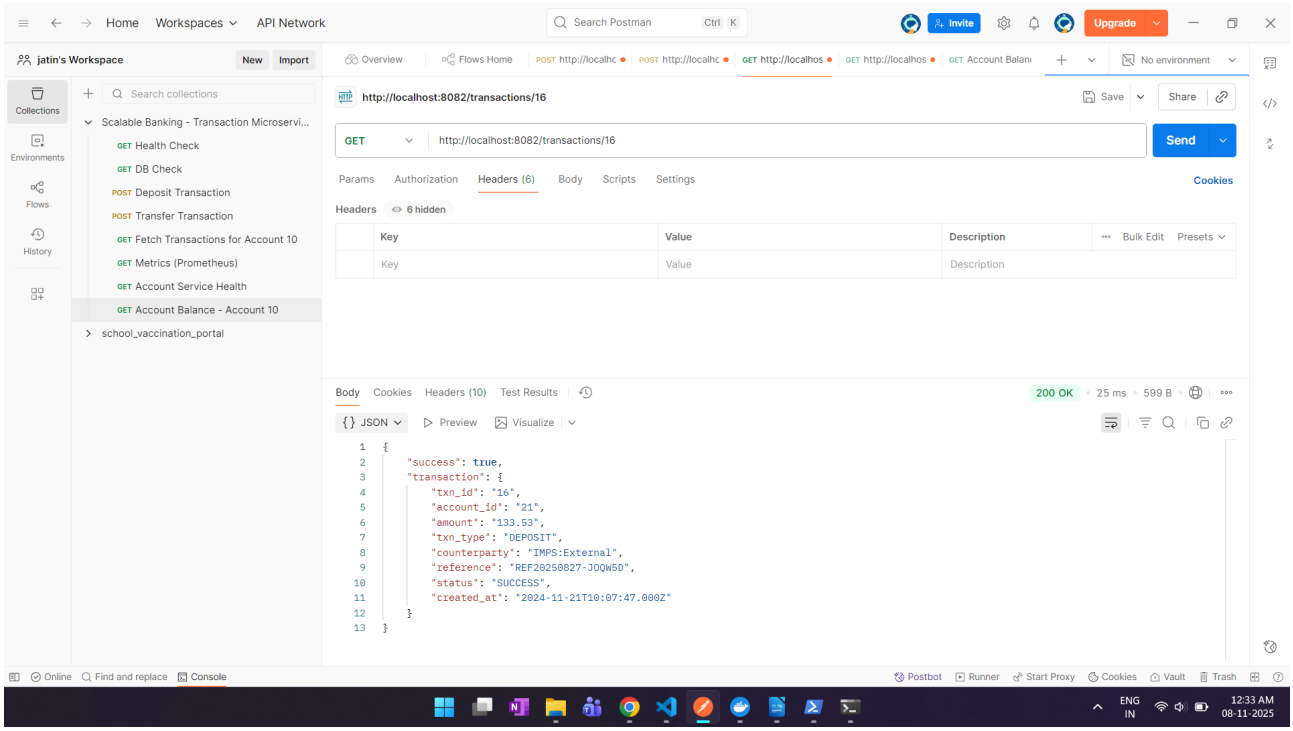
```
    "amount": "2002.00",
    "txn_type": "TRANSFER_OUT",
    "counterparty": "TO:12",
    "reference": "REF-OUT-1762541913324",
    "status": "SUCCESS",
    "created_at": "2025-11-07T18:58:33.324Z"
  },
  "receiver_transaction": {
    "txn_id": "316",
    "account_id": "12",
    "amount": "2002.00",
    "txn_type": "TRANSFER_IN",
    "counterparty": "FROM:10",
    "reference": "REF-IN-1762541913332",
    "status": "SUCCESS",
    "created_at": "2025-11-07T18:58:33.324Z"
  }
}
```


If same request is send again, getting reused:true



JATIN SAINI (2024TM93002) Repository: Github

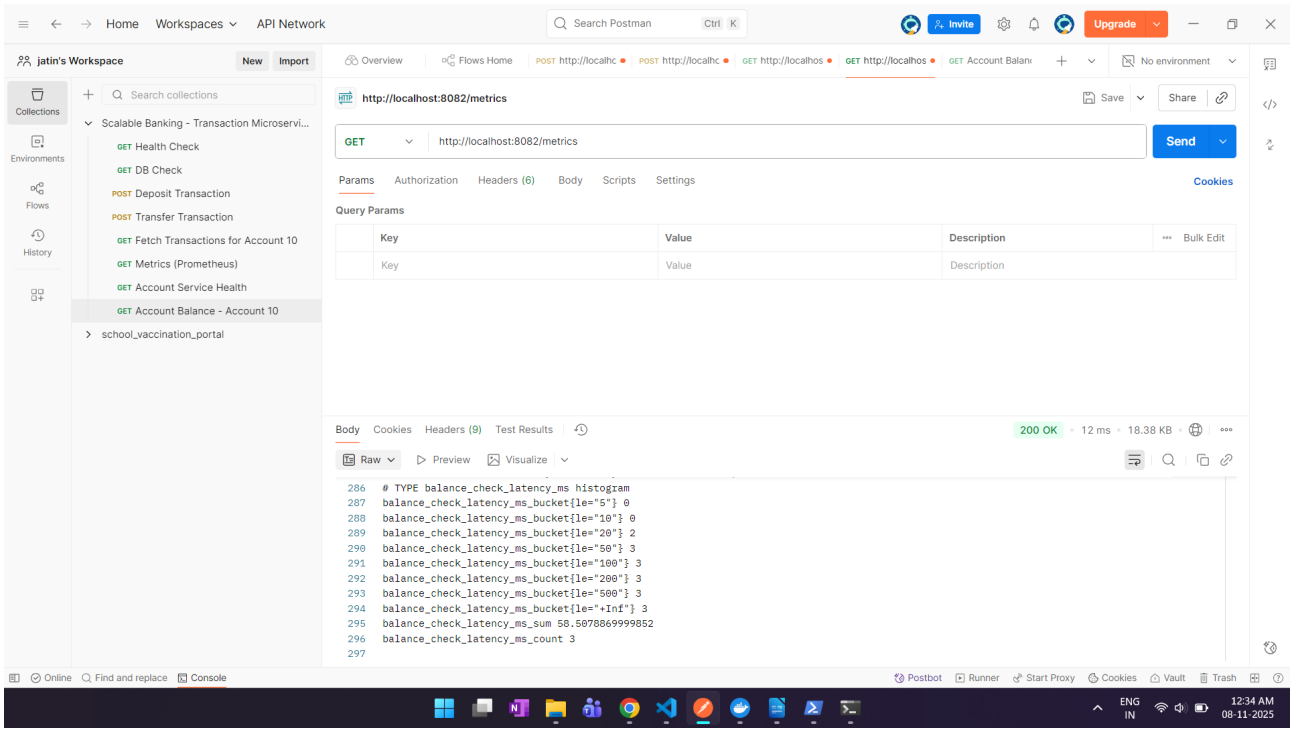
Get http://localhost:8082/transactions/16



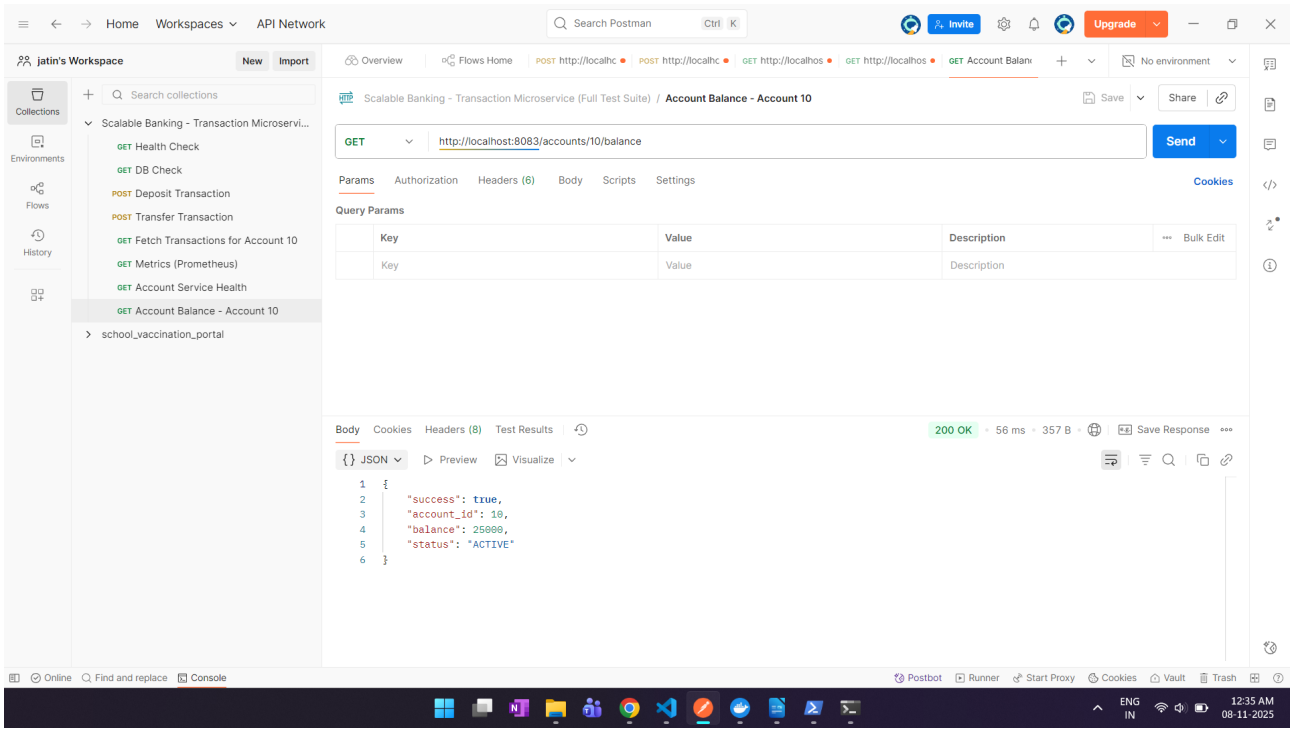

```
{
  "success": true,
  "transaction": {
    "txn_id": "16",
    "account_id": "21",
    "amount": "133.53",
    "txn_type": "DEPOSIT",
    "counterparty": "IMPS:External",
    "reference": "REF20250827-JOQW5D",
    "status": "SUCCESS",
    "created_at": "2024-11-21T10:07:47.000Z"
  }
}
```


Get http://localhost:8082/metrics

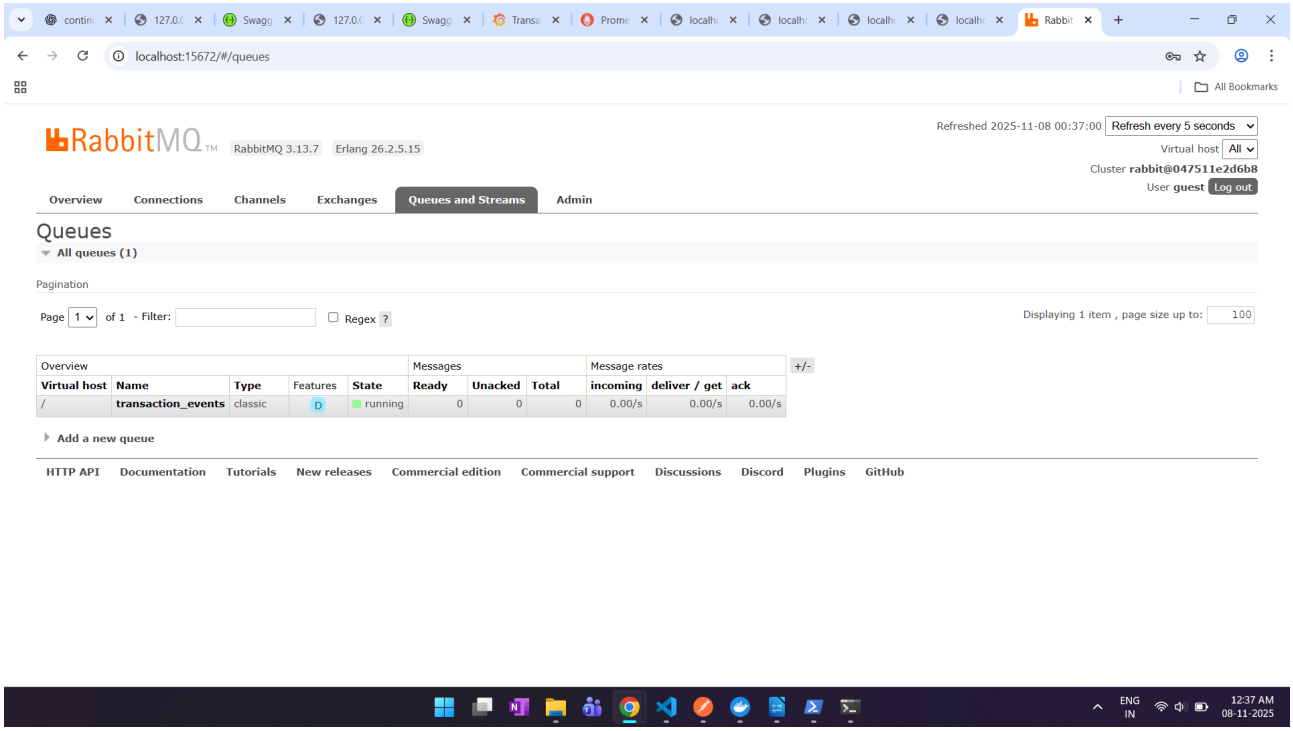
JATIN SAINI (2024TM93002) Repository: Github

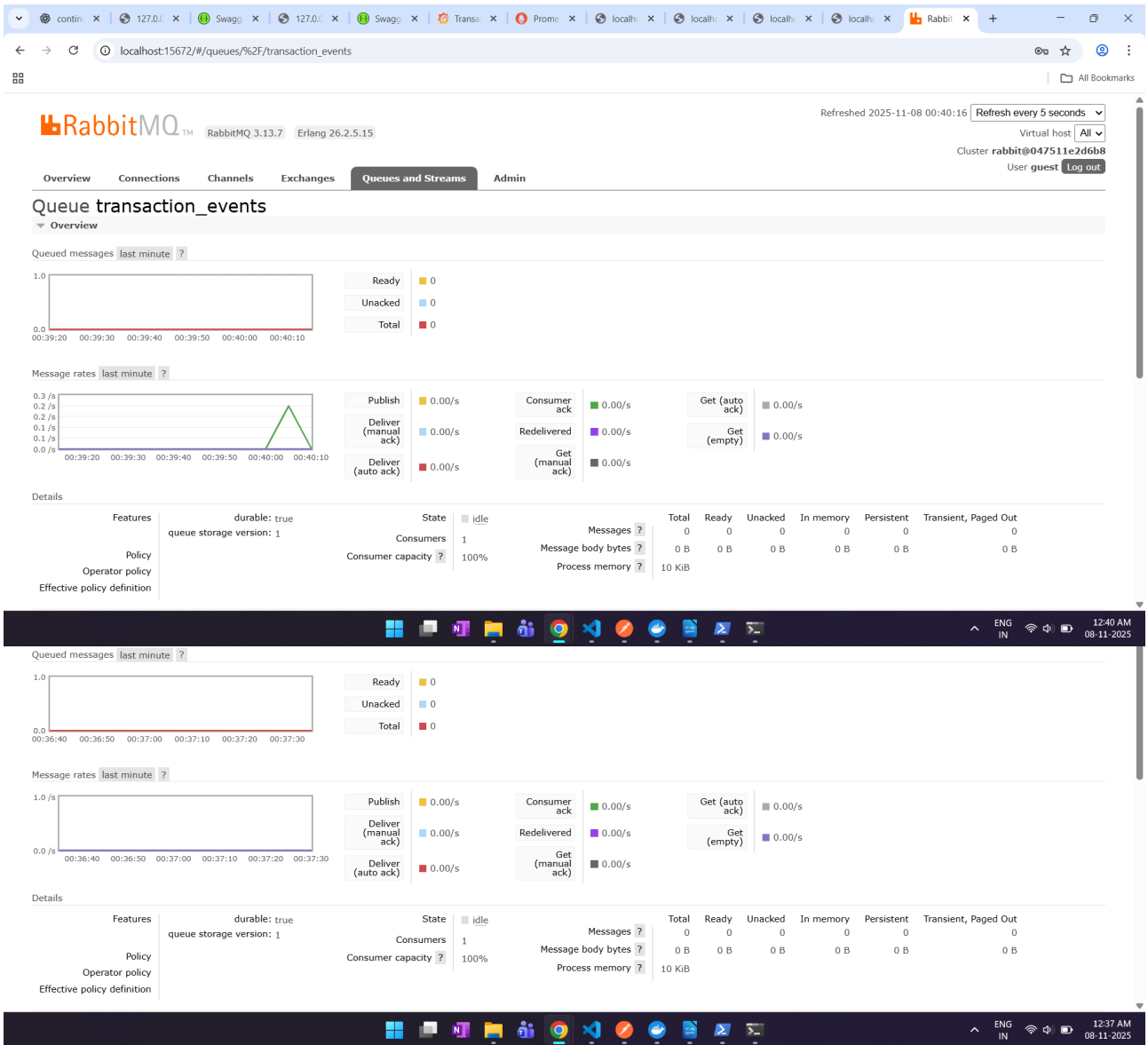


Get http://localhost:8083/accounts/10/balance



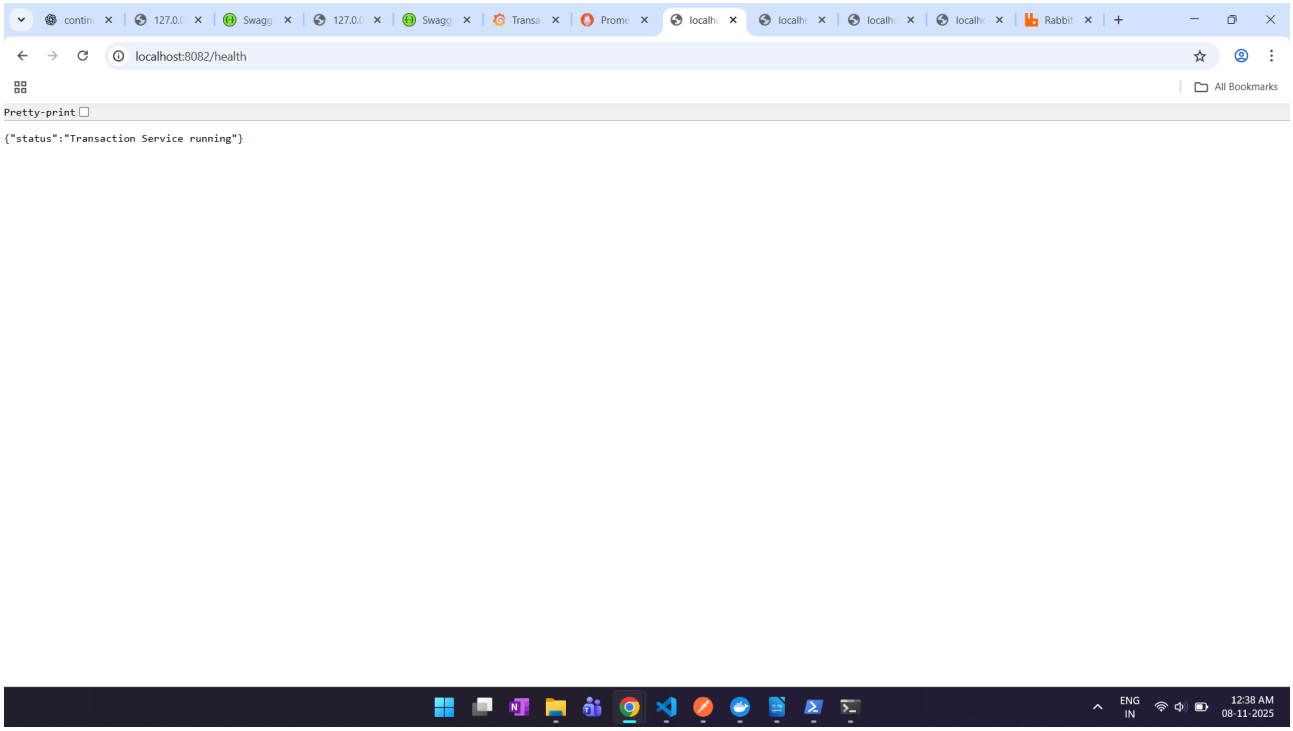
RabbitMQ: transaction_events



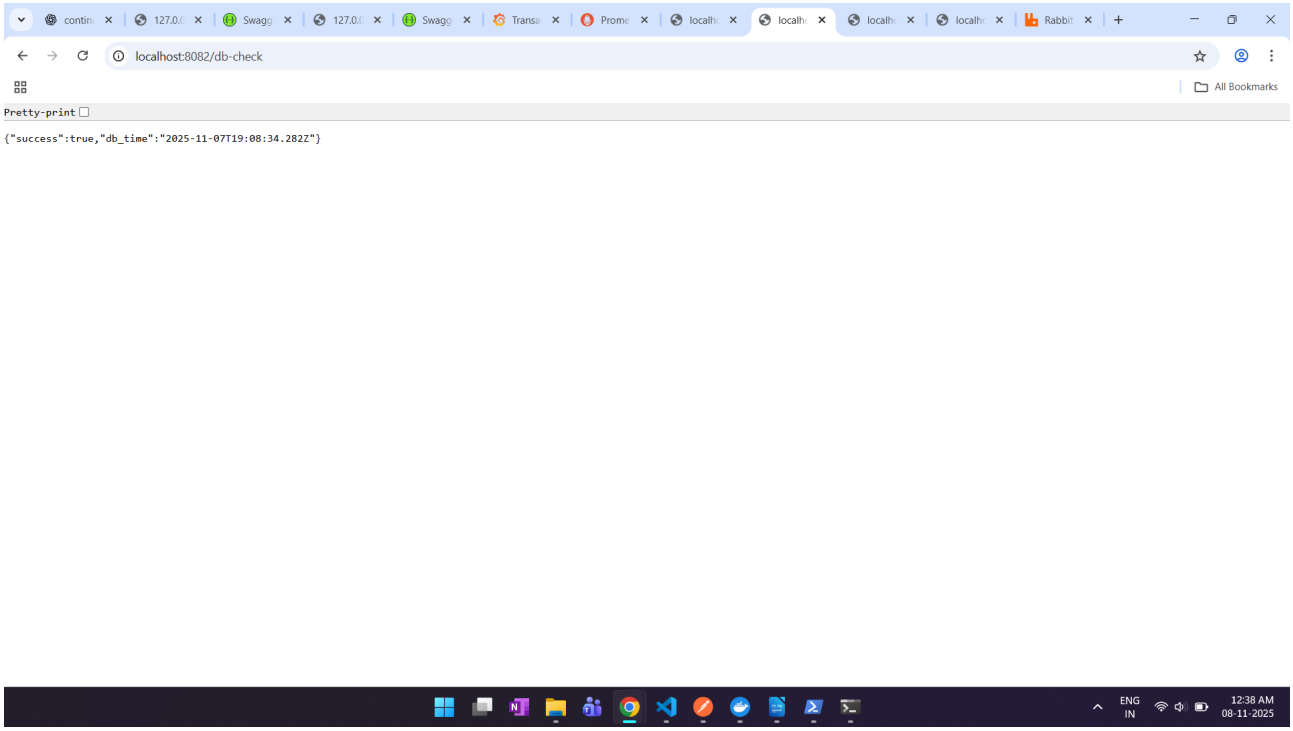


JATIN SAINI (2024TM93002) Repository: Github

Transaction-service health check:

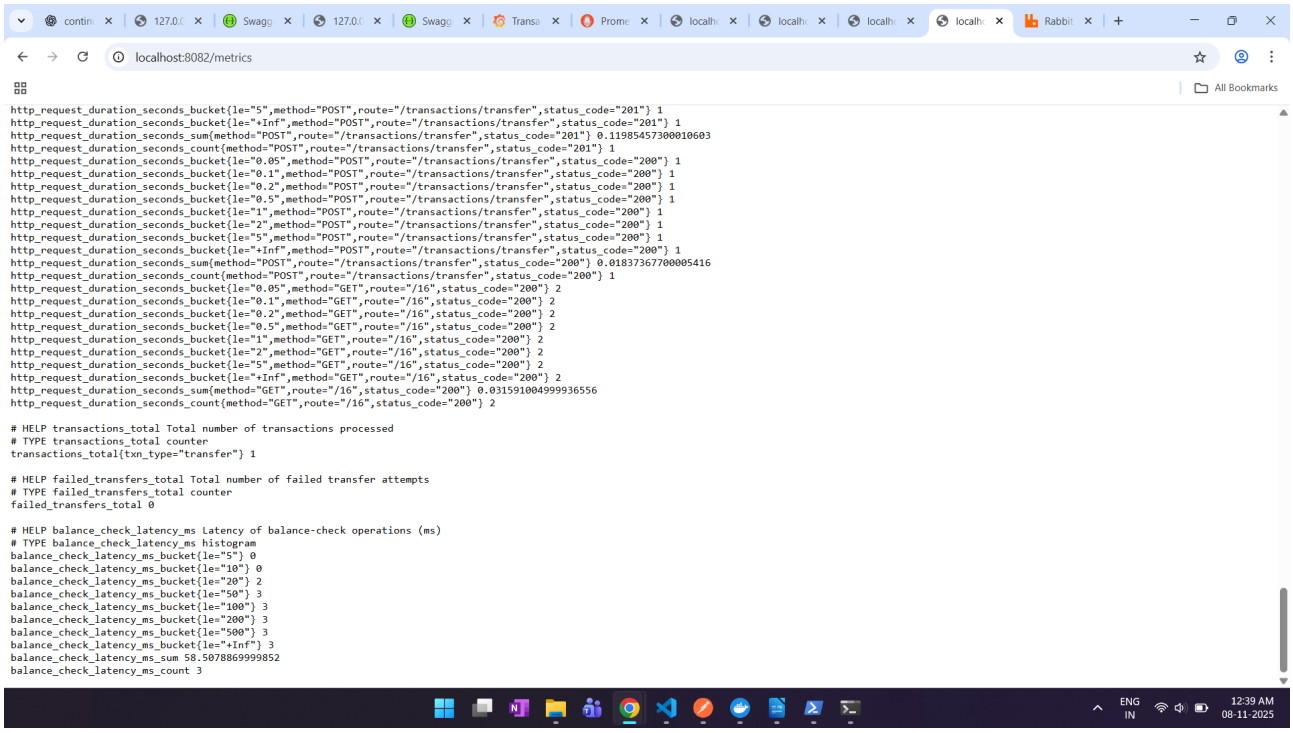


Transaction database health check:



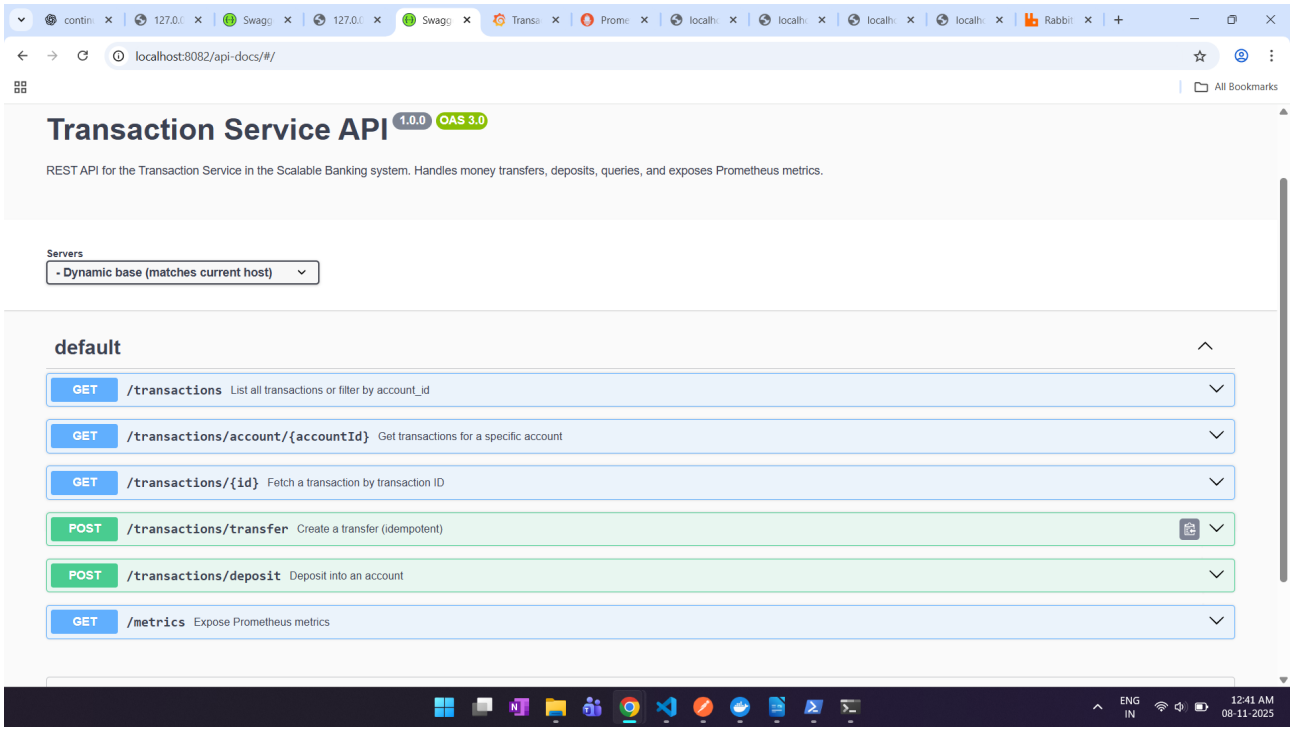
Transaction-service metrics logging:

JATIN SAINI (2024TM93002) Repository: Github

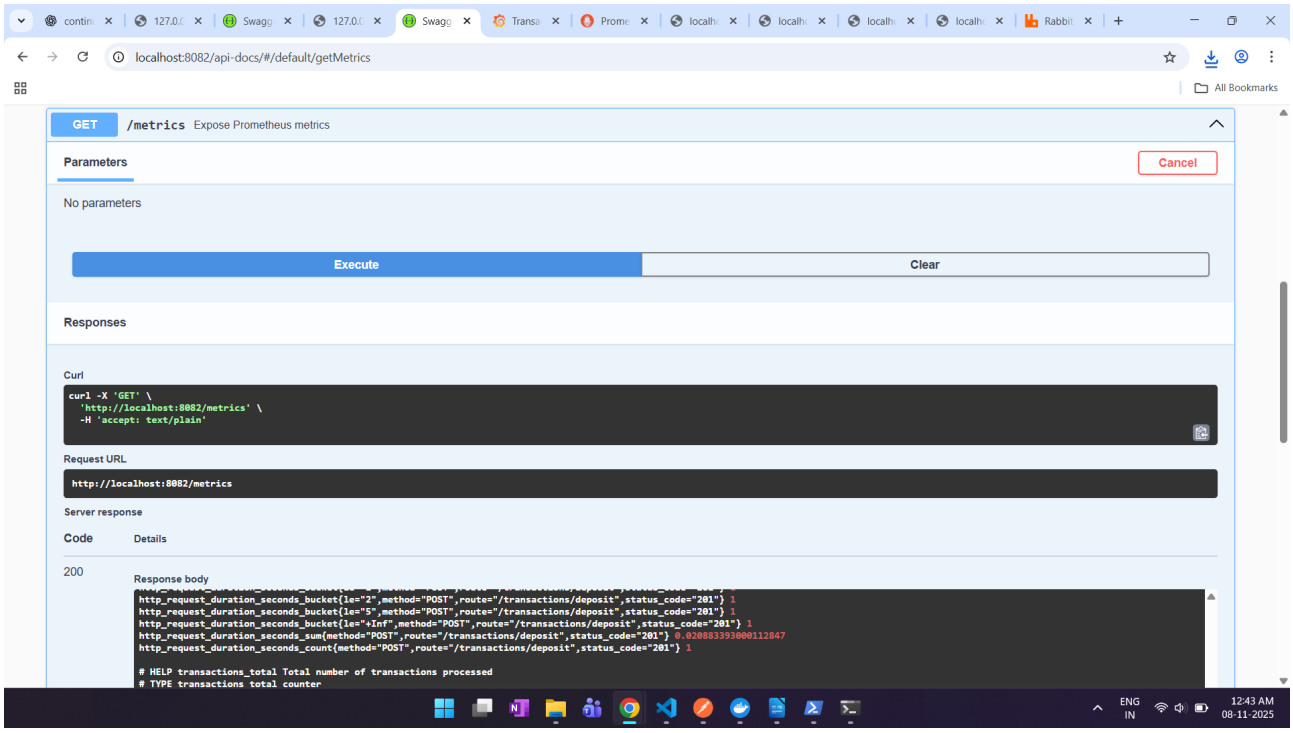


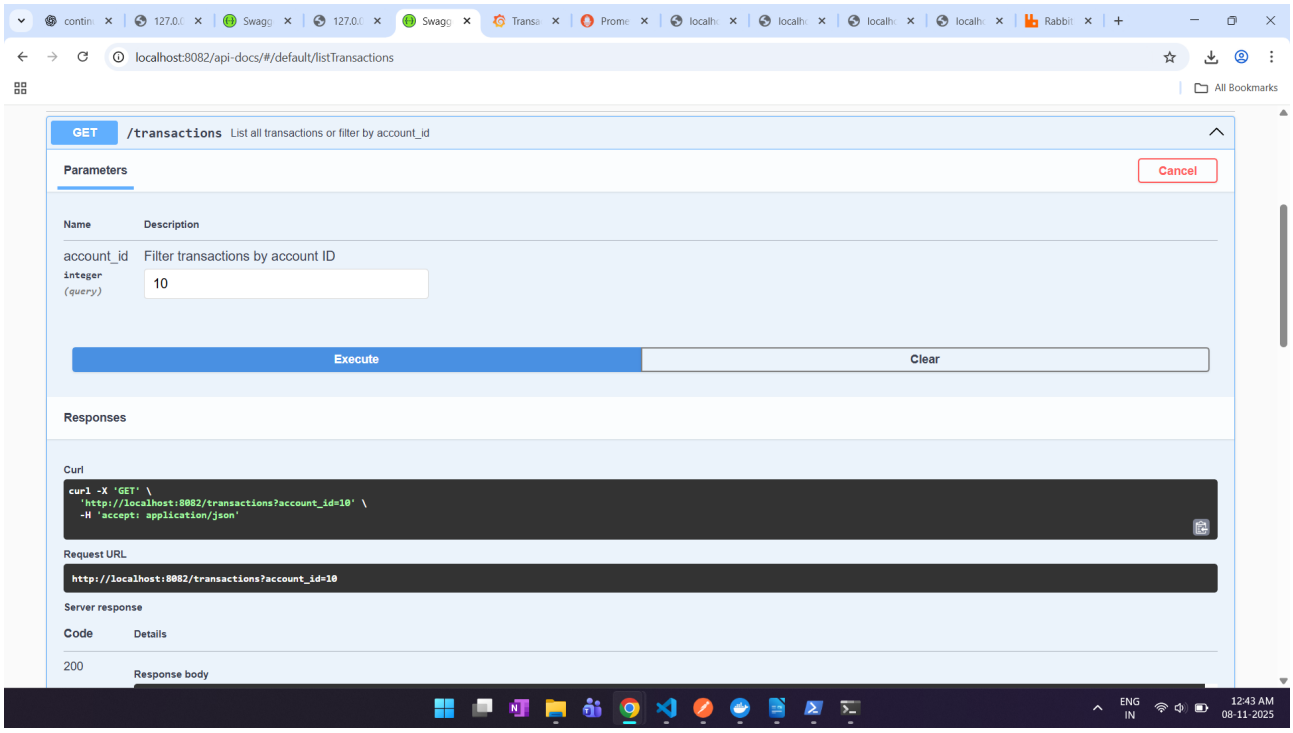
Swagger

Showing all API of transaction-service

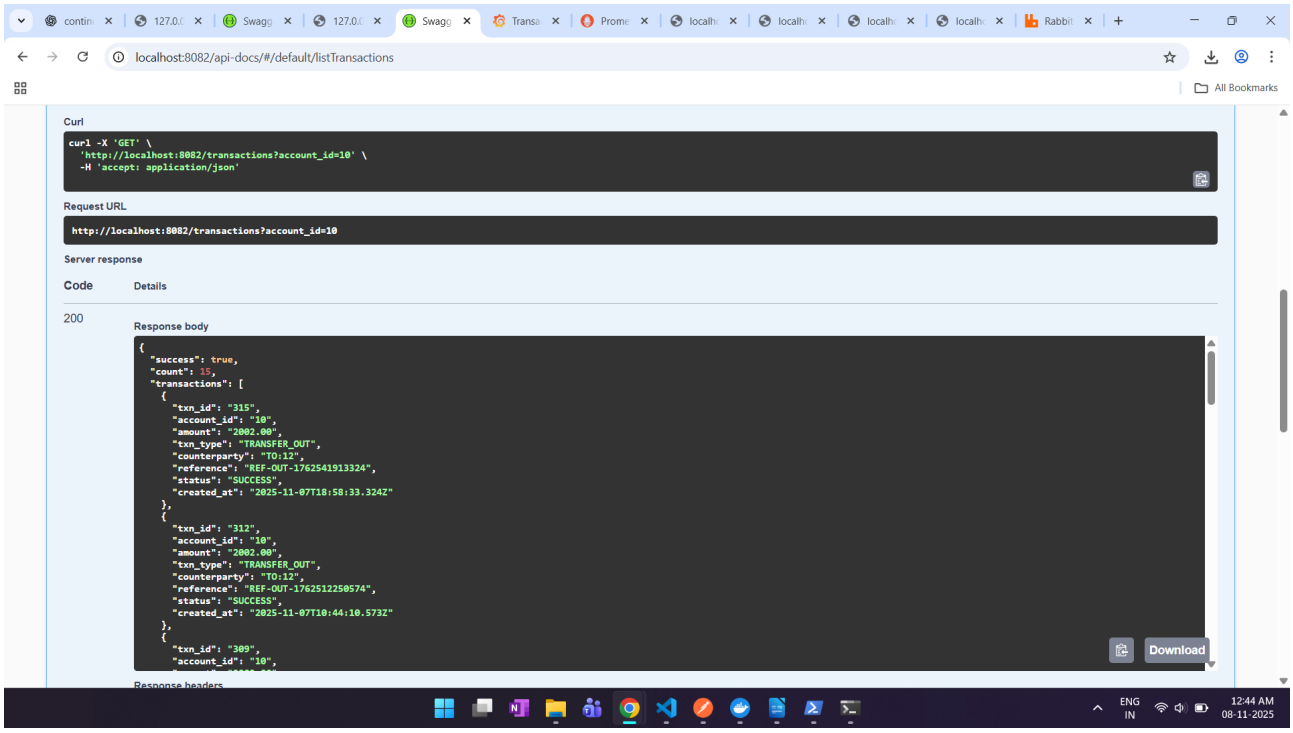


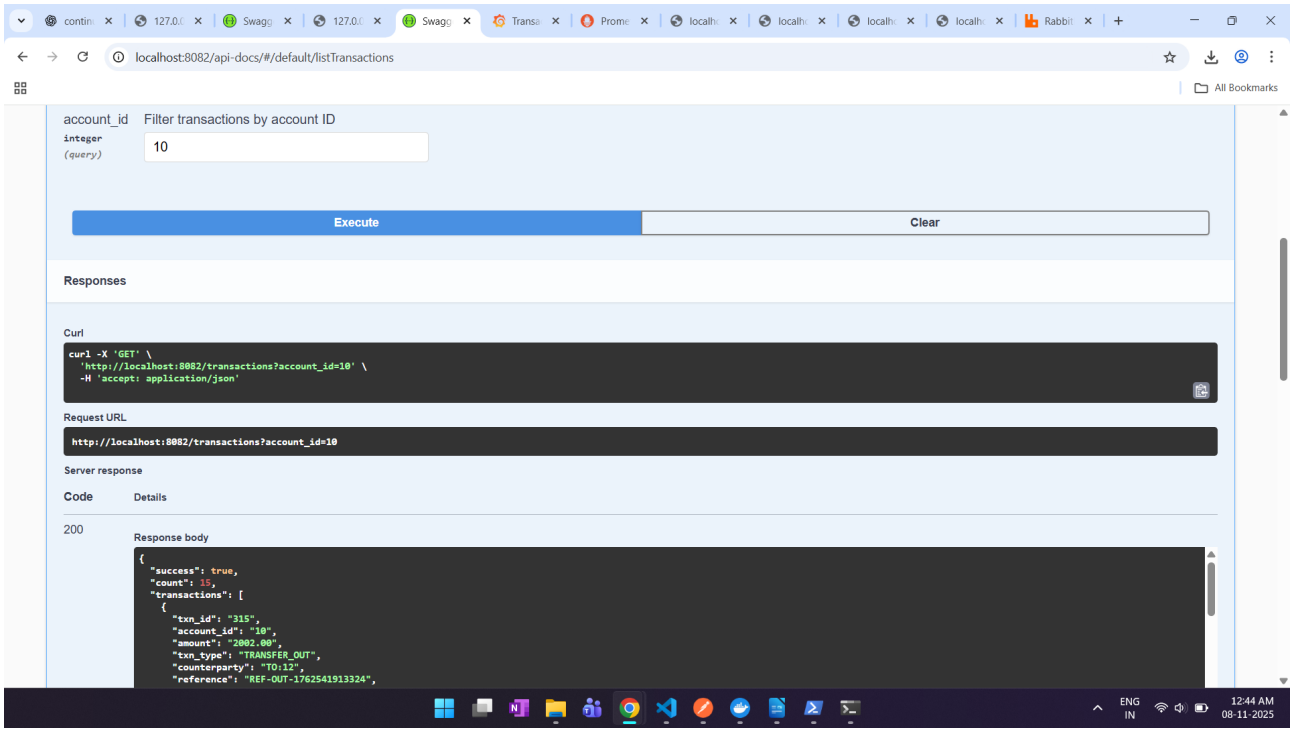
JATIN SAINI (2024TM93002) Repository: Github



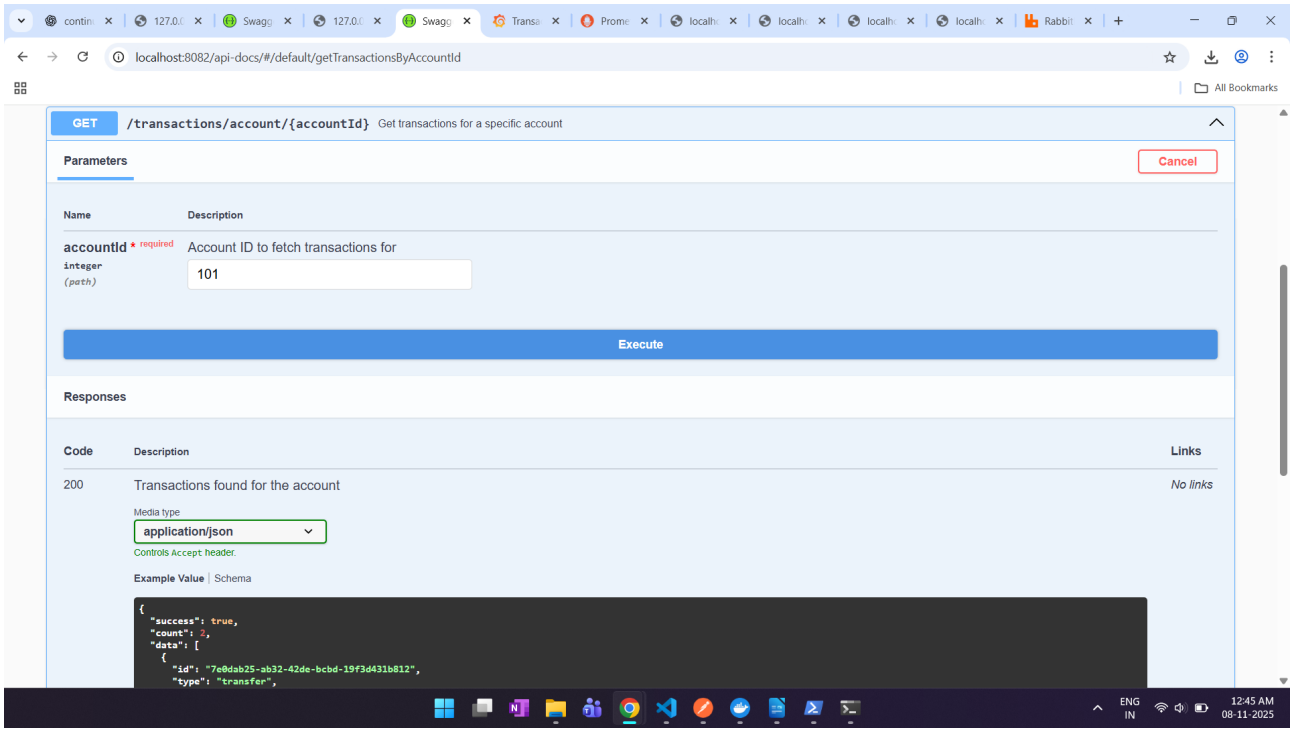


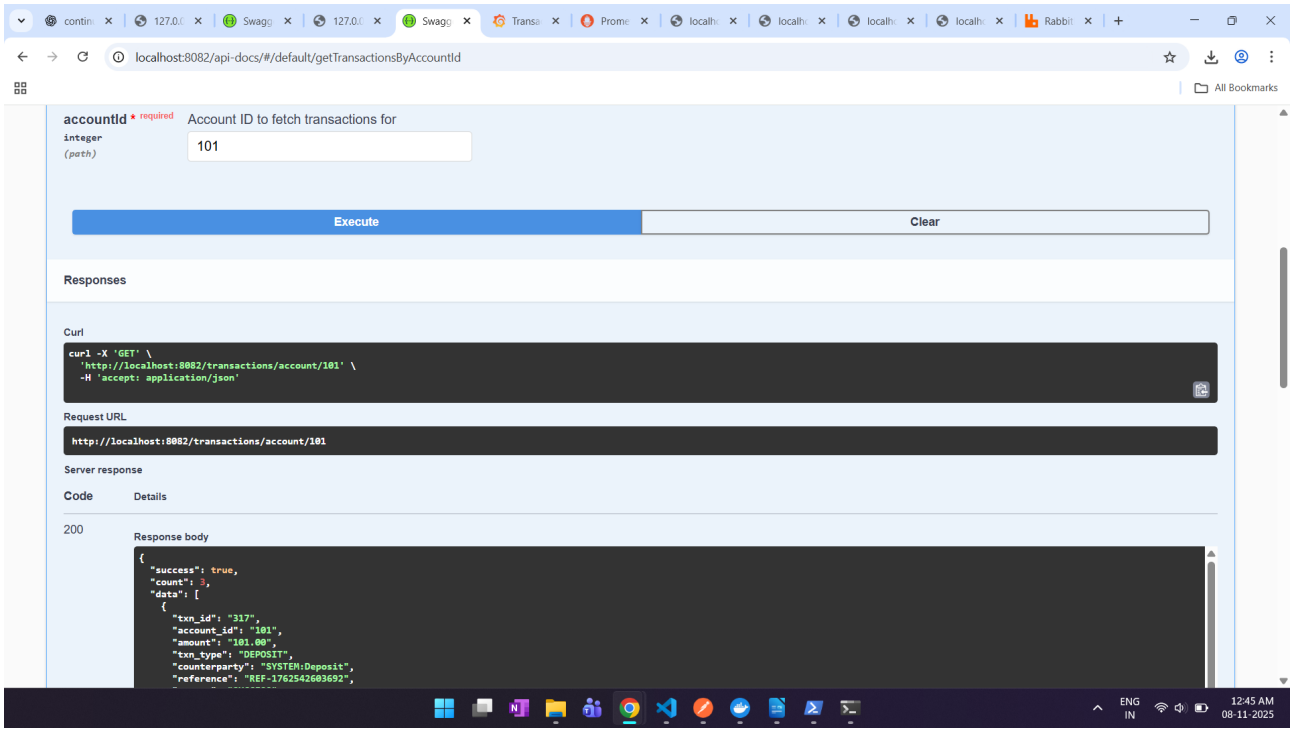
JATIN SAINI (2024TM93002) Repository: Github



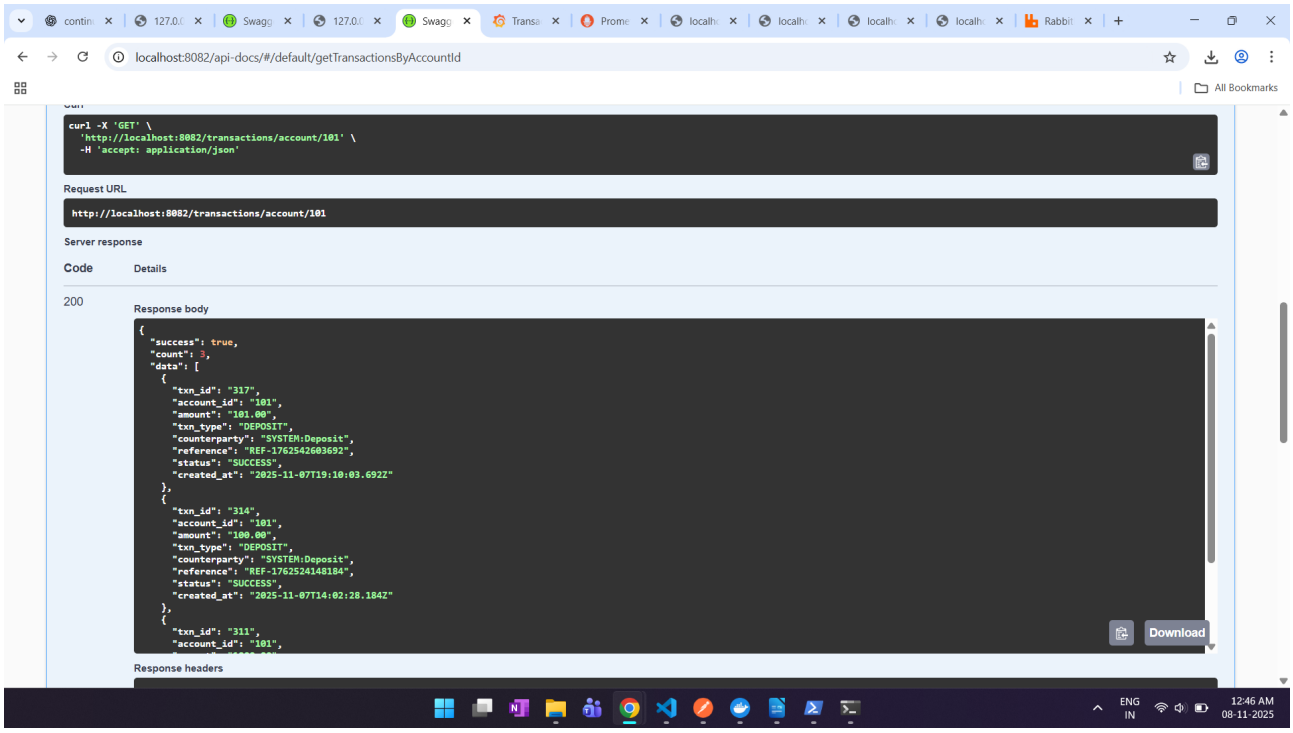


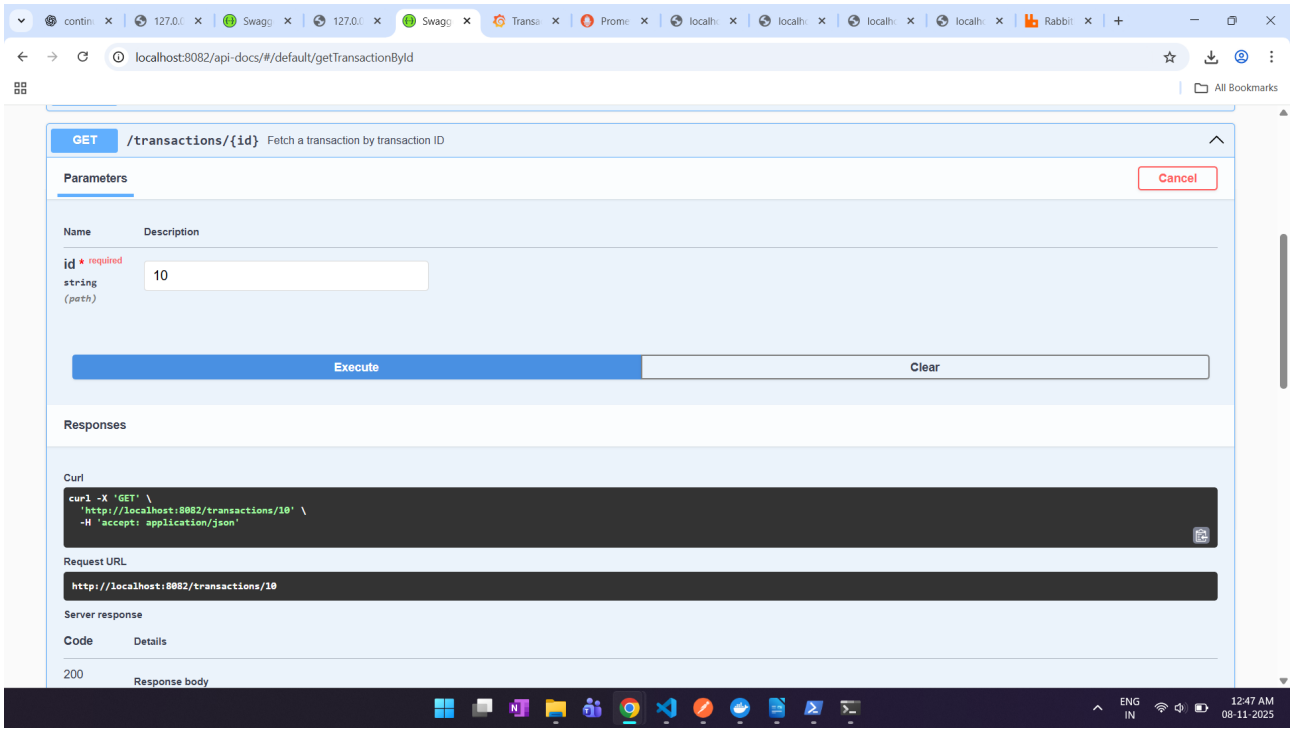
JATIN SAINI (2024TM93002) Repository: Github



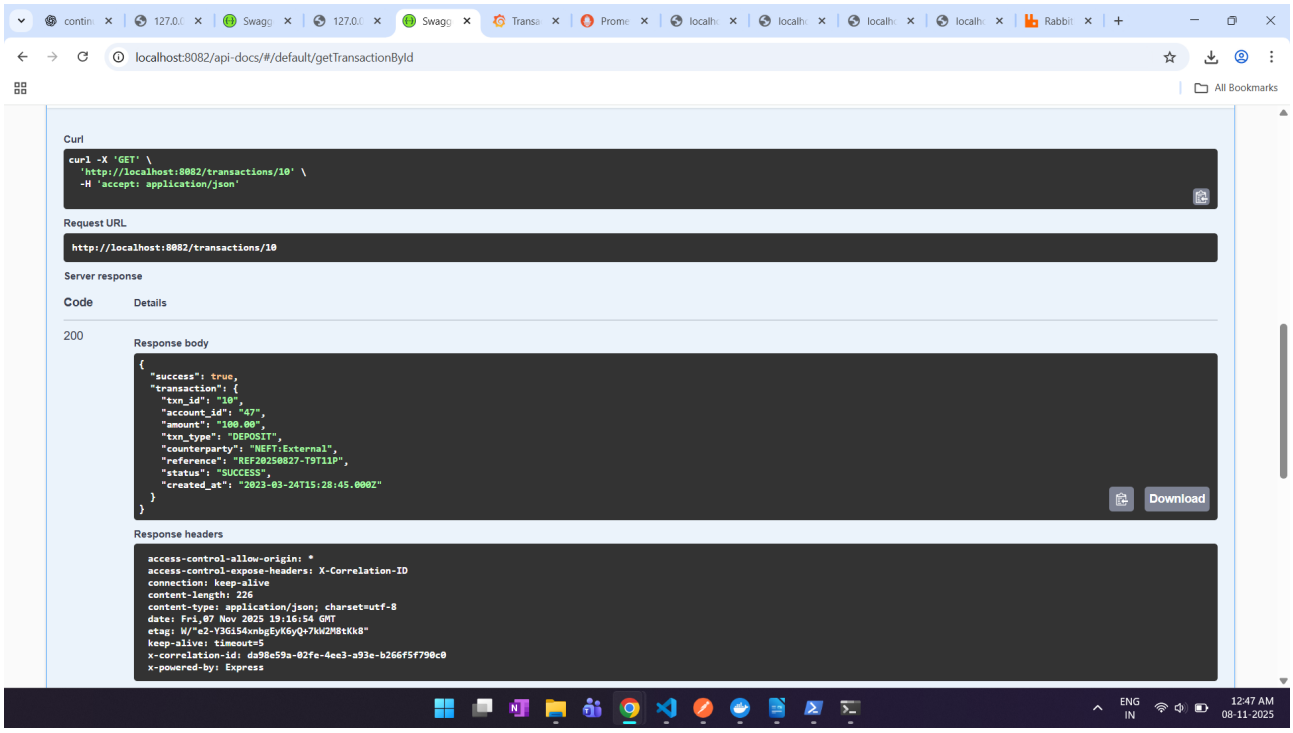


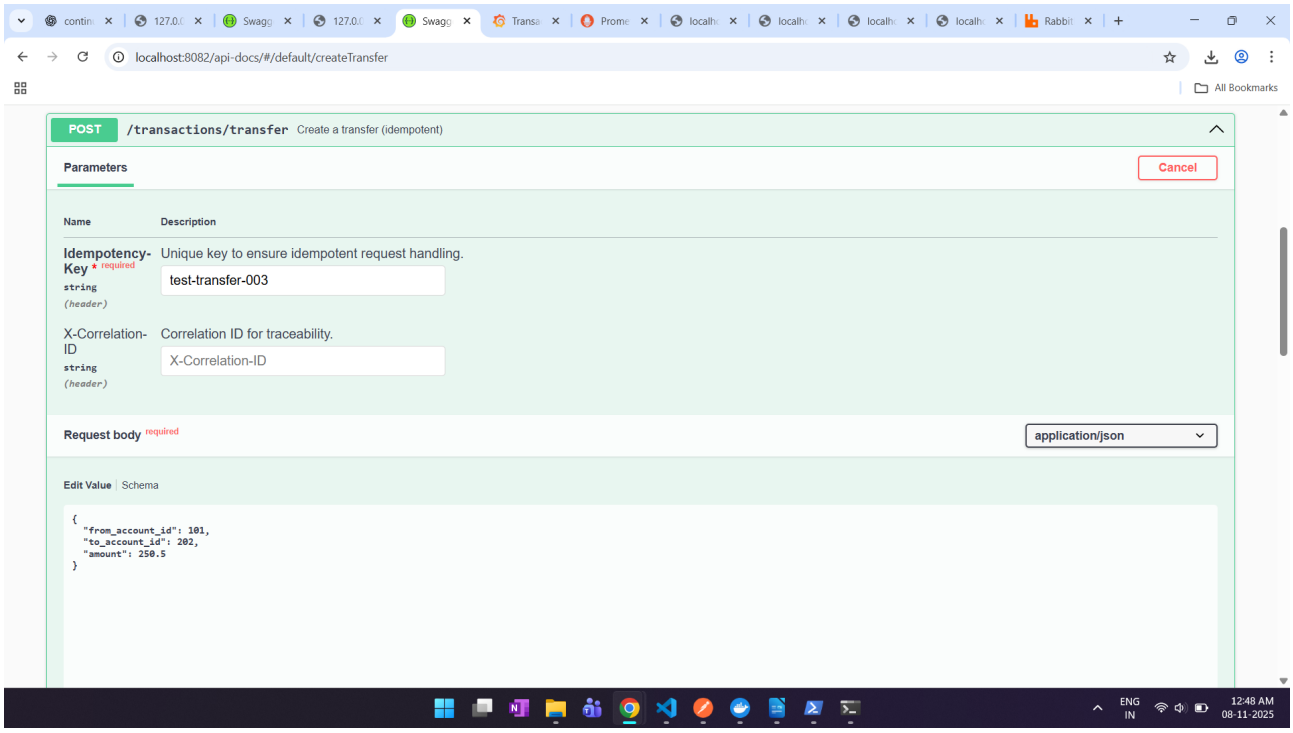
JATIN SAINI (2024TM93002) Repository: Github



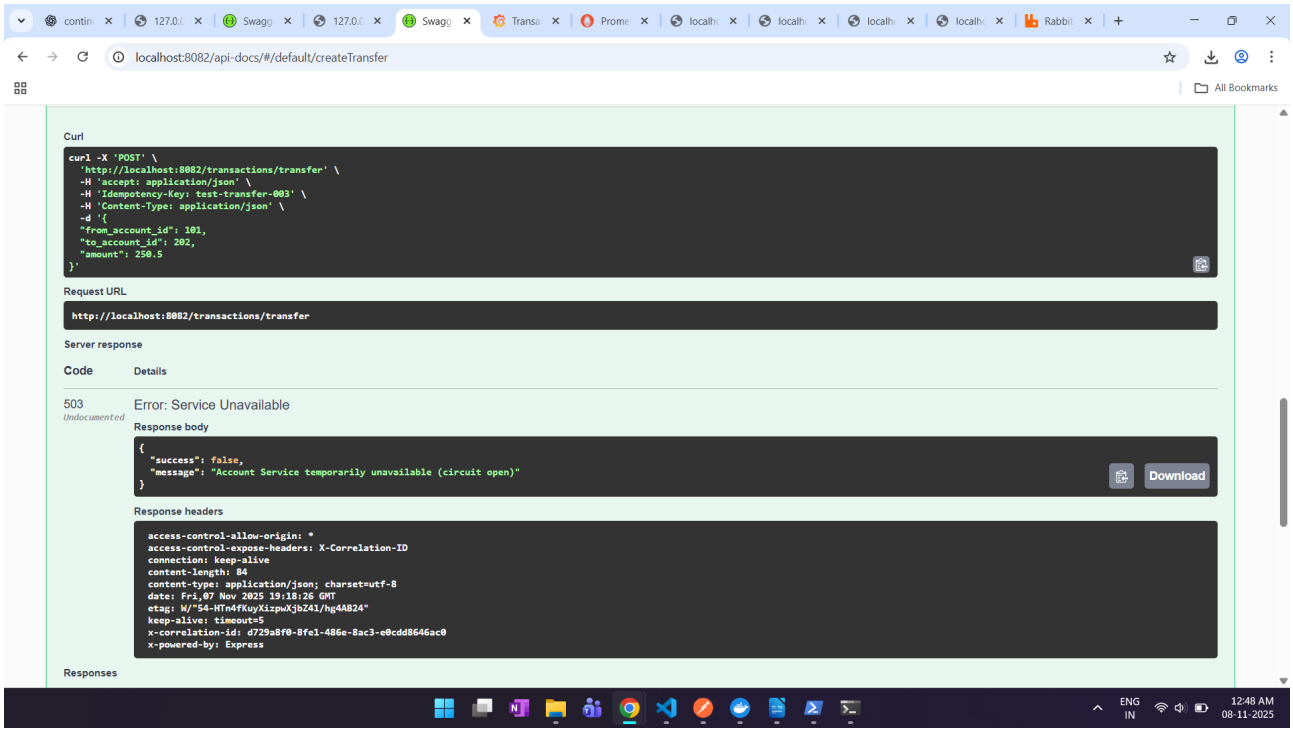


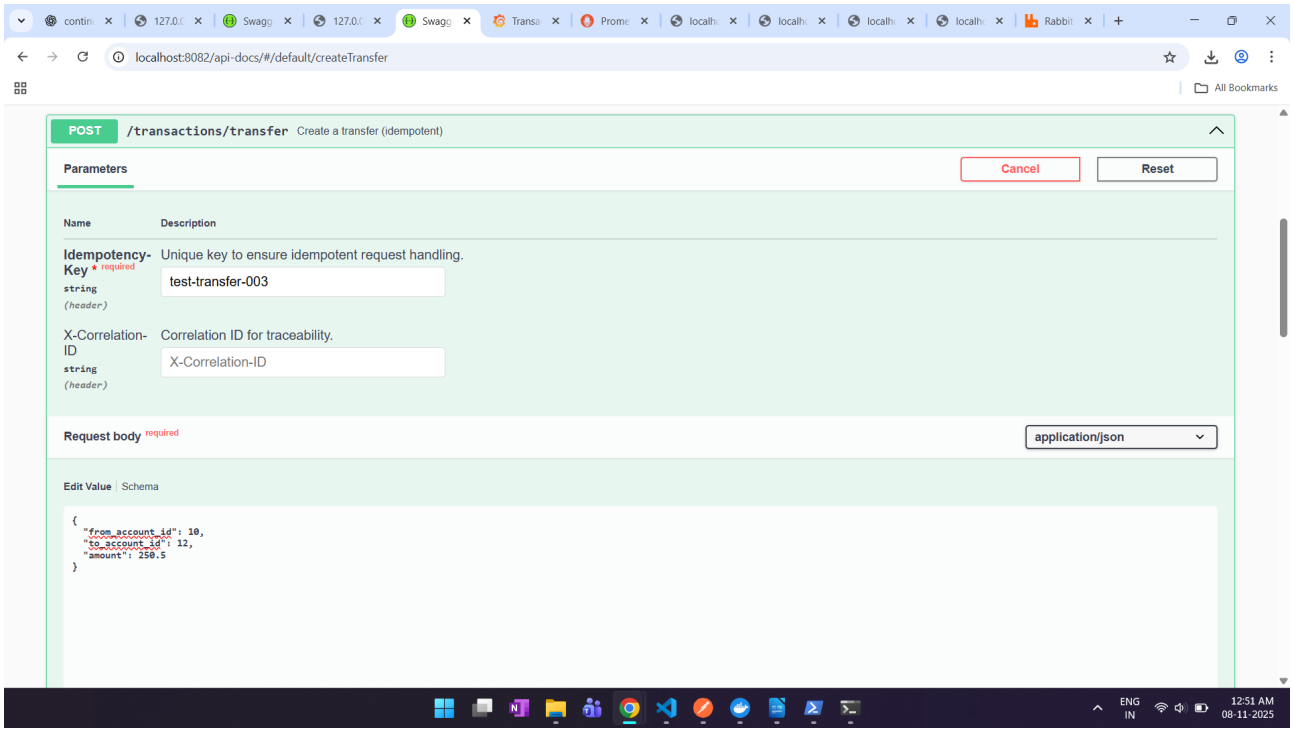
JATIN SAINI (2024TM93002) Repository: Github



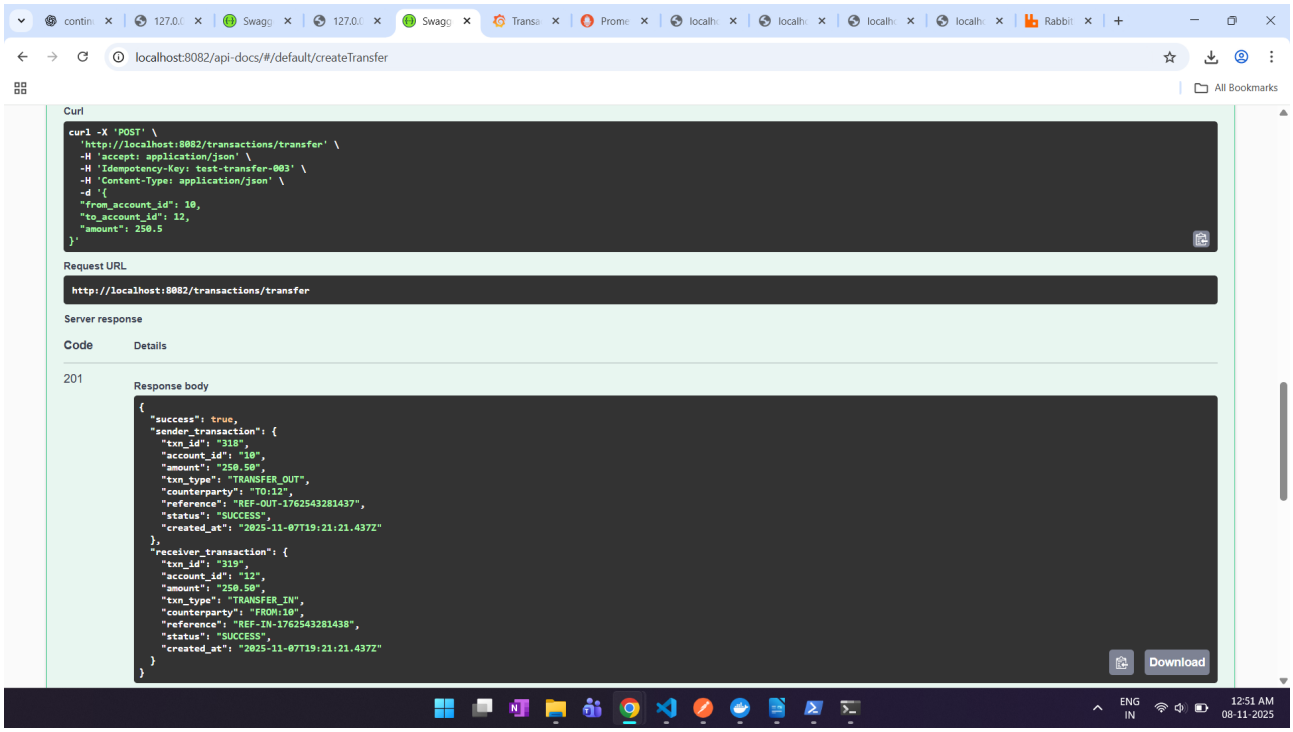


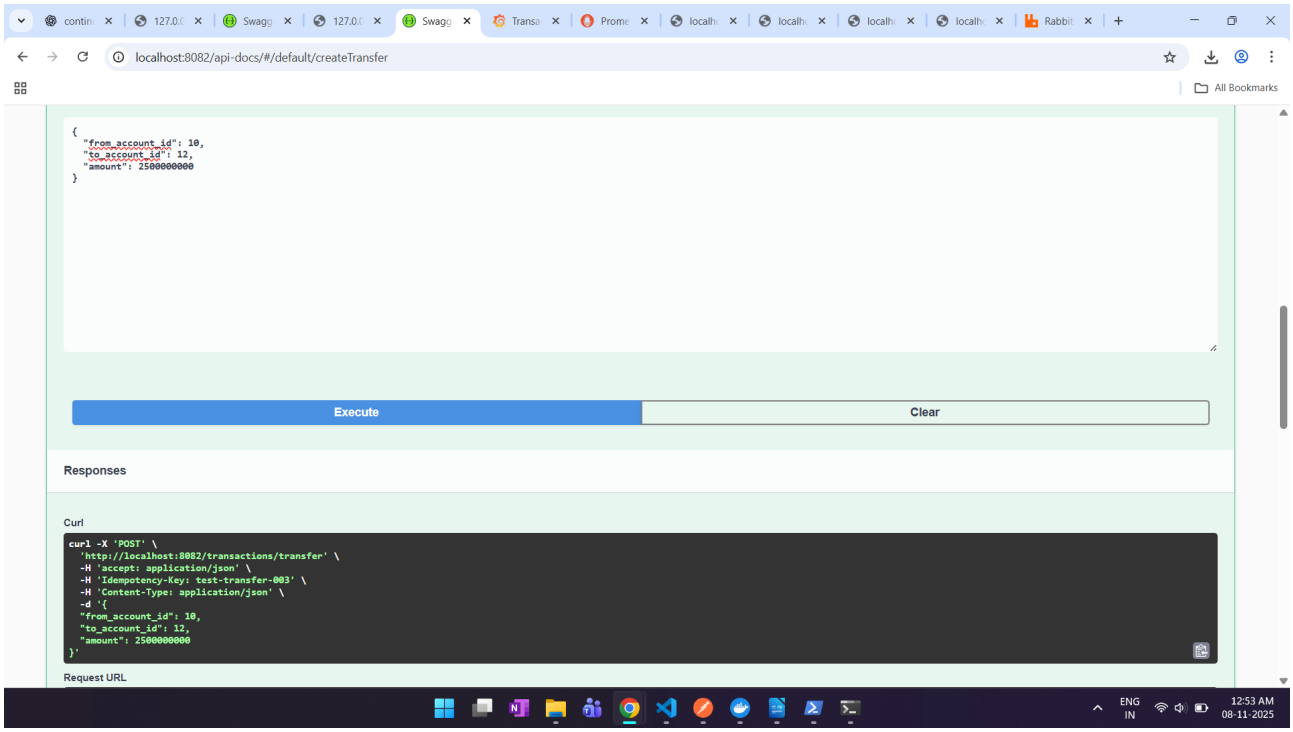
JATIN SAINI (2024TM93002) Repository: Github



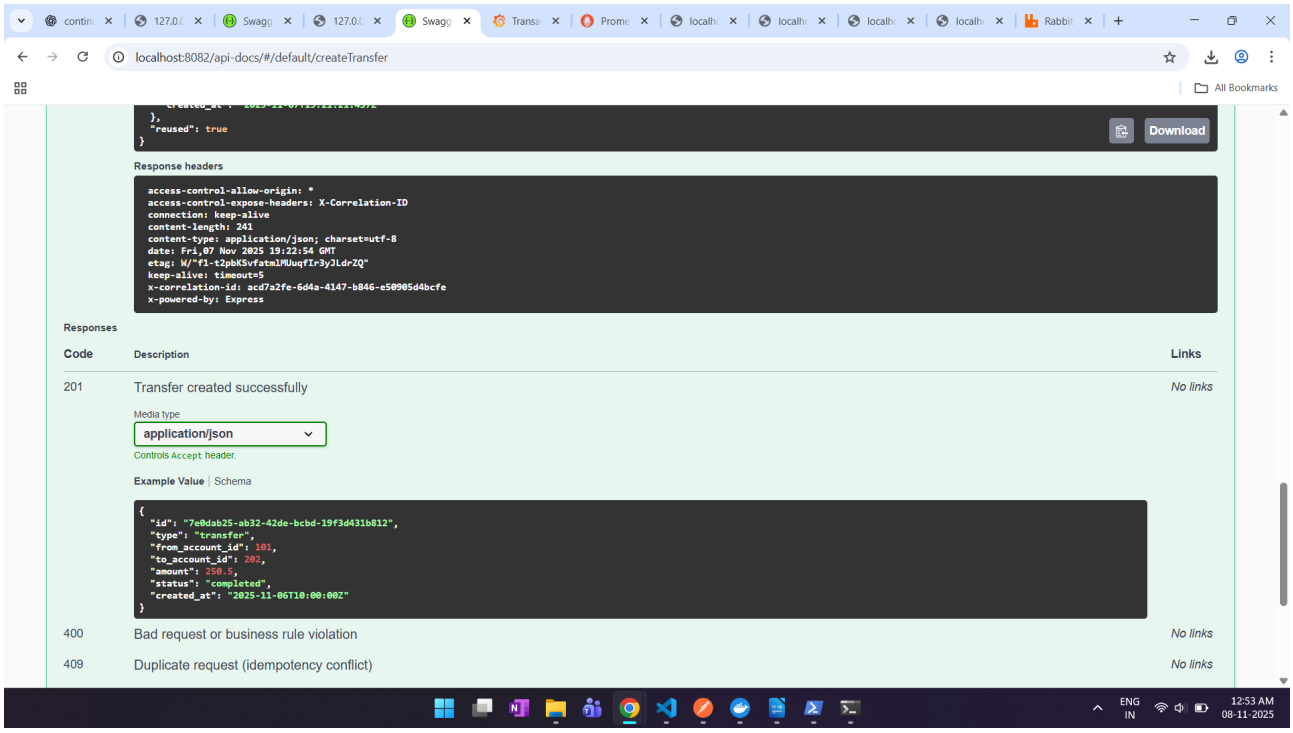


JATIN SAINI (2024TM93002) Repository: Github

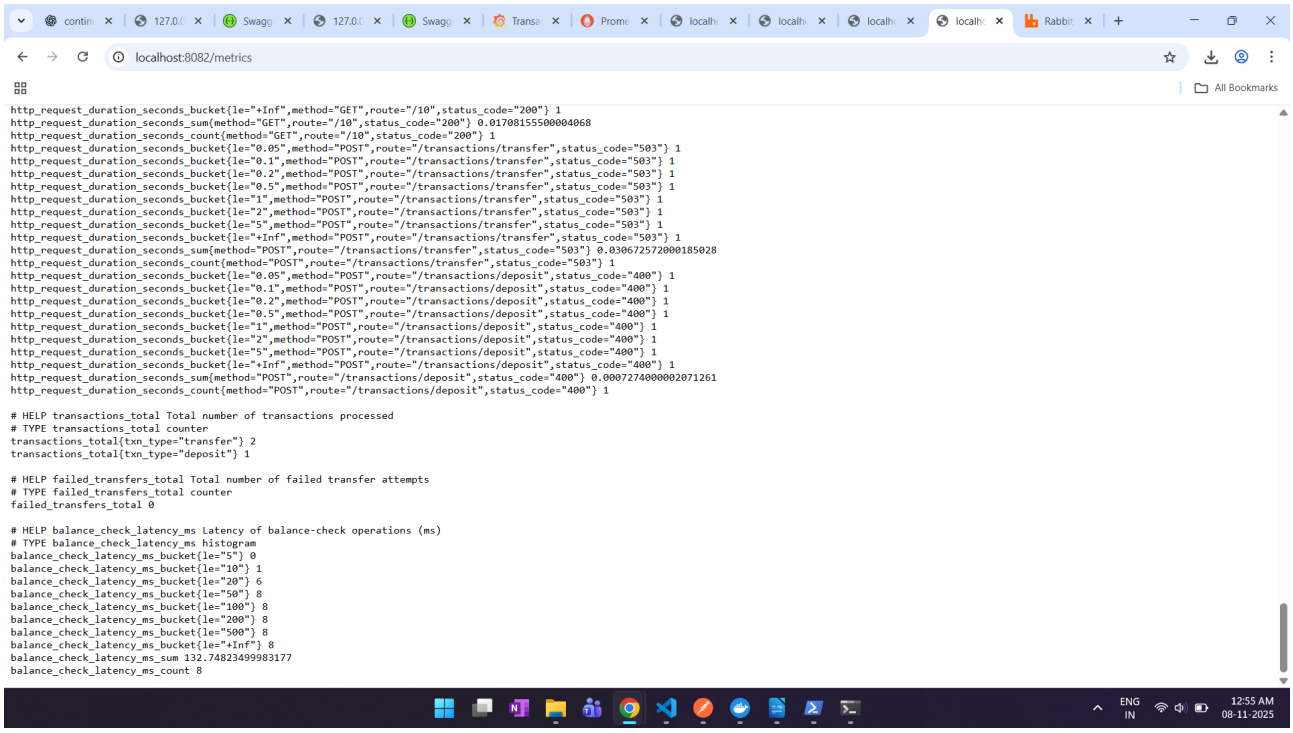




JATIN SAINI (2024TM93002) Repository: Github

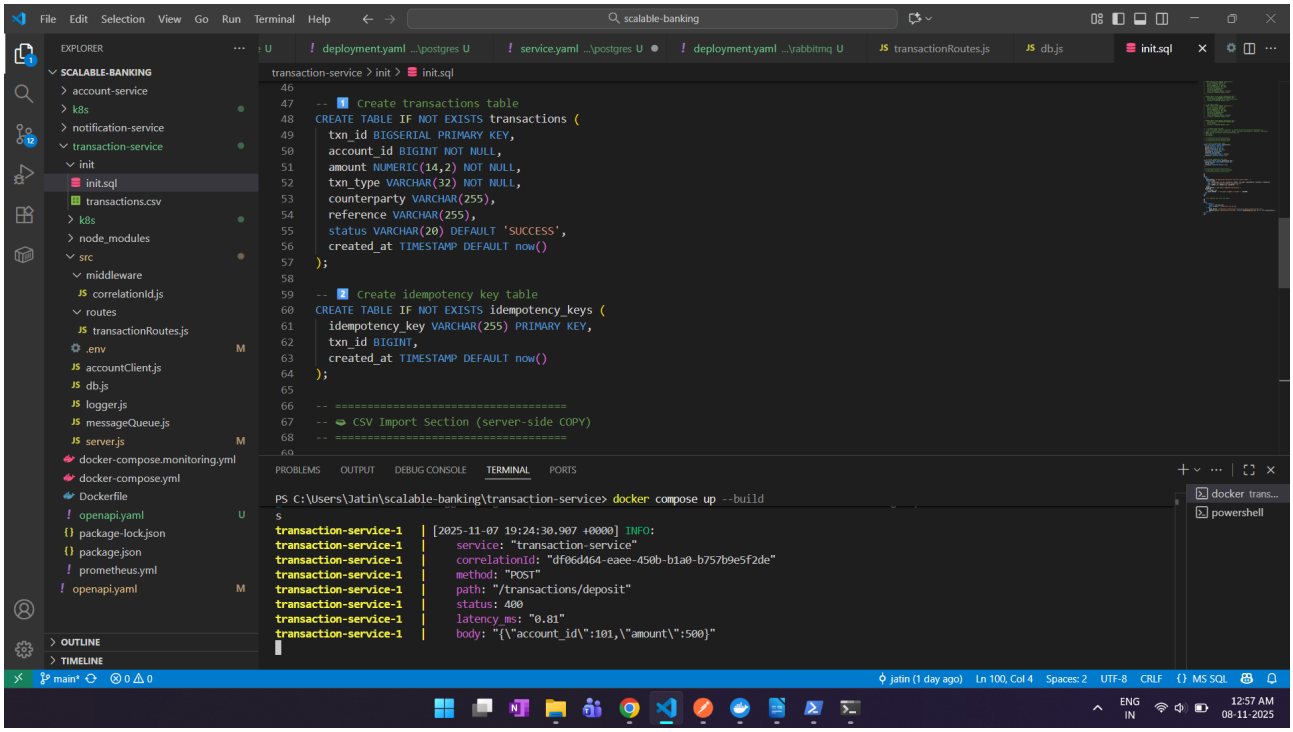


Metrics getting logged: no. of transaction failed, success and response time

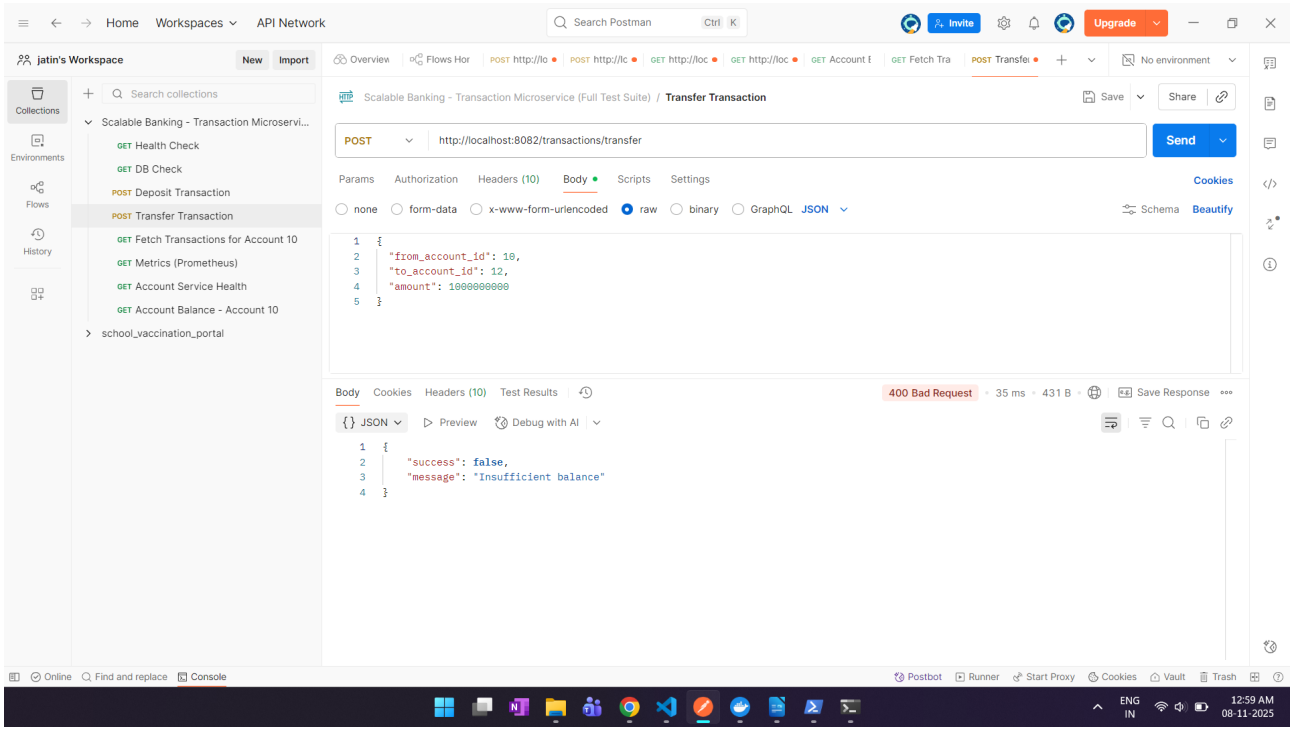


Same check be checked from docker as well in logs are in JSON format:

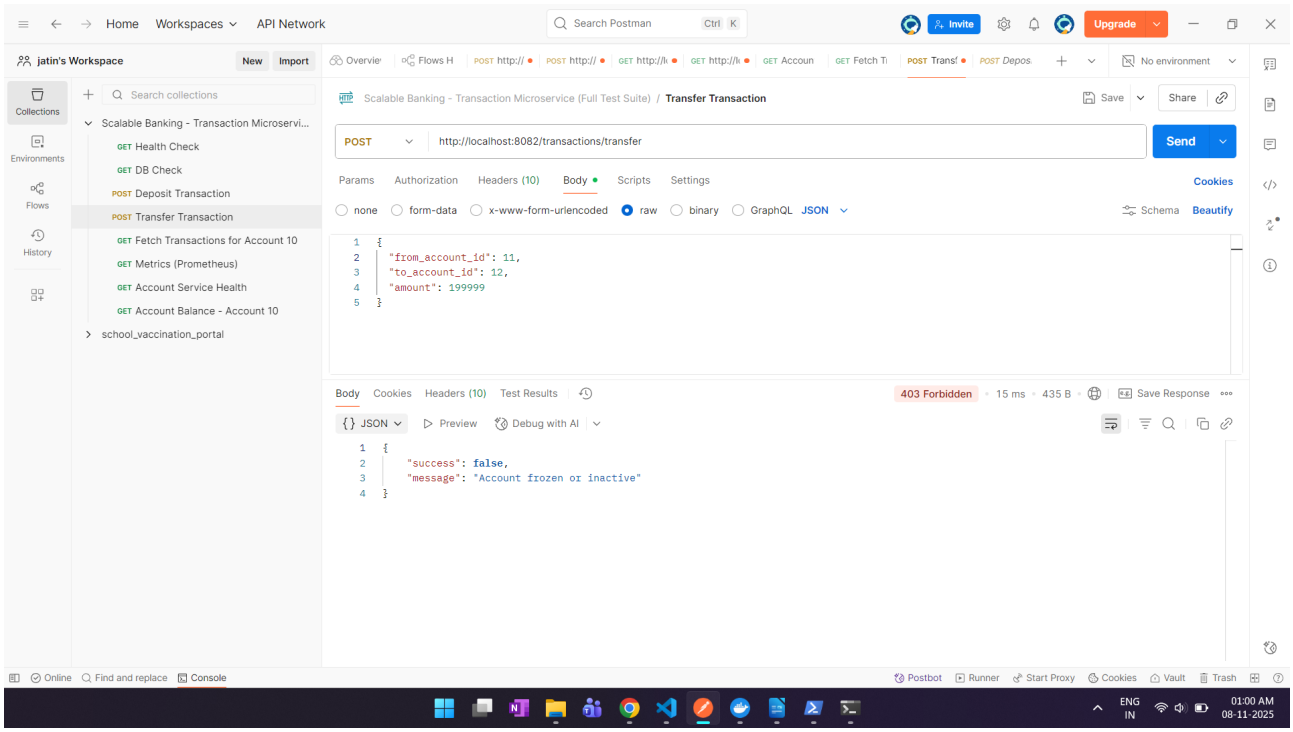
JATIN SAINI (2024TM93002) Repository: Github



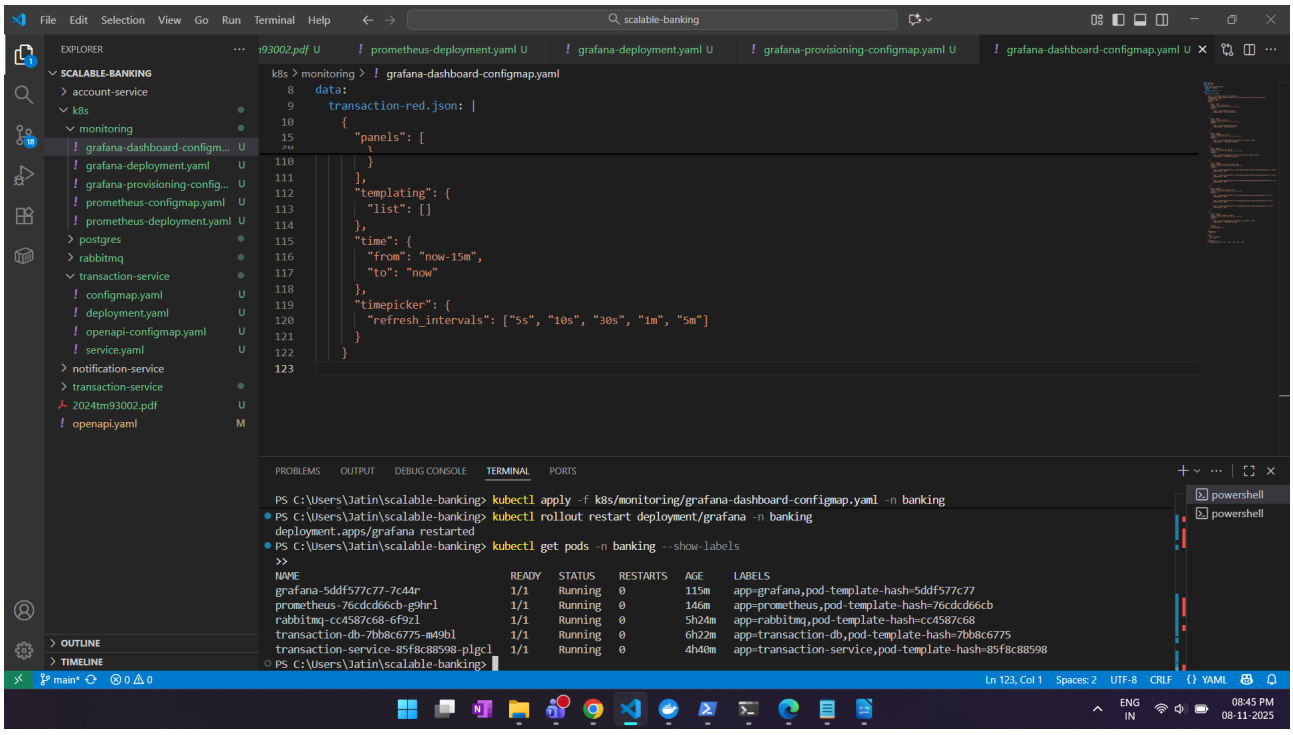
Business validation:



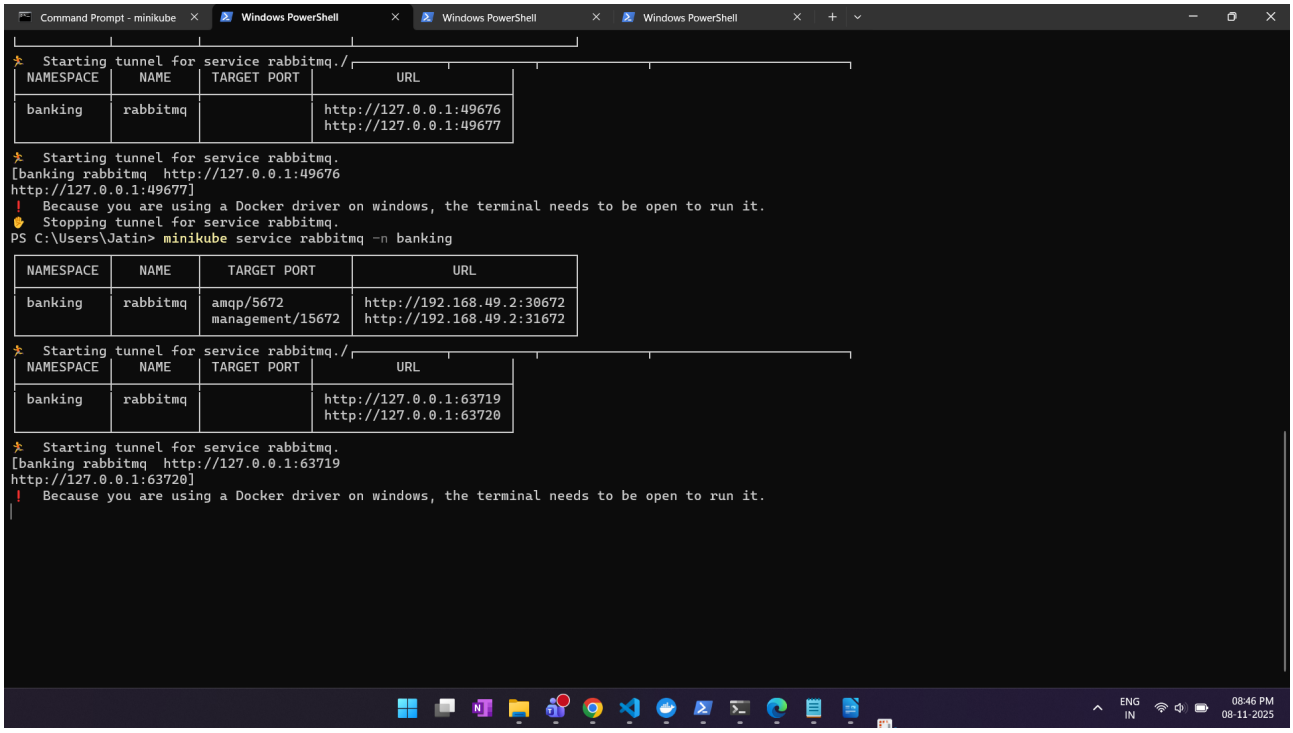
JATIN SAINI (2024TM93002) Repository: Github

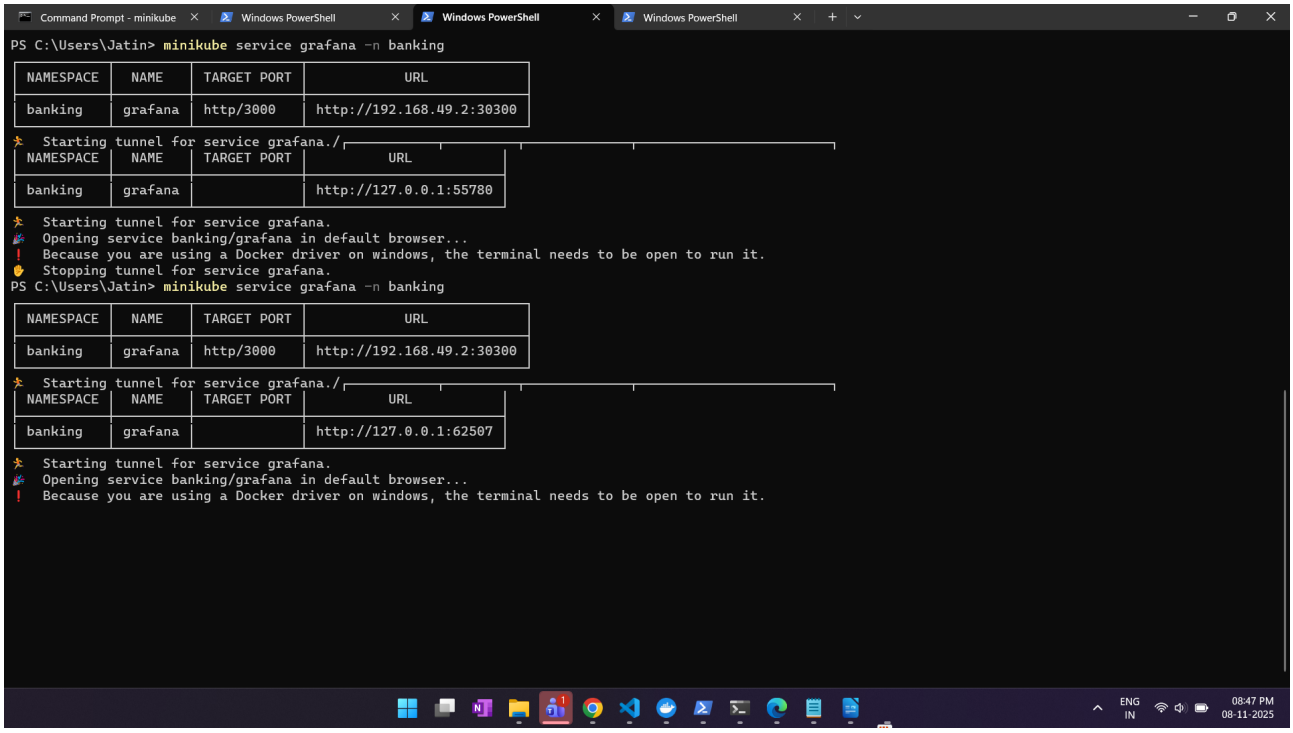


Minikube:

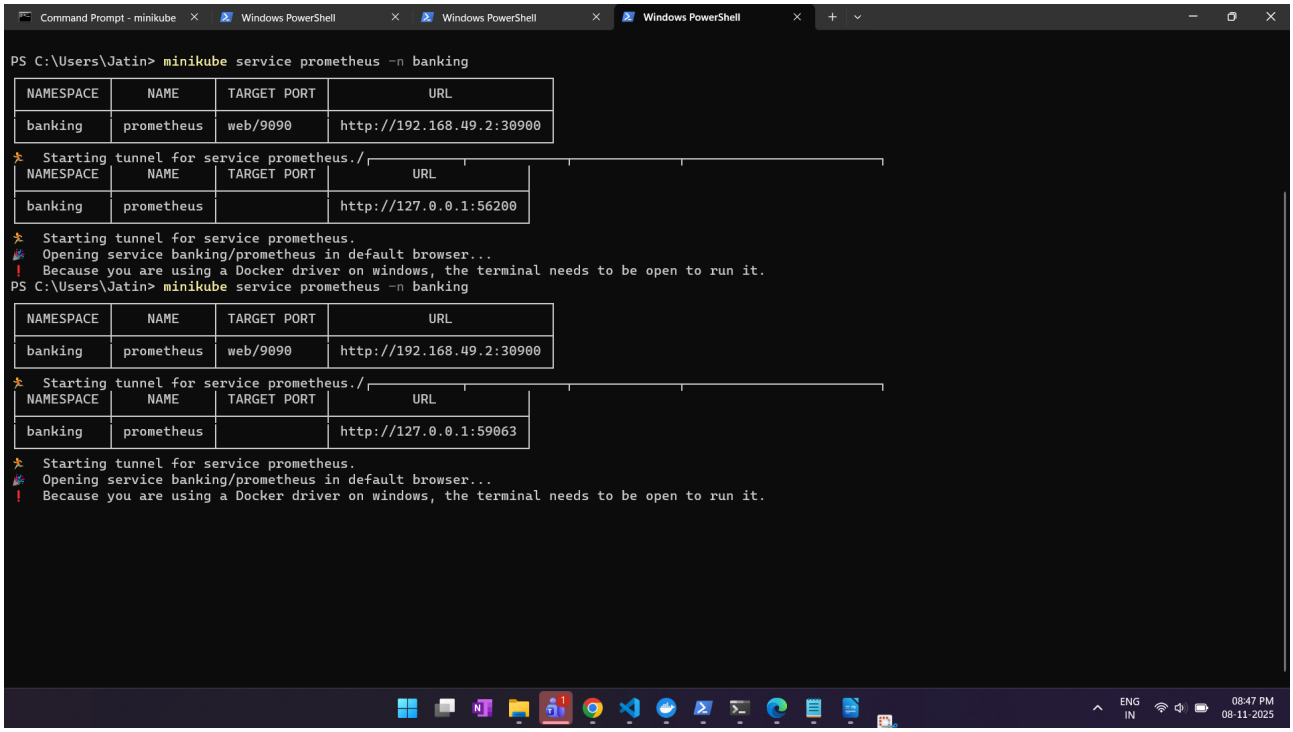


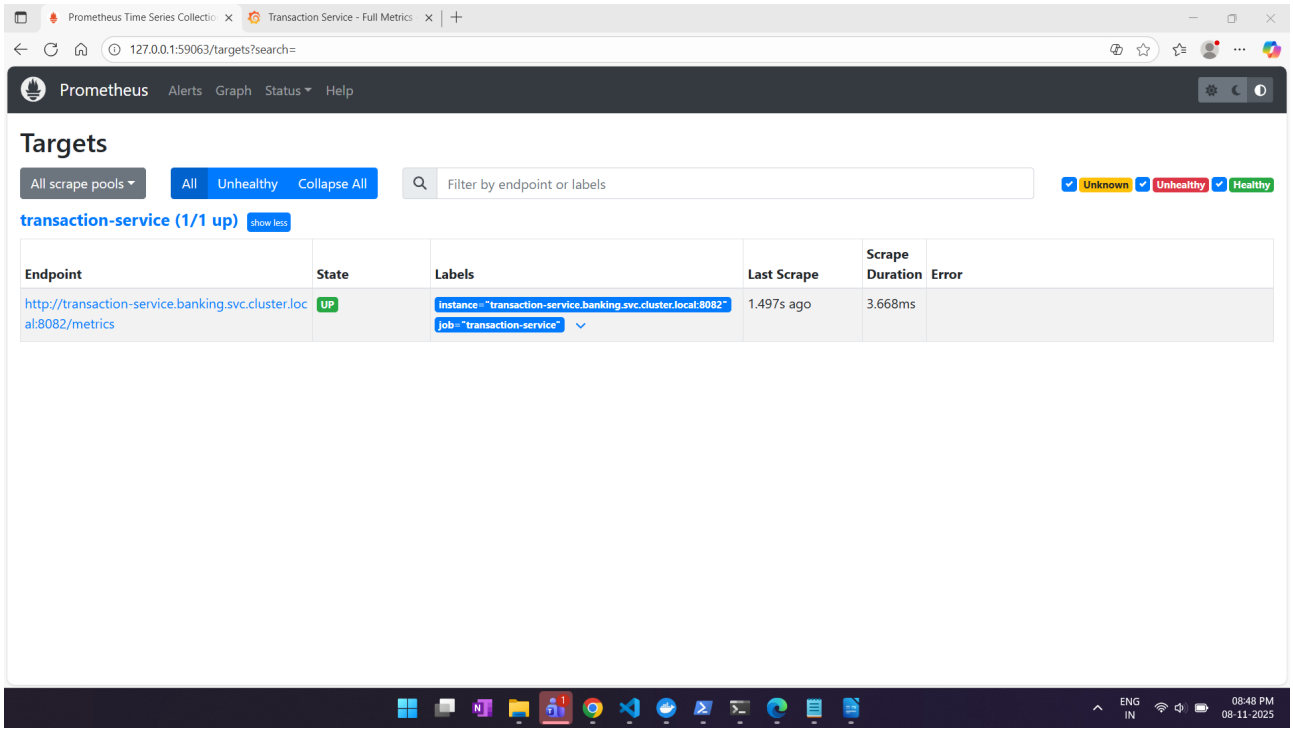
JATIN SAINI (2024TM93002) Repository: Github



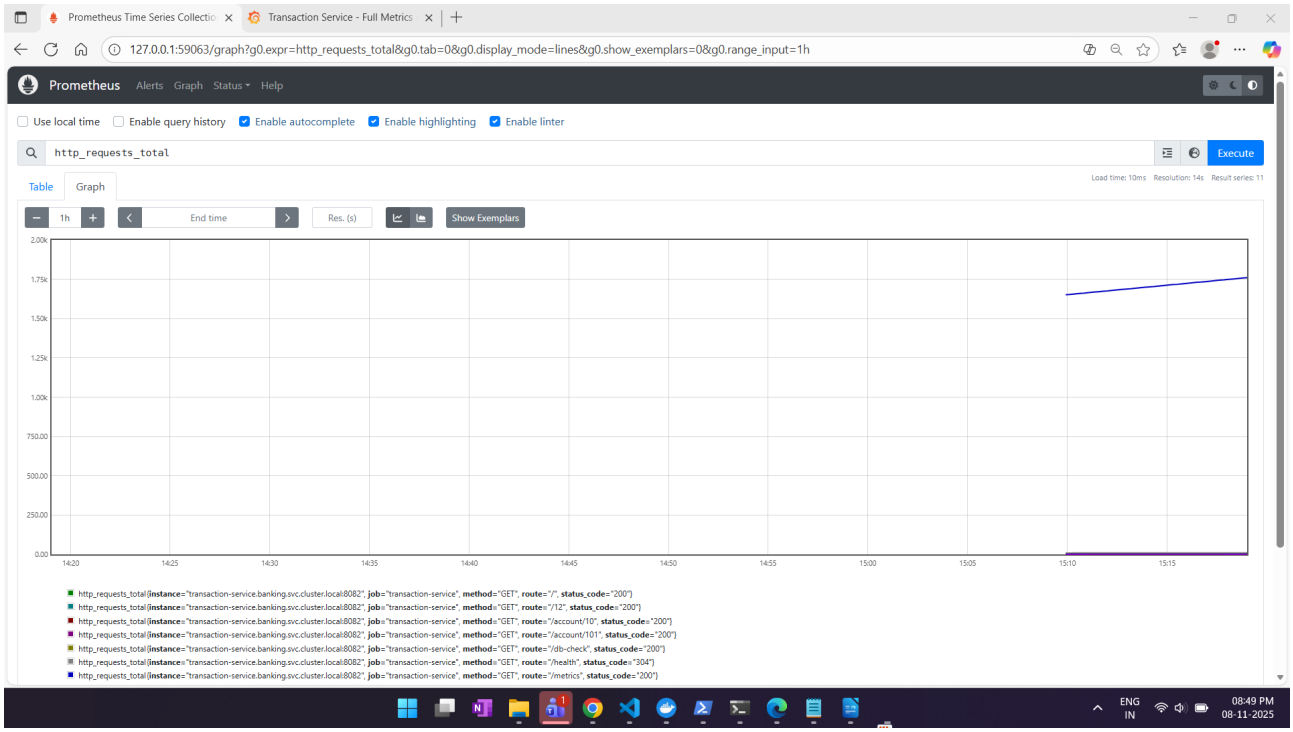


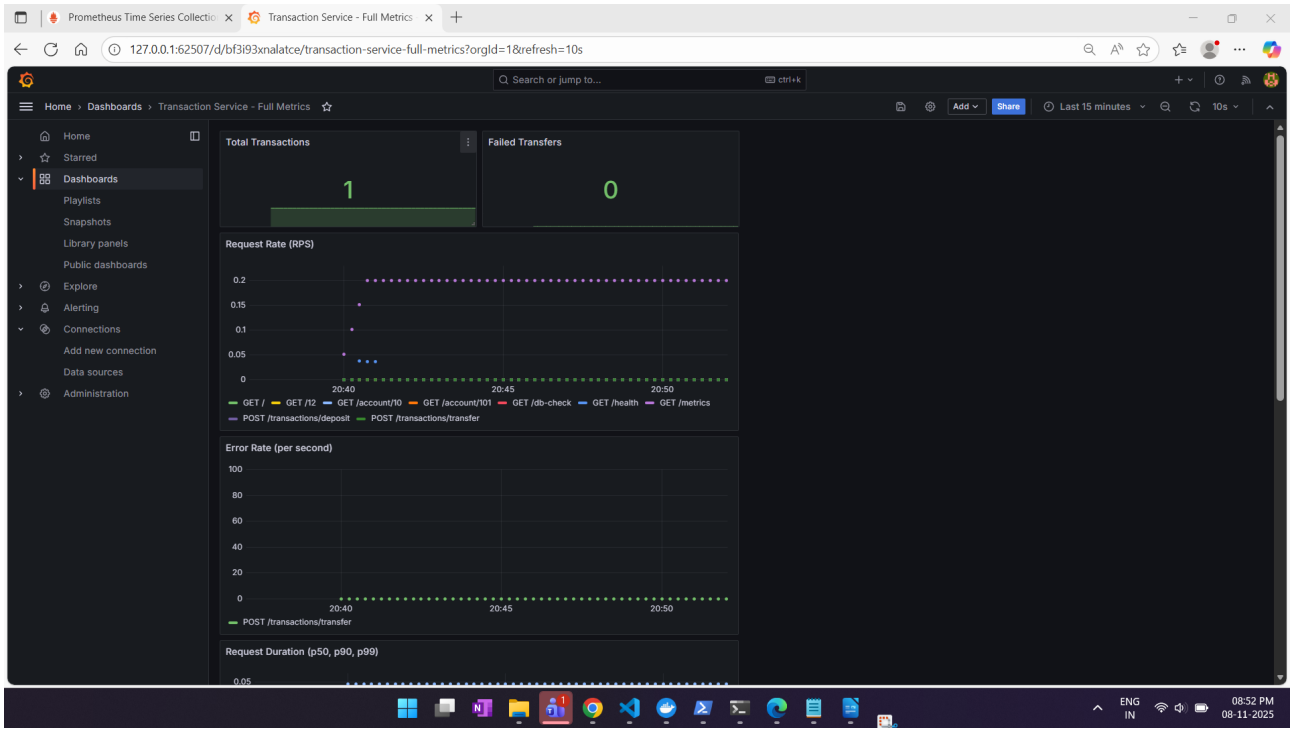
JATIN SAINI (2024TM93002) Repository: Github



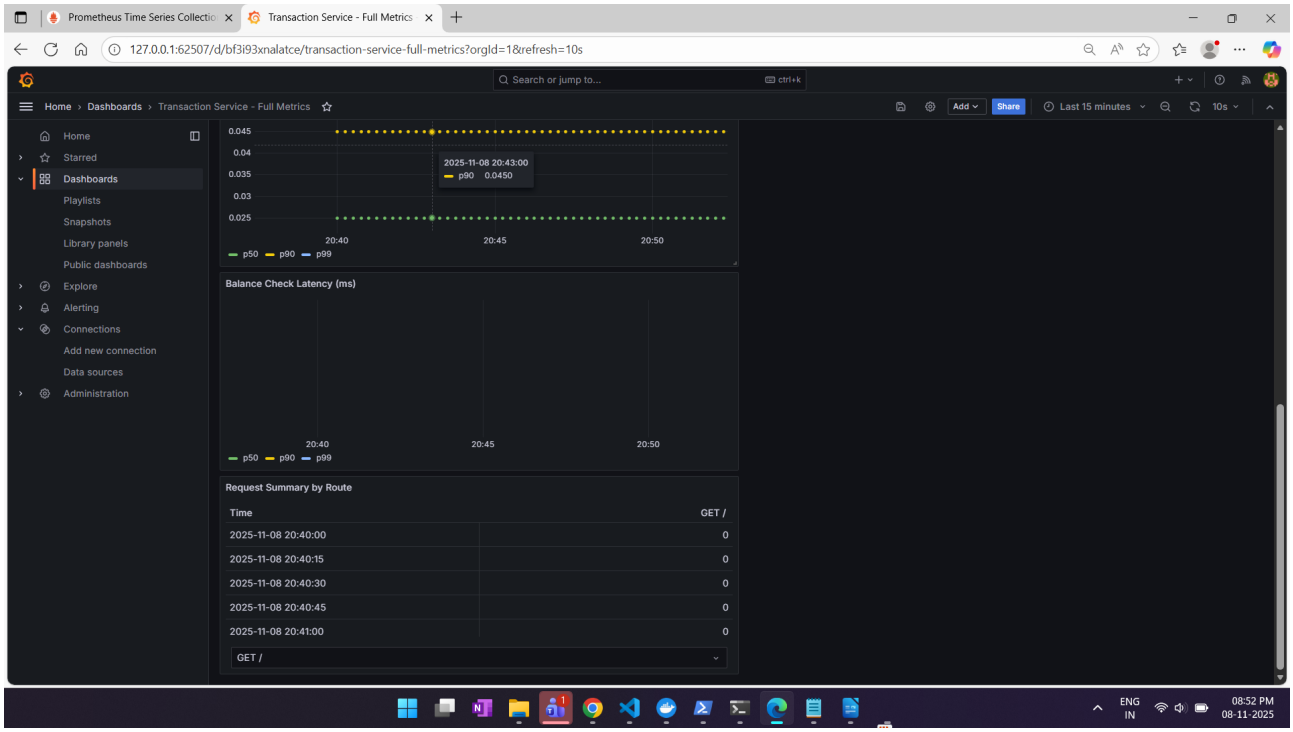


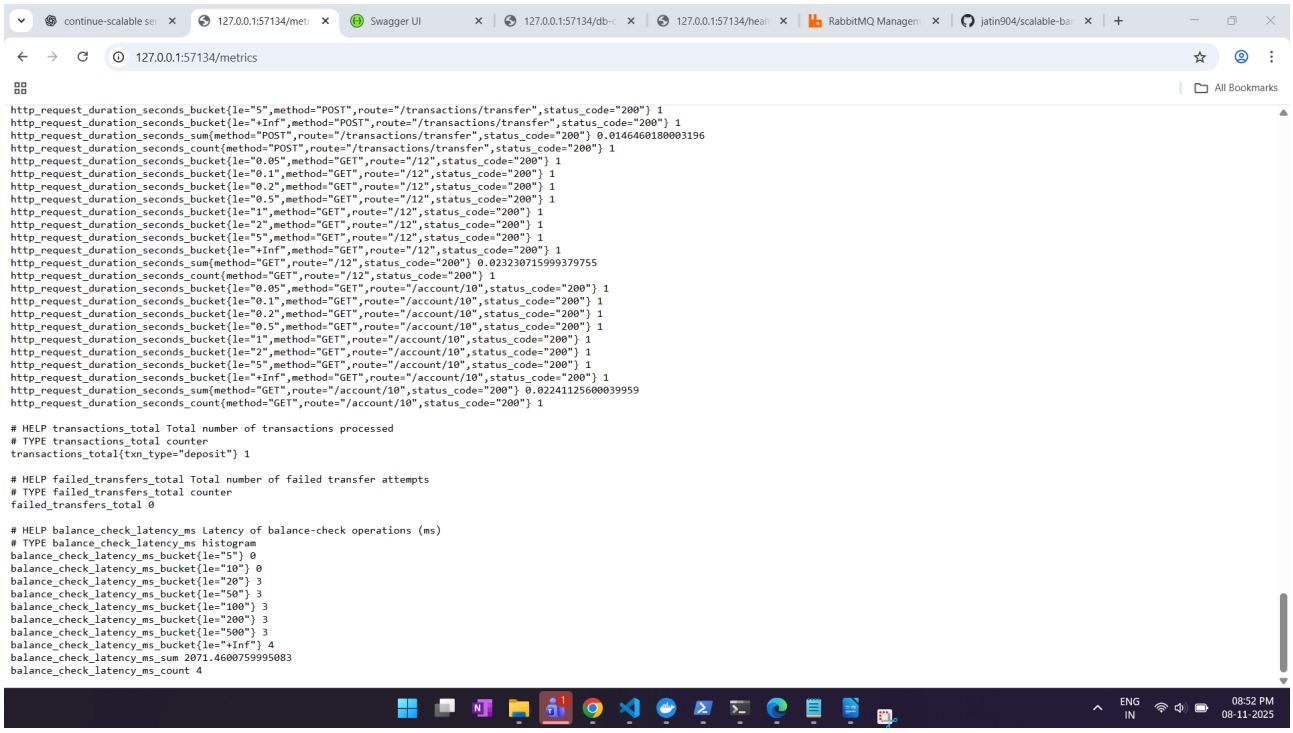
JATIN SAINI (2024TM93002) Repository: Github



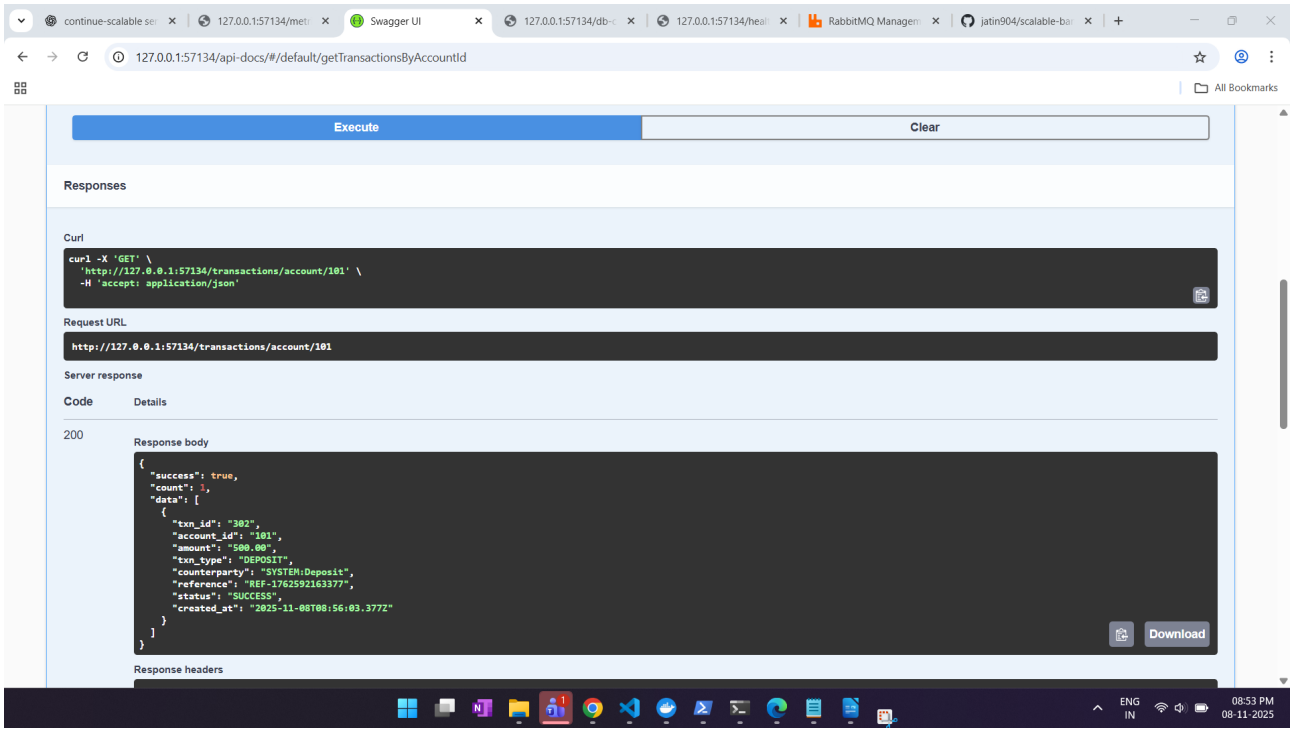


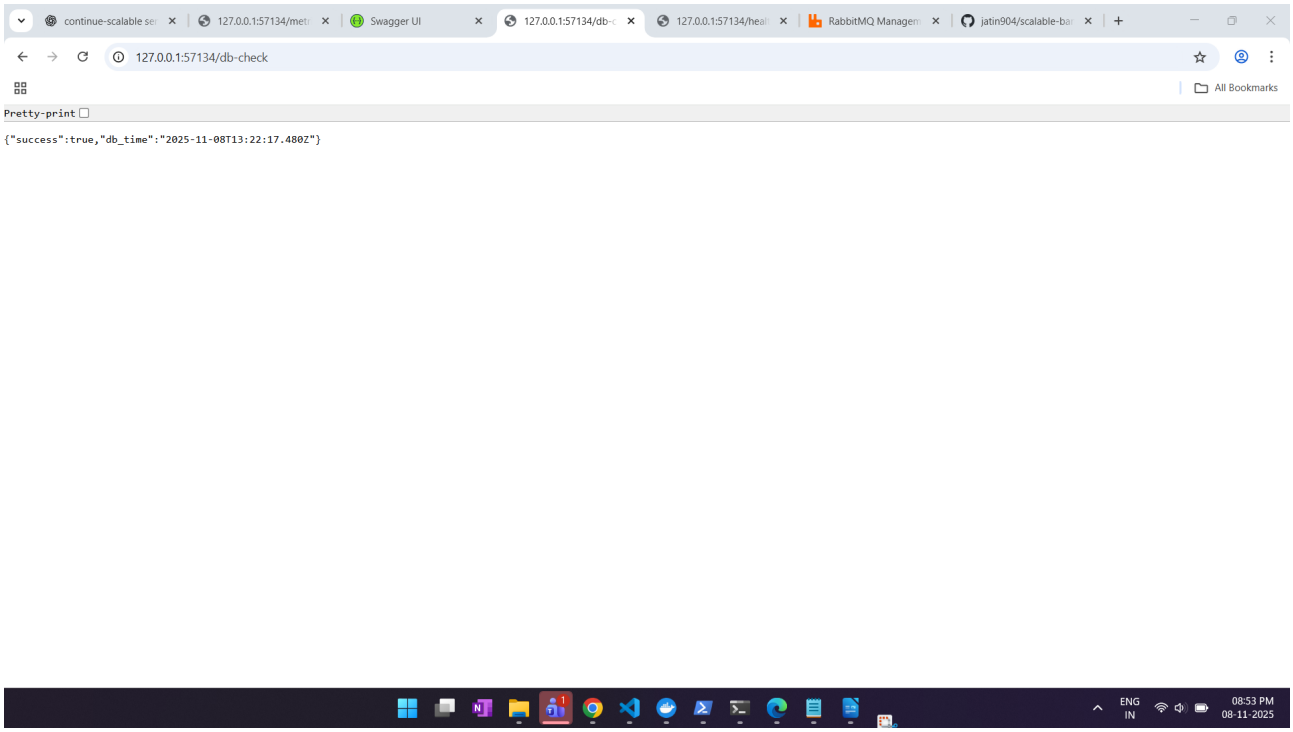
JATIN SAINI (2024TM93002) Repository: Github



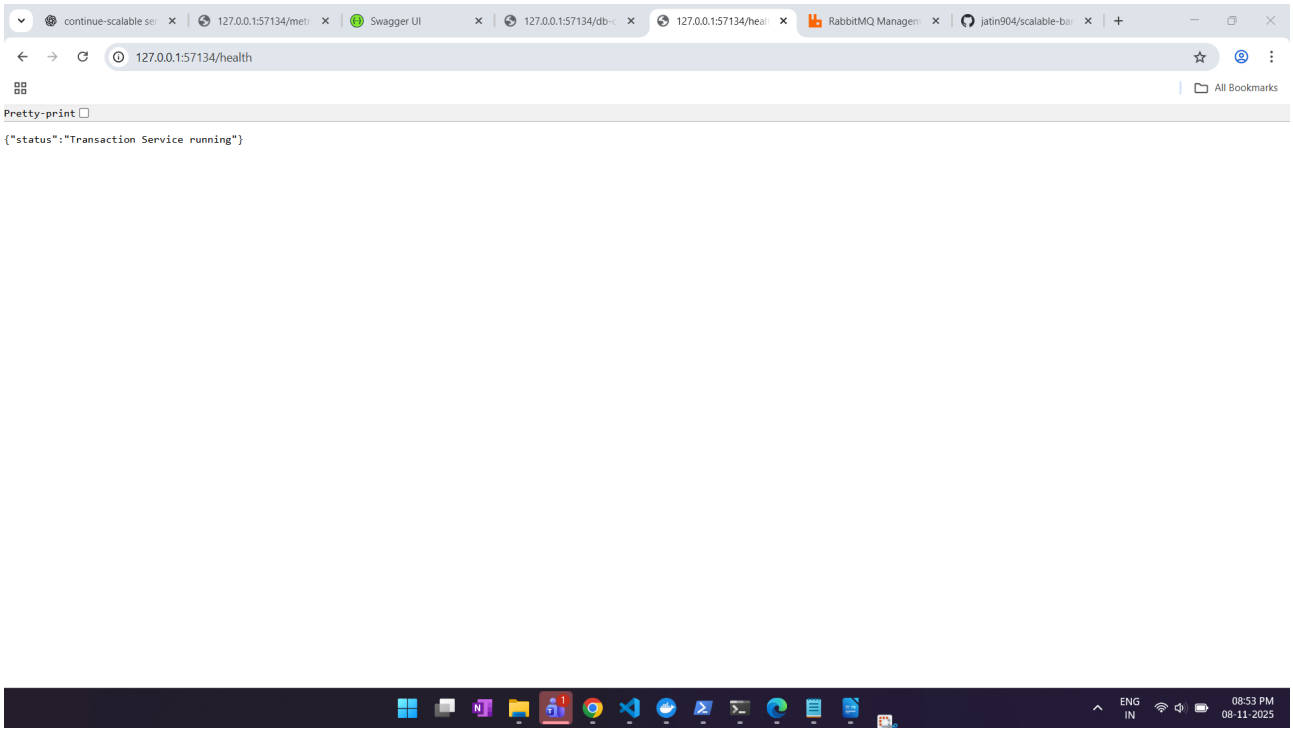


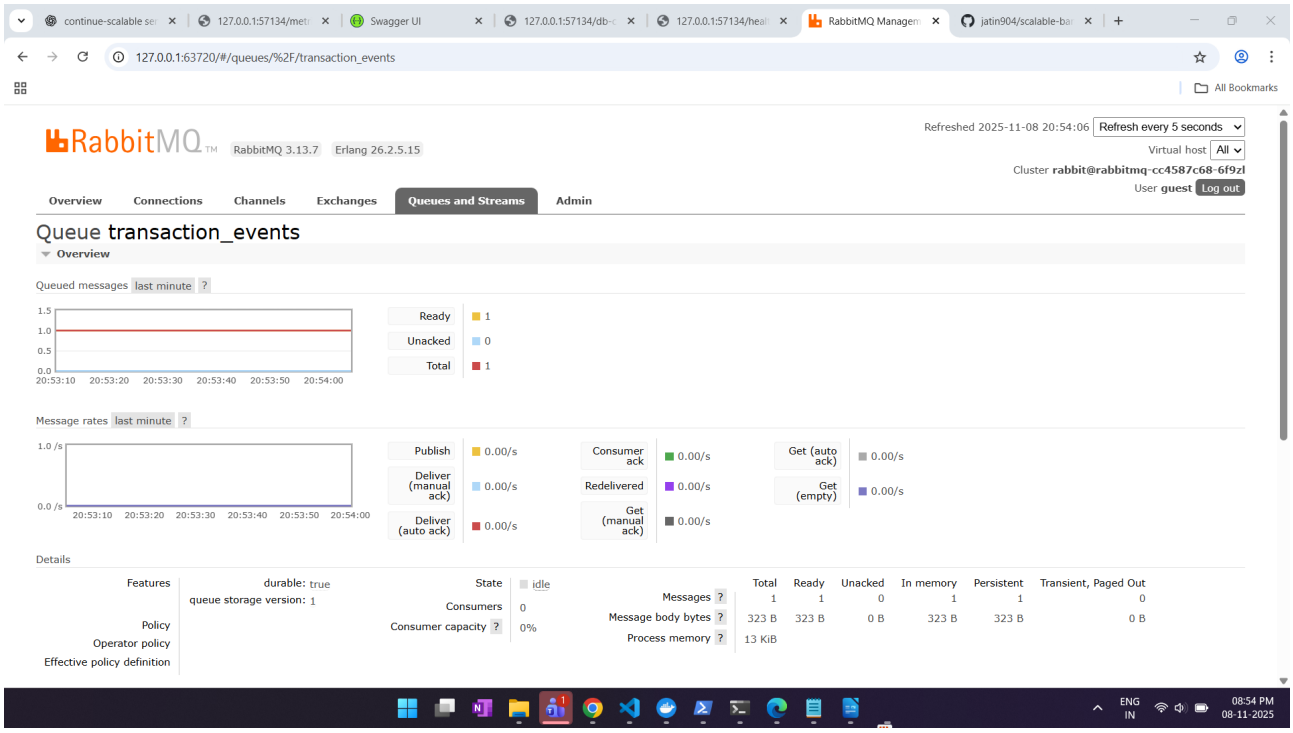
JATIN SAINI (2024TM93002) Repository: [Github]

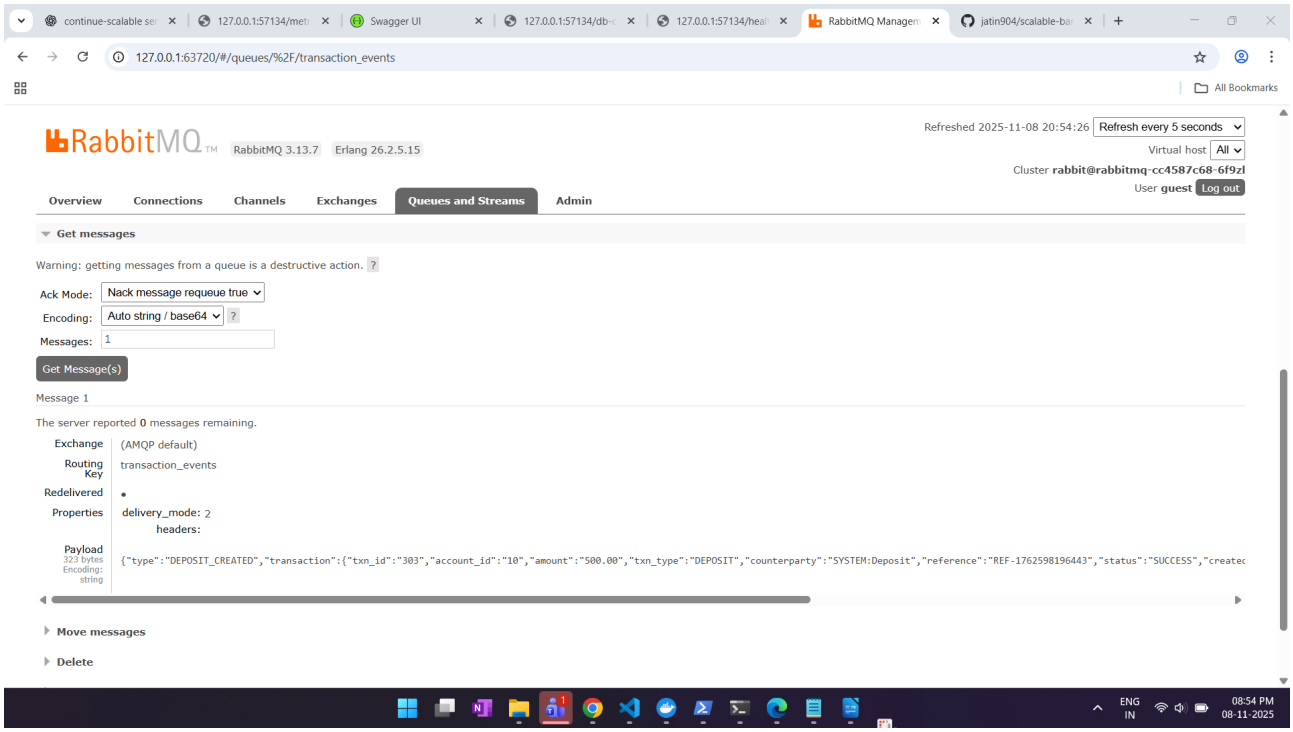




JATIN SAINI (2024TM93002) Repository: Github







Outcome

Transaction service run consistently in isolated containers, with working inter-service connectivity and event streaming. The container setup satisfies:

- Independent deployment and scaling per service
- Portable environment for local and Minikube deployment
- Verified health, metrics, and logging endpoints