ASSIGNMENT-2

STATEMENT:

Assignment 2	* Design a system that replicates a working of a banking system. It will contain multiple banks and branches, a customer can open a savings account or can take a loan at an interest rate of 12%, can view his account, transactions and loan details (such as loan pending, interest to be paid this year etc), a customer can perform various actions such as deposit and withdraw cash, repay loan, take a loan, etc
	* List all the features
	* Design a database ER diagram
	* list all the APIs
	* Implement all APIs

To do this assignment first i need to determine what entities are there in this ->

- **Bank**
- **Branch**
- **Customer**
- **Account**
- **Transaction**
- **Loan**
- **LoanPayment**

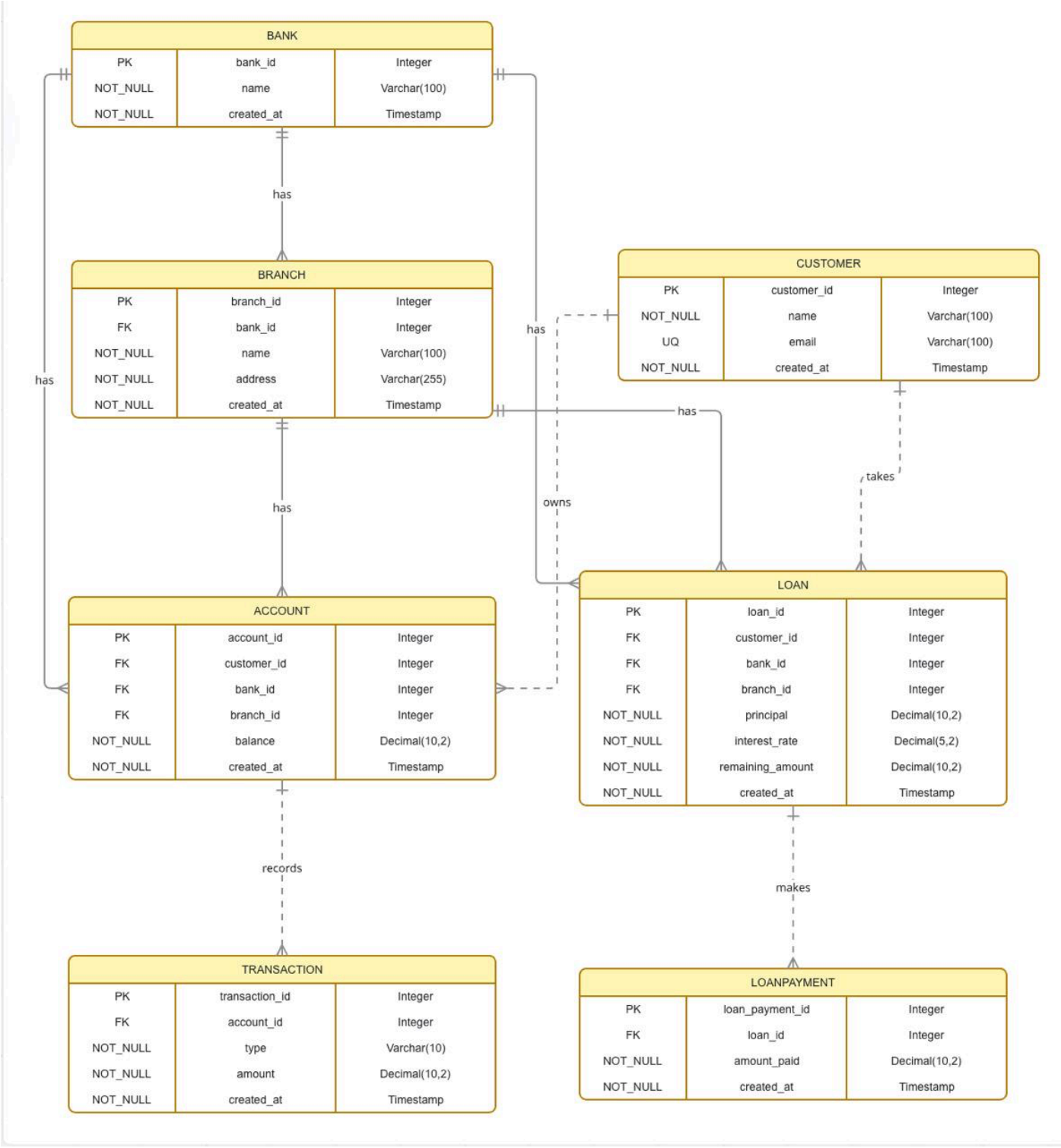
Now after identifying these entities I need to see how these are related also from question what i think is specified:

Multiple Banks —----------> Need API,Databse connection
Multiple Branches —----------> Need API, Database connection
Customer opens Savings Account—--------> API,Databse Connection, Business Logic
Loan Take with interest rate 12% —---------->API,Databse Connection, Business Logic
Customer can view his account —----------> API,Databse Connection, Business Logic
Customer can see his transactions—--------> API,Databse Connection, Business Logic
Customer can see his loan details —---------> API,Databse Connection, Business Logic

So for now i see relationships so now I DRAW ER diagram:

Here is what i thought of relationships:

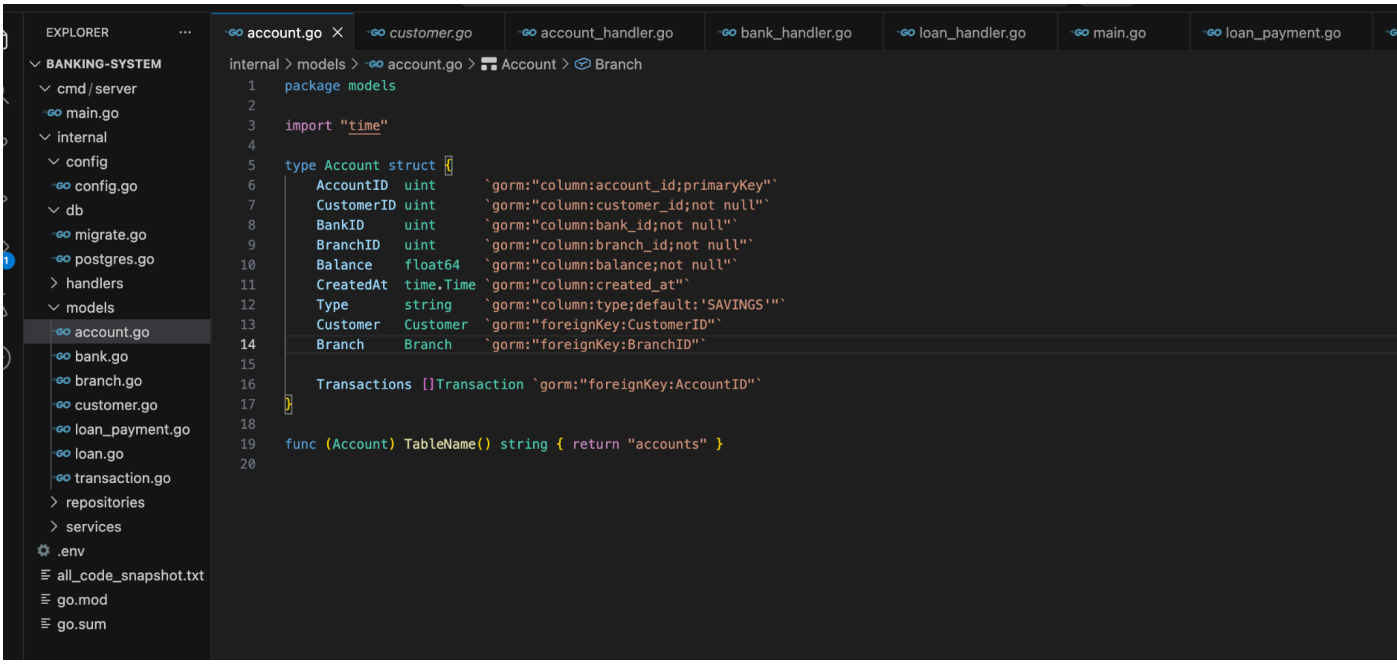
- One Bank has many Branches (1:N)
- One Bank has many Accounts (1:N)
- One Bank has many Loans (1:N)
- One Branch has many Accounts (1:N)
- One Branch has many Loans (1:N)
- One Customer can have many Accounts, and each Account belongs to one Customer (1:N)
- One Customer can have many Loans, and each Loan belongs to one Customer (1:N)
- One Account has many Transactions (1:N)
- One Loan has many LoanPayments (1:N)
- Customer to Bank and Customer to Branch relationships are many-to-many and are achieved indirectly through the Account and Loan entities.



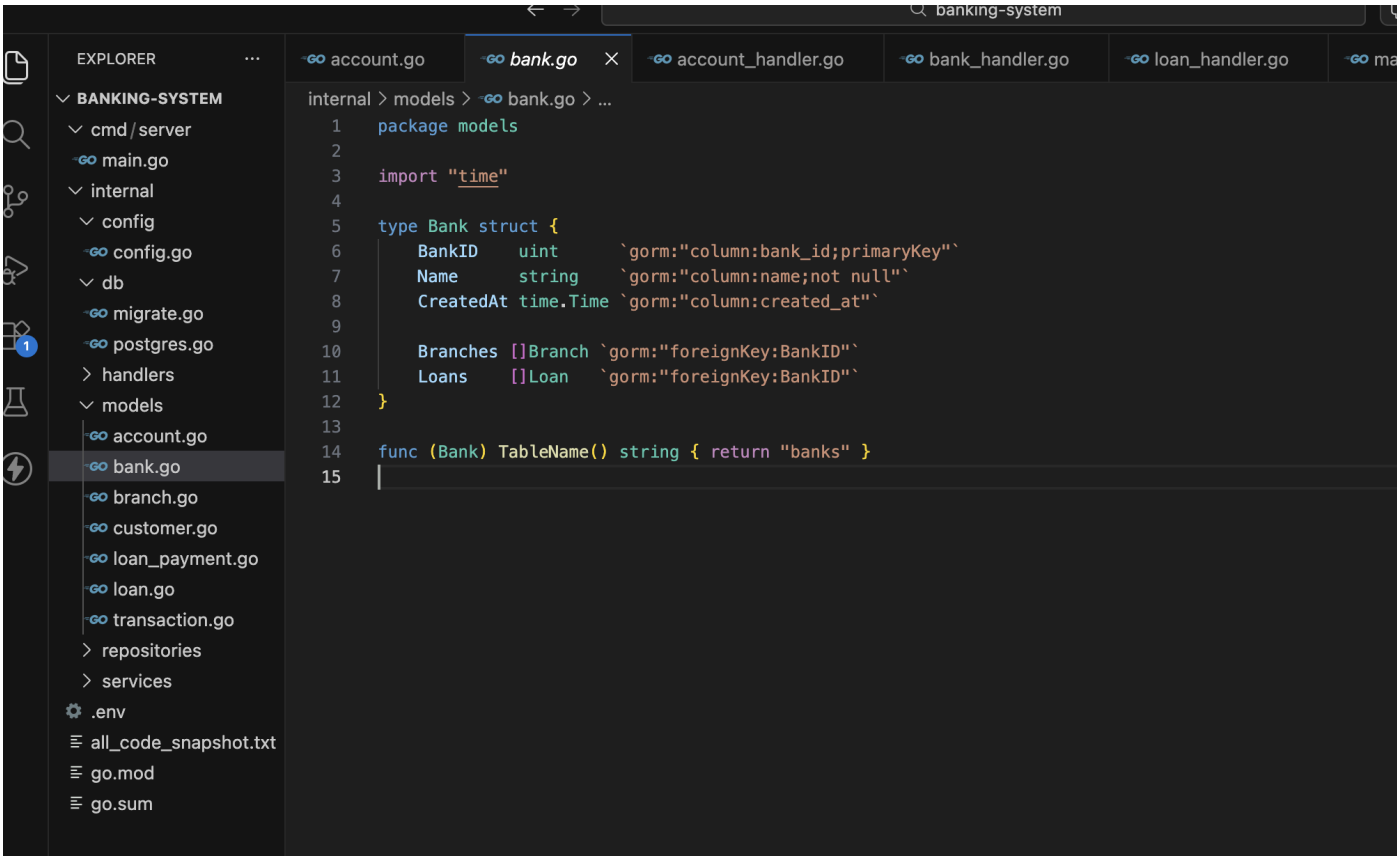
Now after Drawing Er diagram I need to create models that are structs in go and use GORM library so that we can establish GORM->Postgres Connection and tables get autofilled whenever user enters json data after jsonbinding in these structs and from these structs to postgres tables

Now we need 7 GORM models as we have 7 entities 7 postgres tables to fill:

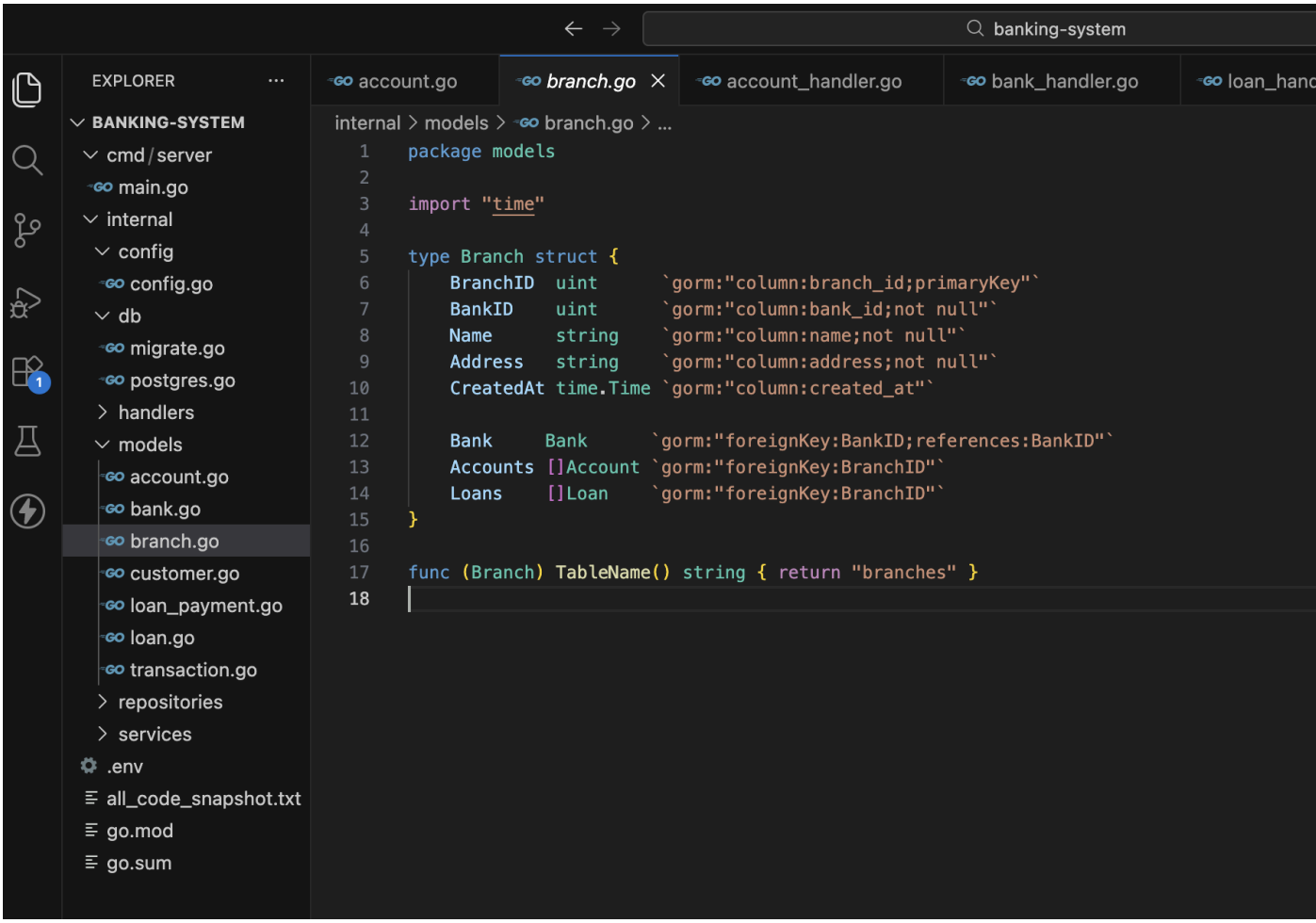
1)Account.go->



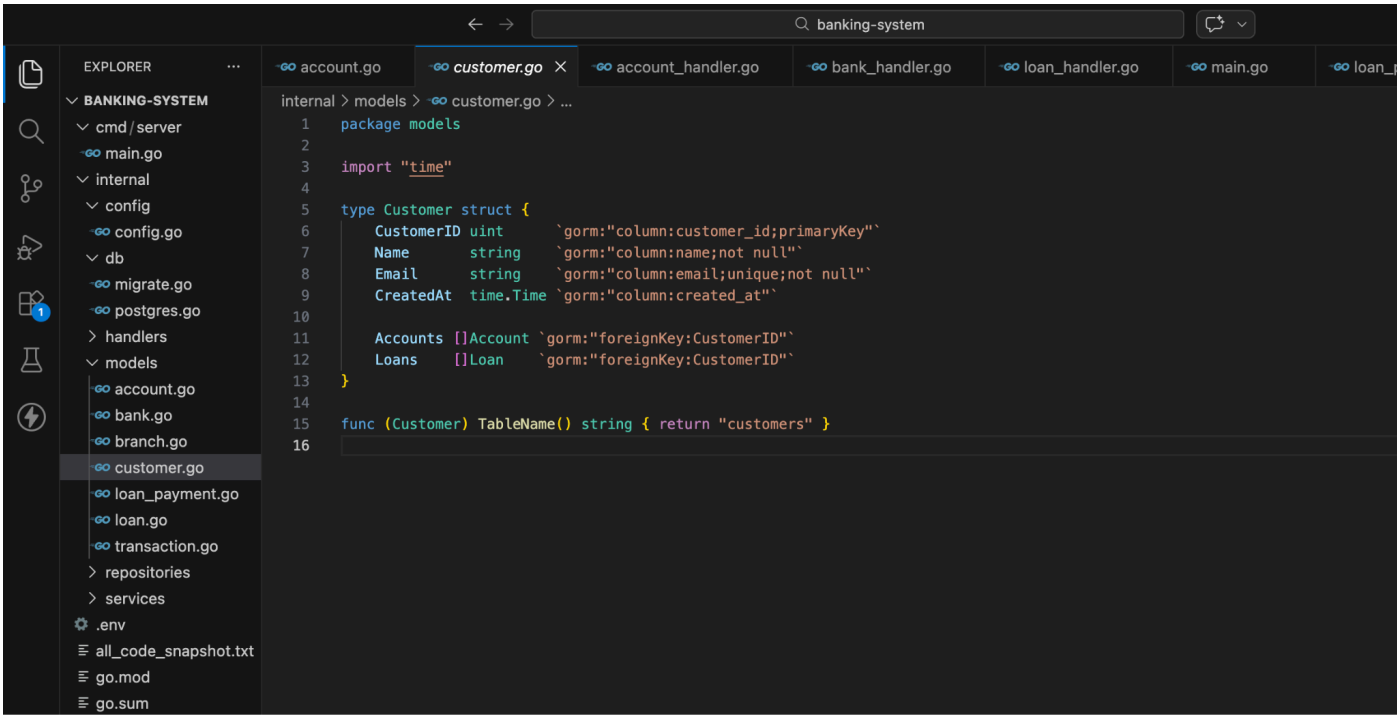
2)Bank.go->



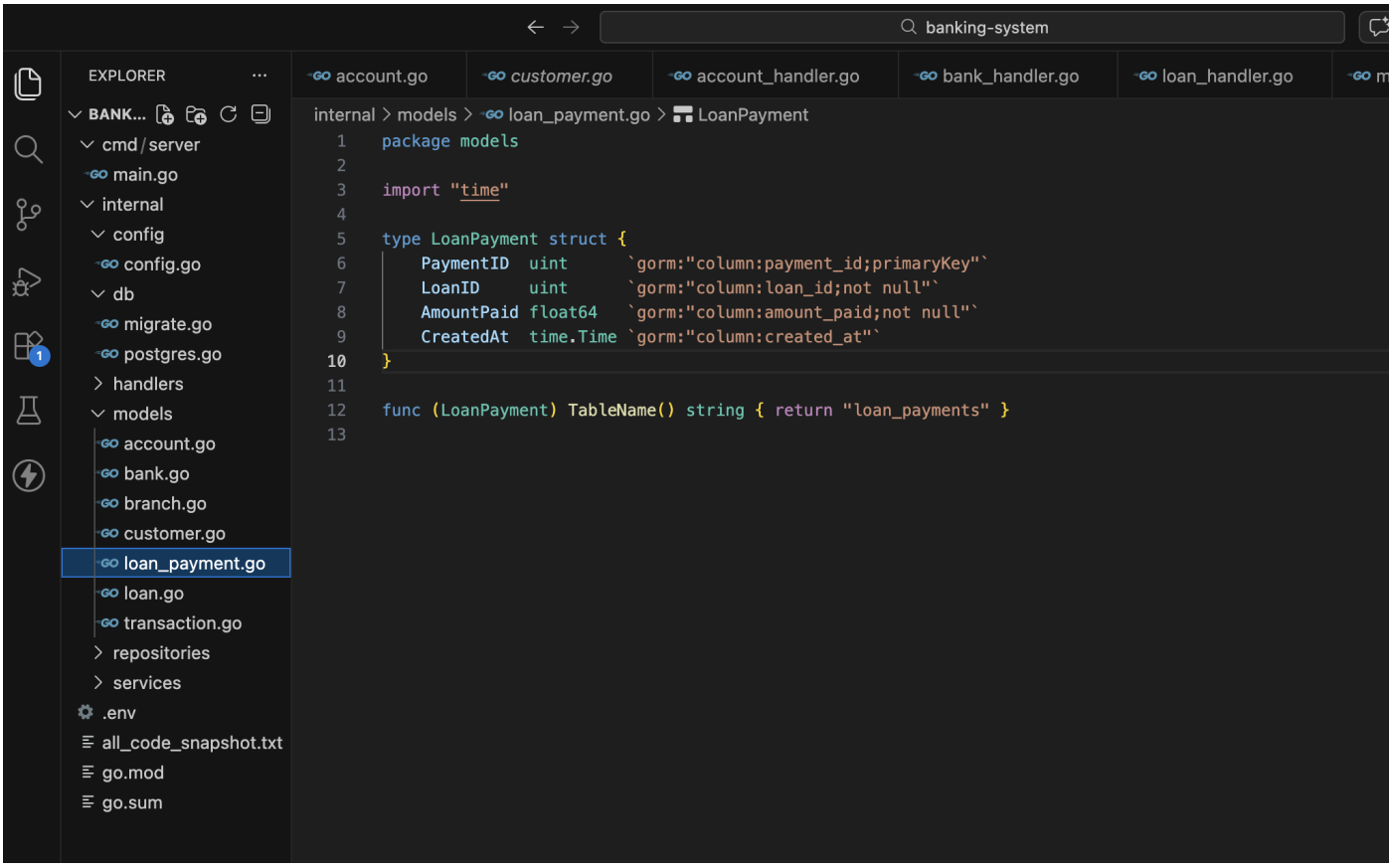
3)Branch.go->



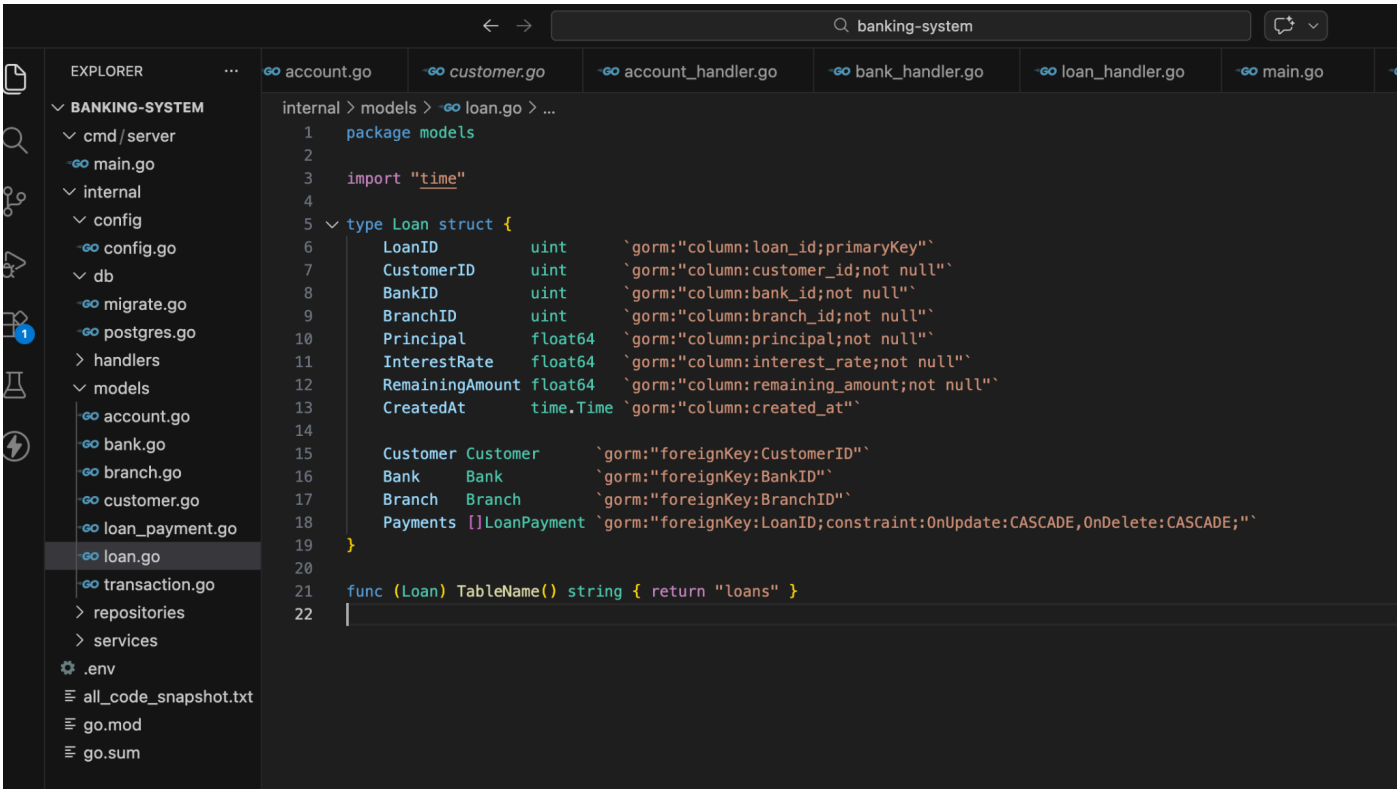
4)Customer.go->



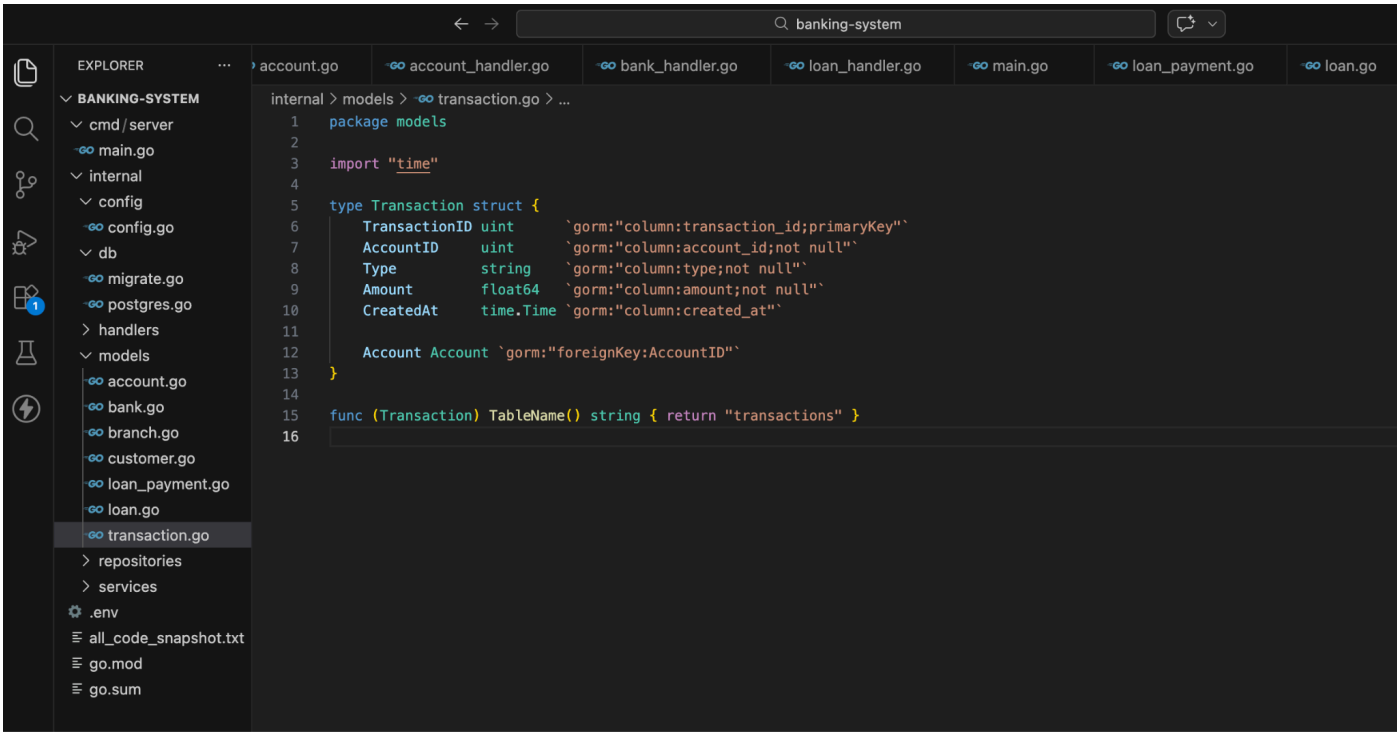
5)loan_payment.go->



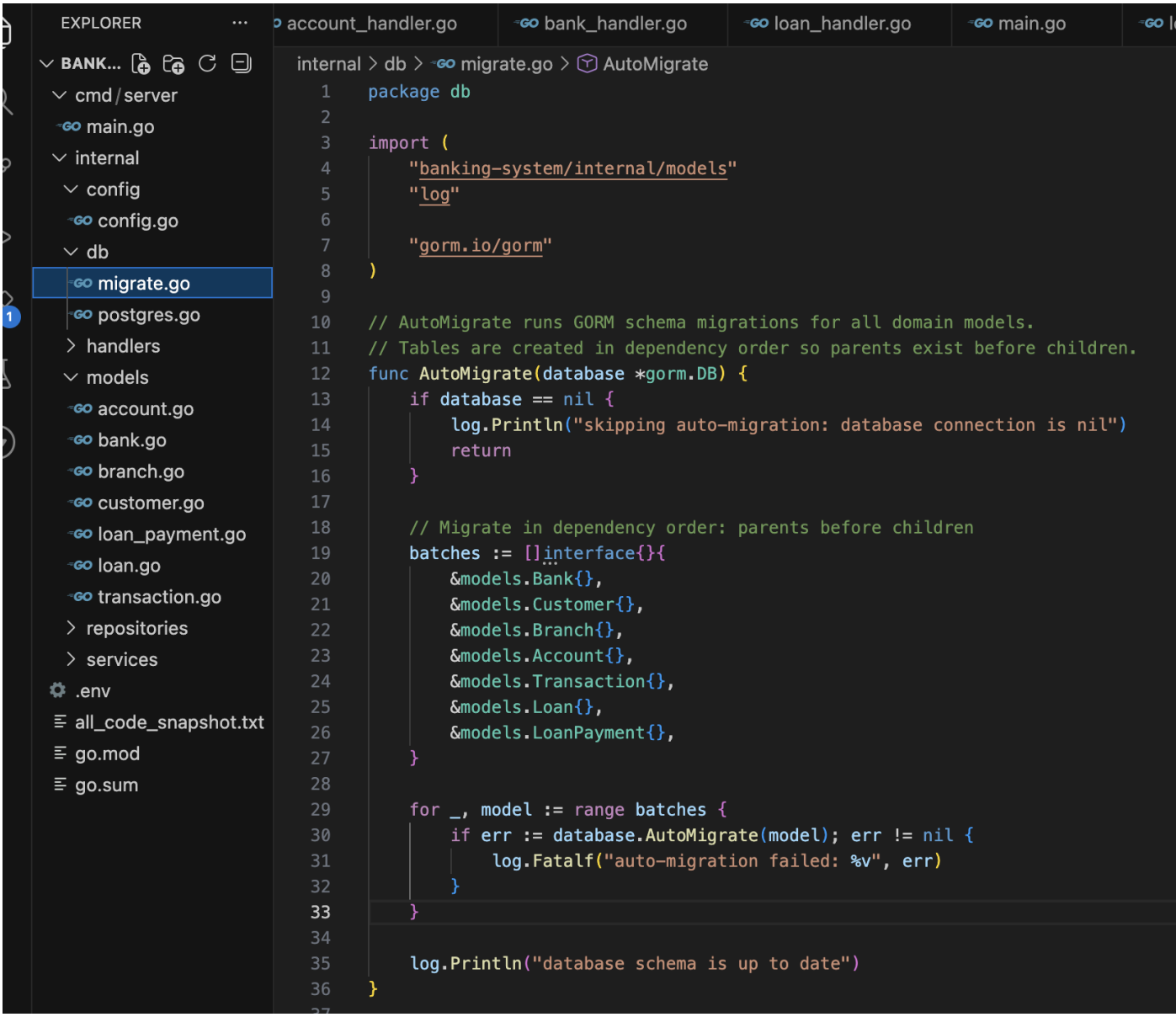
6)loan.go->



7)transaction.go->



8)automigrate.go->



How relationships happen i need foreign keys which i decided like this ->

1.Bank → Branch

A branch cannot exist without bank

So:

branches.bank_id -> banks.bank_id

So branch must belong to a valid bank.

2.Customer → Account

Real-world: A customer can open multiple accounts. An account must belong to exactly one customer.

So:

accounts.customer_id → customers.customer_id

This ensures: No orphan accounts, Clear ownership

---

3. Account → Bank & Branch

I explicitly linked Account to:

accounts.bank_id → banks.bank_id
accounts.branch_id → branches.branch_id

Because:
A bank can have many branches.
Accounts are opened at a specific branch.
This allows querying: All accounts in a branch All accounts under a bank

---

4.Transaction → Account

Transactions are dependent records.

Real-world:

- A transaction cannot exist without an account.

So:

transactions.account_id → accounts.account_id

This guarantees:

All transactions are traceable to one account.

5.Loan → Customer + Bank + Branch

Loan belongs to:

- **A customer**
- **A bank**
- **A branch**

So:

```
loans.customer_id → customers.customer_id
loans.bank_id     → banks.bank_id
loans.branch_id   → branches.branch_id
```

This allows:

Querying loans per customer

Loan portfolio per branch

Loan exposure per bank

6.LoanPayment → Loan

Loan payments depend on a loan:

```
loan_payments.loan_id → loans.loan_id
```

This enforces:

No repayment without a valid loan.

COMPONENTS OF PROJECT:

Handler Layer

What it does

- Handles incoming HTTP requests
- Validates request input (basic format & sanity checks)
- Calls the appropriate method in the **Service layer**
- Converts the result into a structured JSON response (success or error)

It does NOT

- Contain any business logic
- Talk directly to the database

Why we created this separate layer

To achieve clean separation between:

- API / transport concerns
- Business logic

Benefits when things change in future:

- API format changes (v1 → v2)
- Response structure changes
- New authentication middleware added
- New client type (gRPC, CLI, etc.)

→ **Only the handler layer needs to change.**

Service and Repository layers remain untouched.

Example: Deposit Flow

Request

text

```
POST /accounts/1/deposit
{
  "amount": 500
}
```

Inside Handler (pseudo-code)

1. Extract account_id from URL/path
2. Bind & parse JSON body
3. Validate: amount > 0 (simple check)
4. **Call service: h.service.Deposit(accountID, amount)**
5. Return JSON:

   Success → { "status": "success", "new_balance": 1500 }

   Error → { "error": "insufficient funds", "code": 400 }

Key responsibility (one-liner)
Convert **HTTP input** → Business call
Convert **Business result** → JSON output

Service Layer (Business Logic Layer)

This is the brain of the application.

What it does

- Implements real business rules
- Validates domain-specific rules
- Manages transactions (atomic operations)
- Coordinates calls to one or more repositories

Why we created this layer

Business rules should **never** live in:

- Handlers (HTTP/transport concern)
- Repositories (data access concern)

Examples of business rules that belong here

- Cannot withdraw more than current balance
- Branch must belong to the correct bank
- Loan interest is fixed at 12% per year
- Money transfer must update **both** accounts atomically

Example: Transfer Money

Inside AccountService.Transfer()

1. Validate amount > 0
2. Fetch fromAccount
3. Fetch toAccount
4. Check: fromAccount.balance >= amount
5. Start database transaction
6. Decrease fromAccount.balance
7. Increase toAccount.balance
8. Create two transaction records (debit + credit)
9. Commit transaction

→ If **anything** fails → rollback everything

→ Guarantees **data consistency** and **atomicity**

Why transactions belong in Service (not Repository)
Because a **transaction = one business operation**

Not a low-level database concern.

Key responsibilities

- Enforce business & domain rules
- Orchestrate / coordinate repositories
- Guarantee data consistency

Repository Layer (Data Access Layer)

This layer talks directly to the database.

What it does

- Executes GORM (or sqlx / raw SQL) queries
- Performs CRUD operations
- Maps domain models to database tables

Typical methods

- Create(account *Account) error
- GetByID(id uint) (*Account, error)
- UpdateBalance(id uint, newBalance float64) error
- FindByCustomerID(customerID uint) ([]*Account, error)

Why I created this layer

To **isolate** all database-related code.

Future-proof benefit

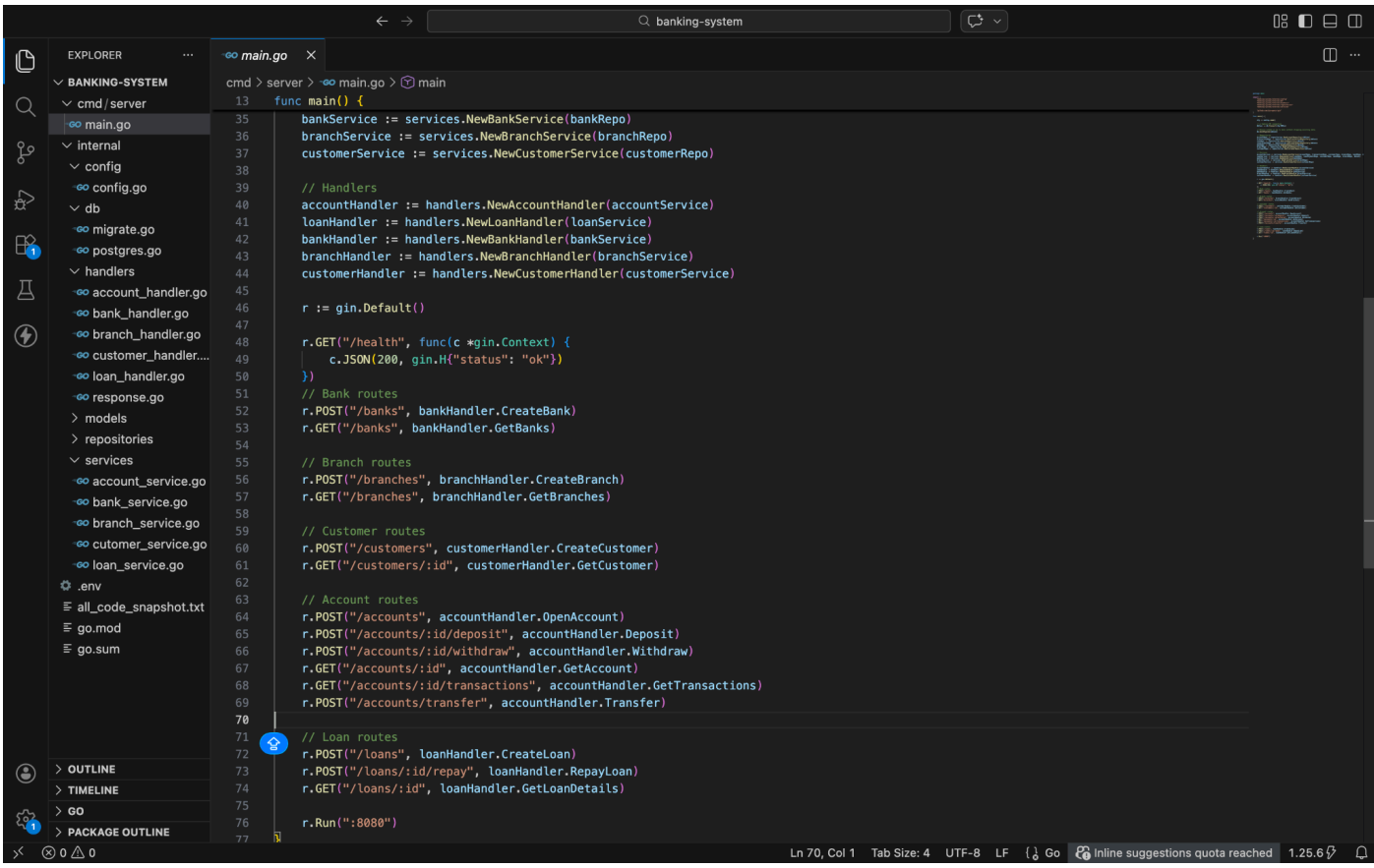
If tomorrow we need to:

- Switch from PostgreSQL → MySQL
- Replace GORM → raw SQL / sqlx
- Move to MongoDB / DynamoDB

→ **Only repository layer changes**

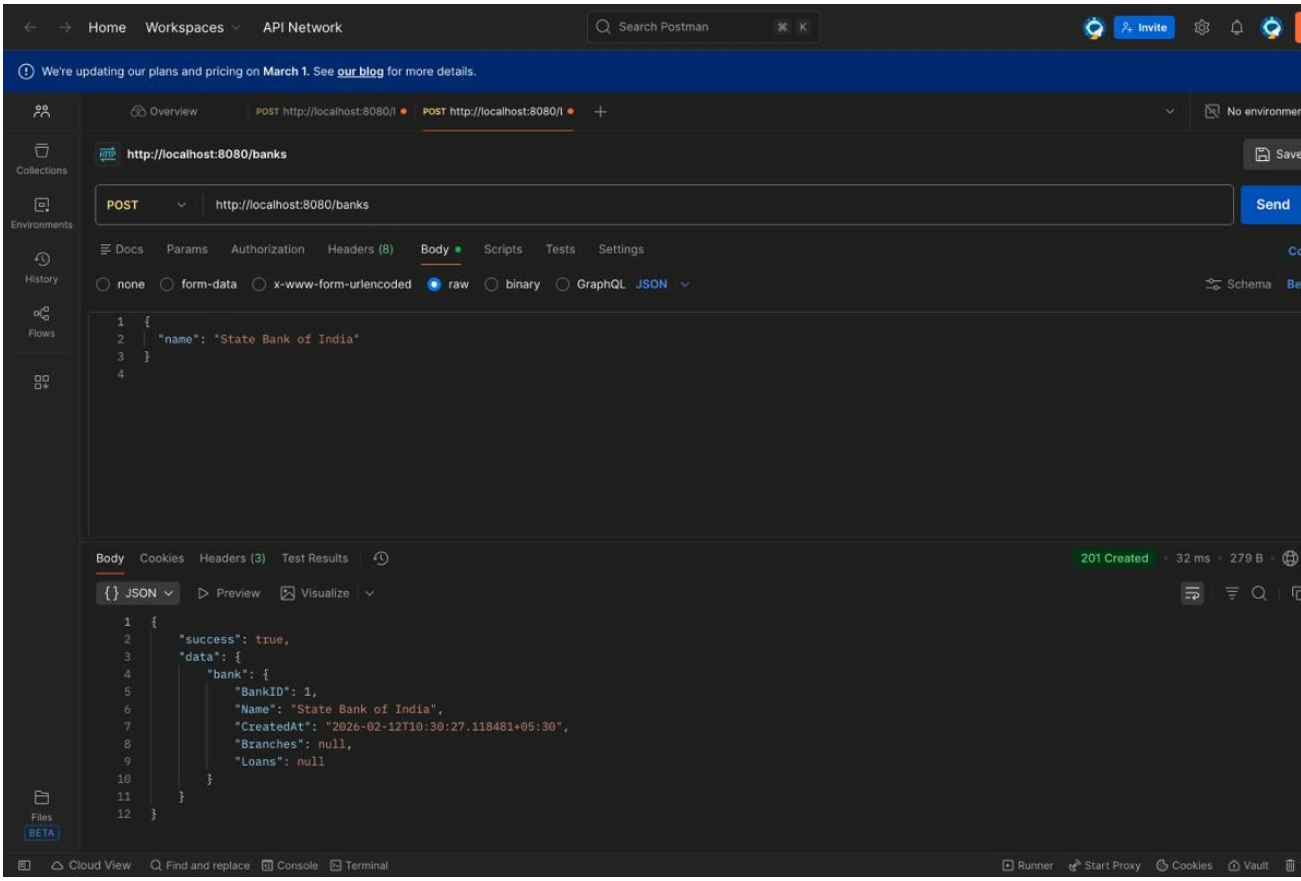
Service + Handler layers stay almost the same.

API'S IN MY PROJECT:

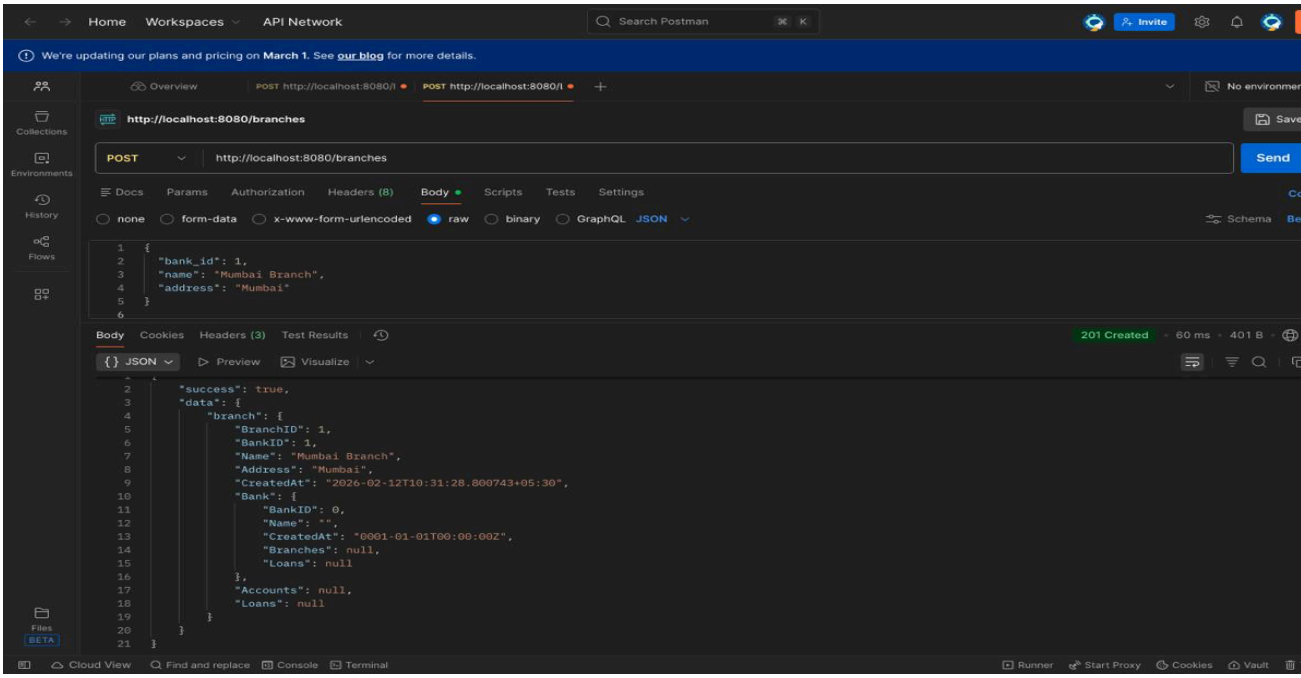


OUTPUT (IMPLEMENTATION OF ALL API'S)

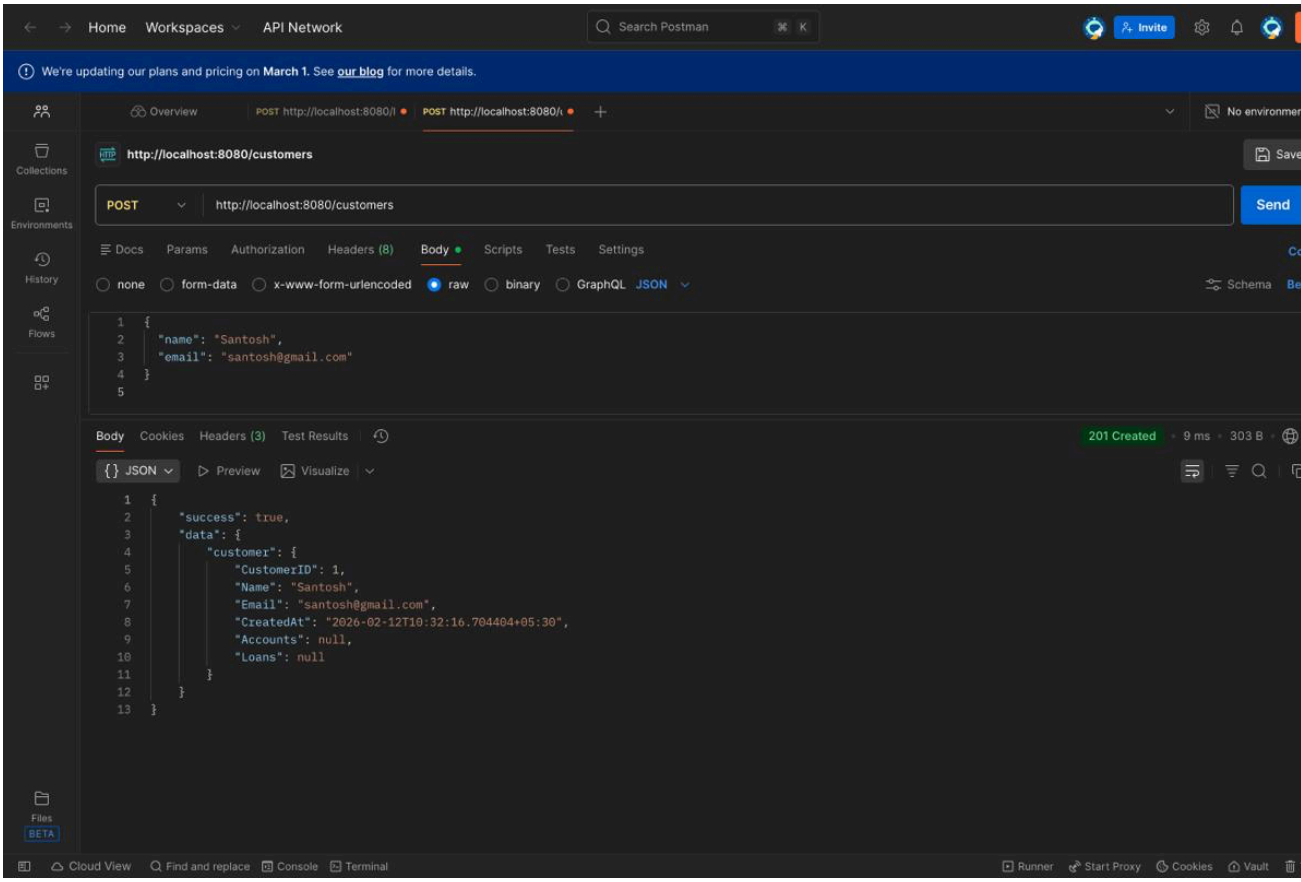
1) CREATING BANK



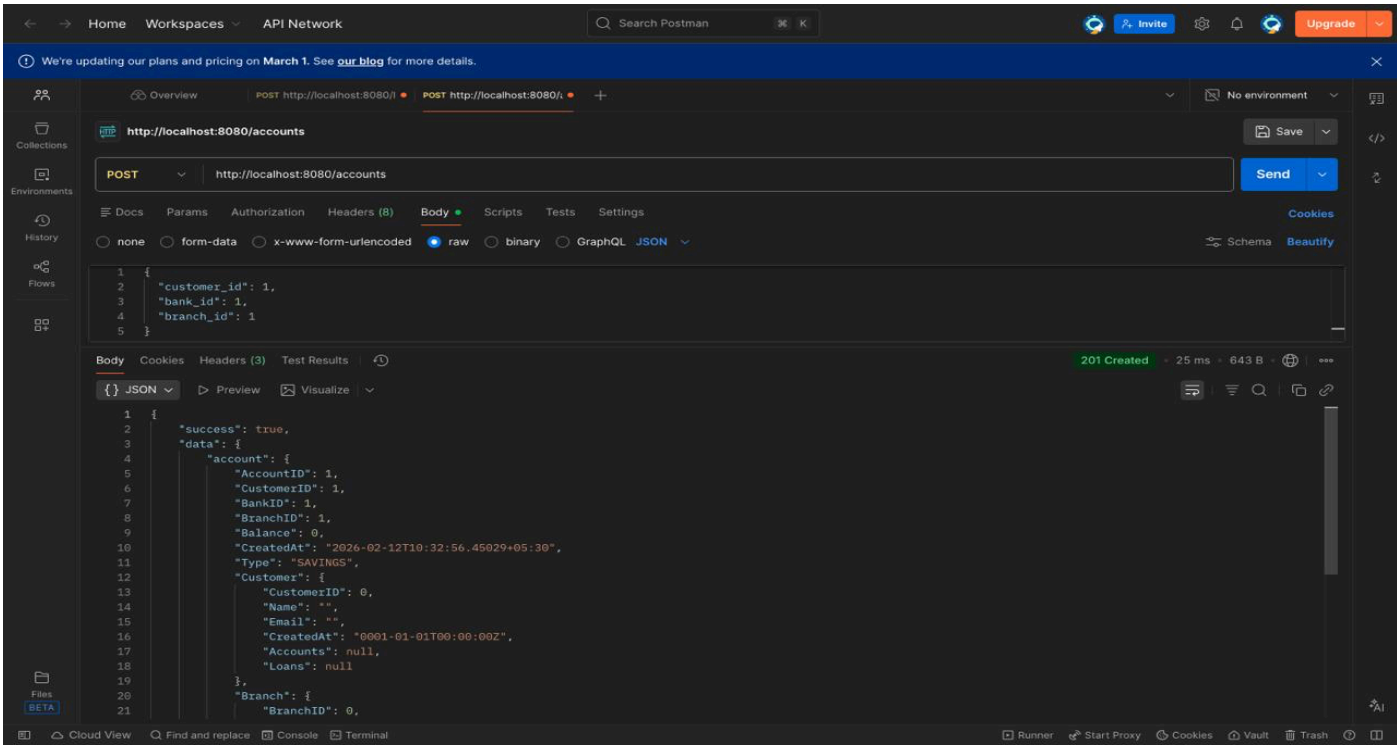
2) CREATING BRANCH

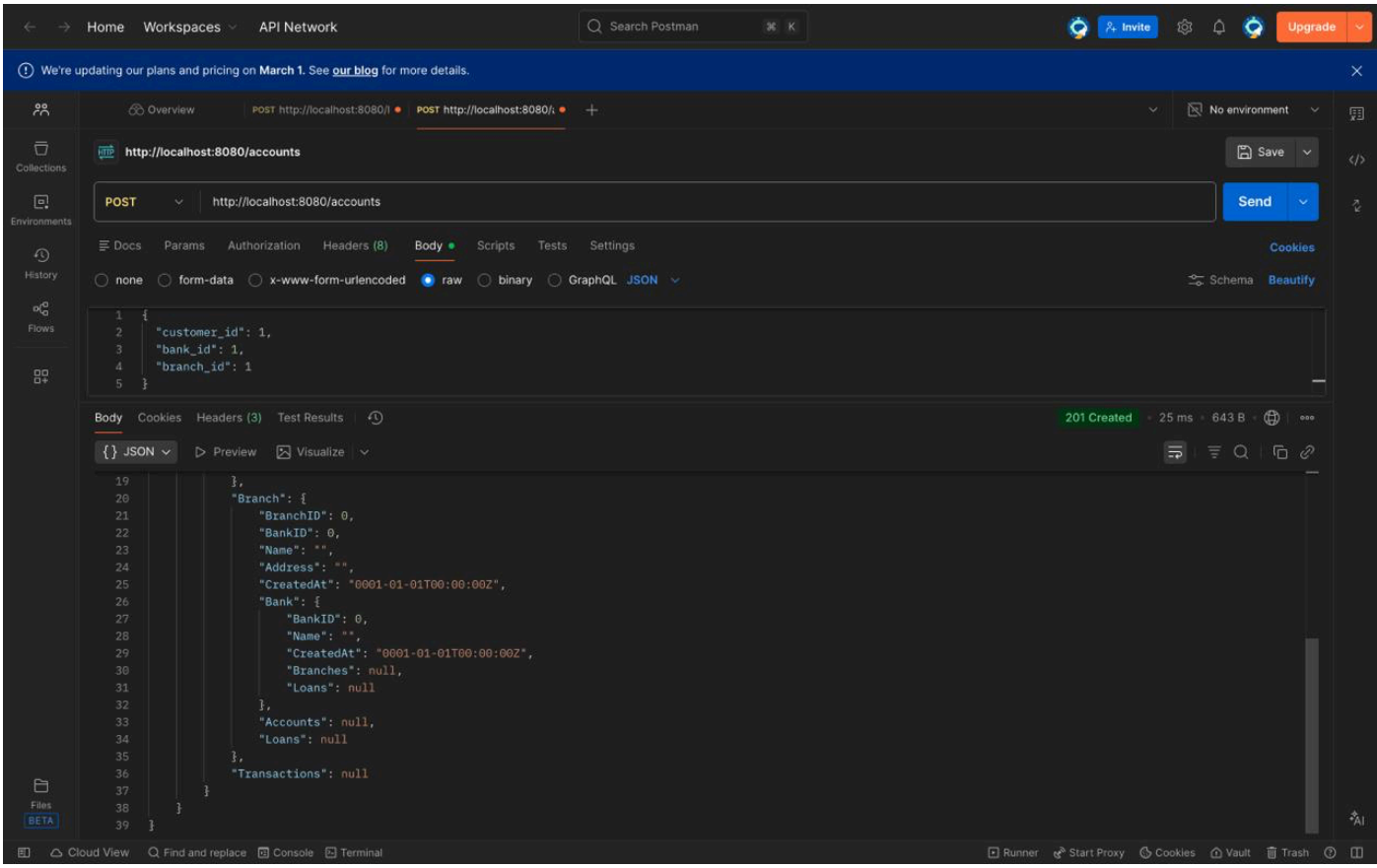


3) CREATING CUSTOMER

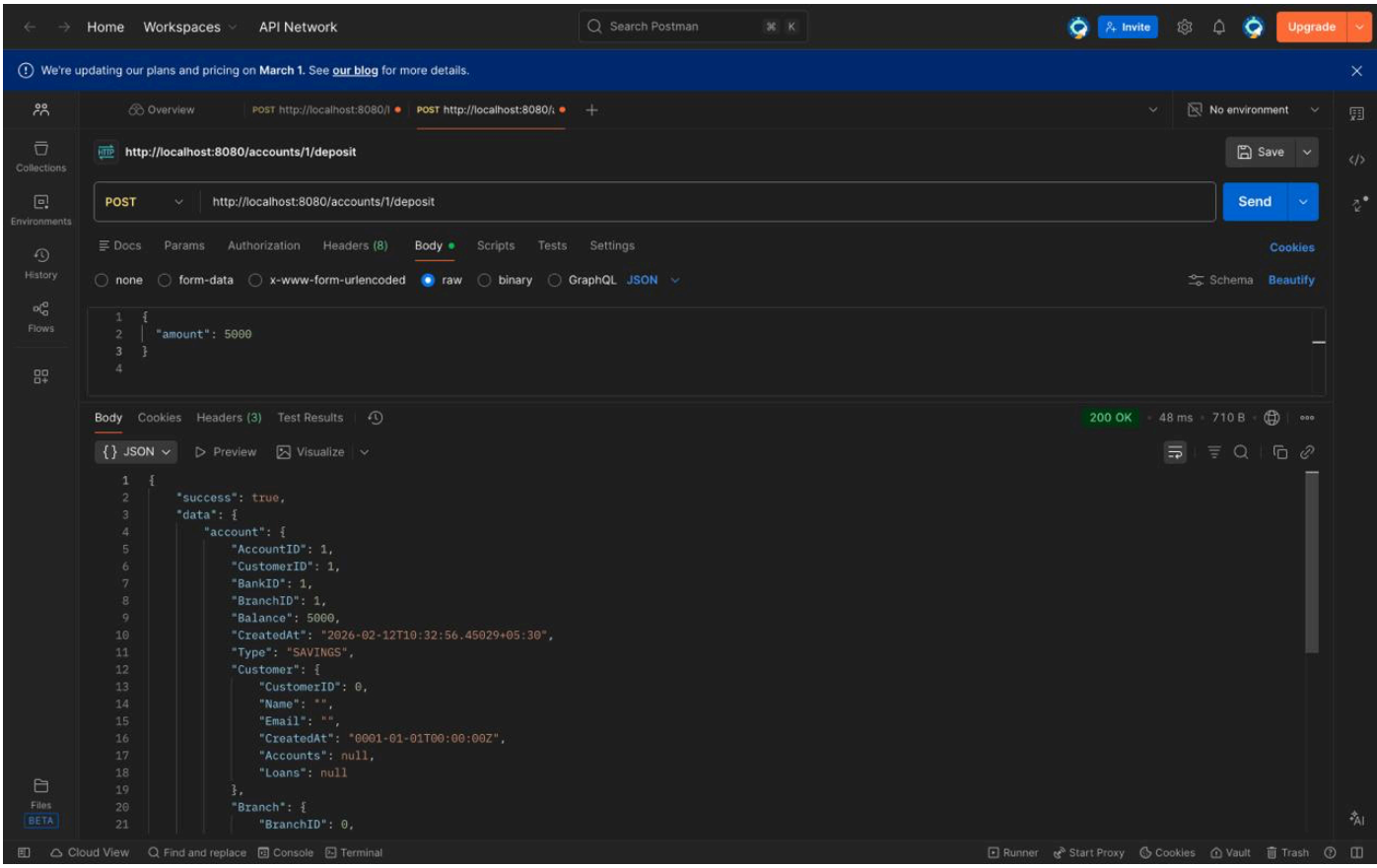


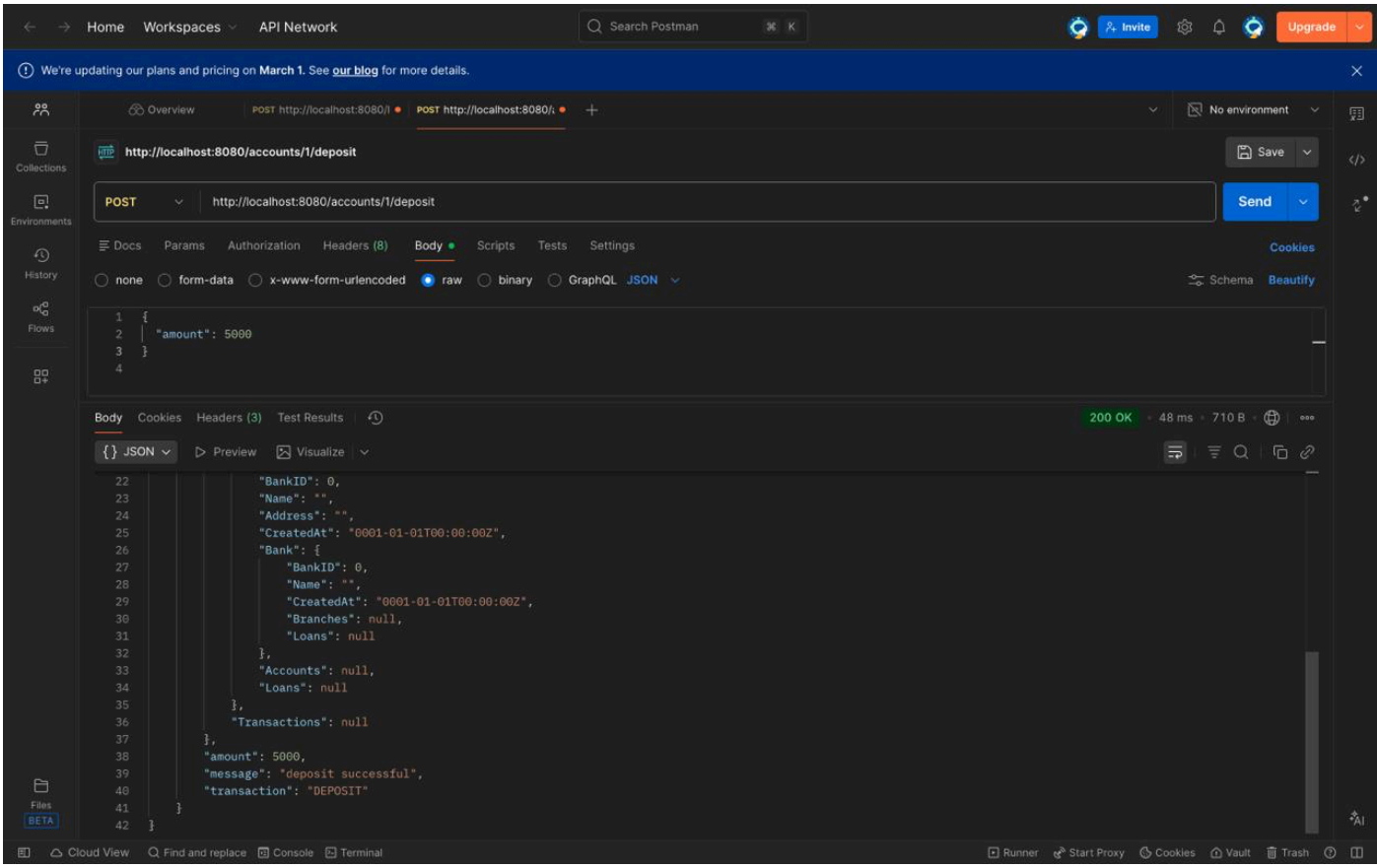
4) OPEN ACCOUNT



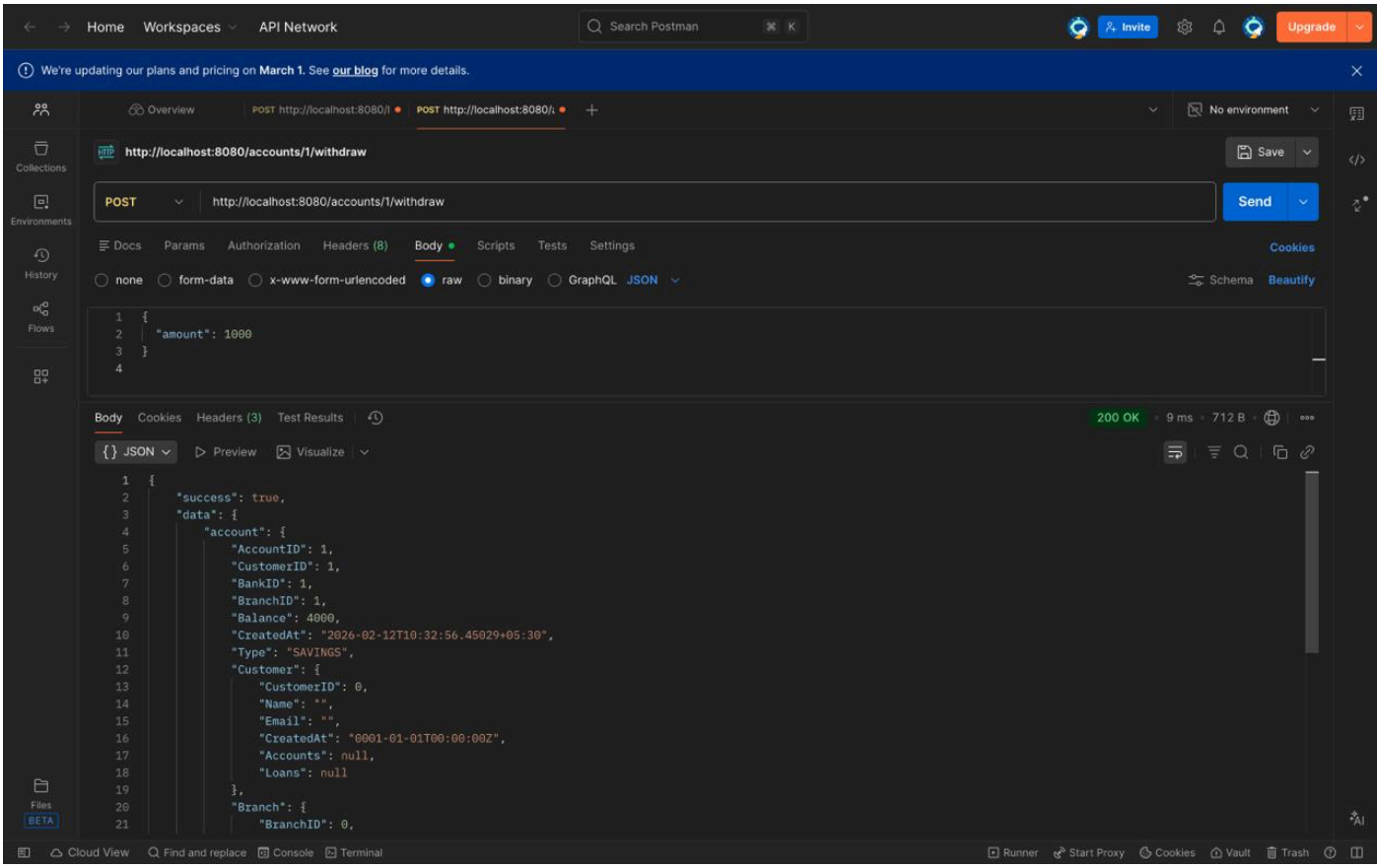


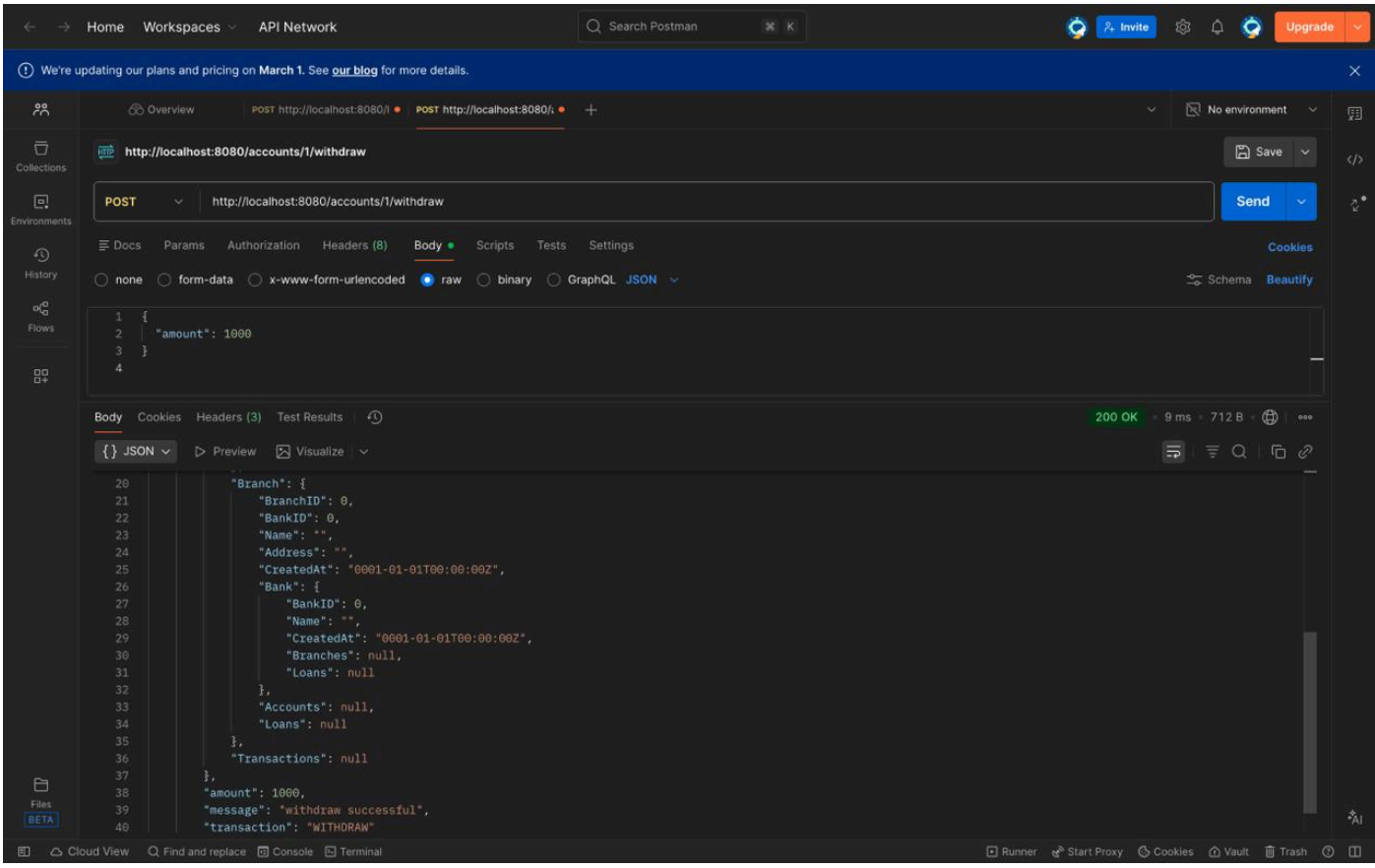
6)DEPOSIT MONEY -> SANTOSH DEPOSIT 5000



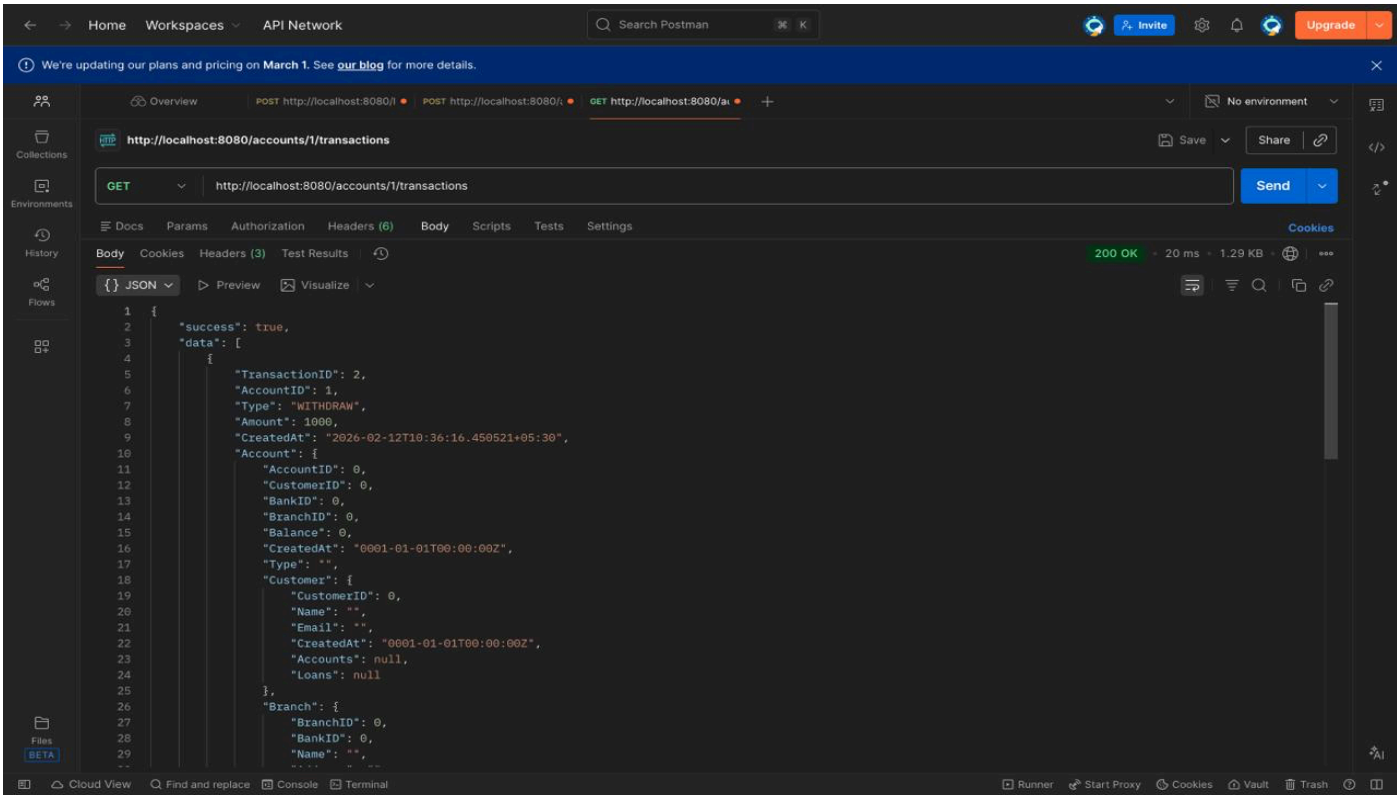


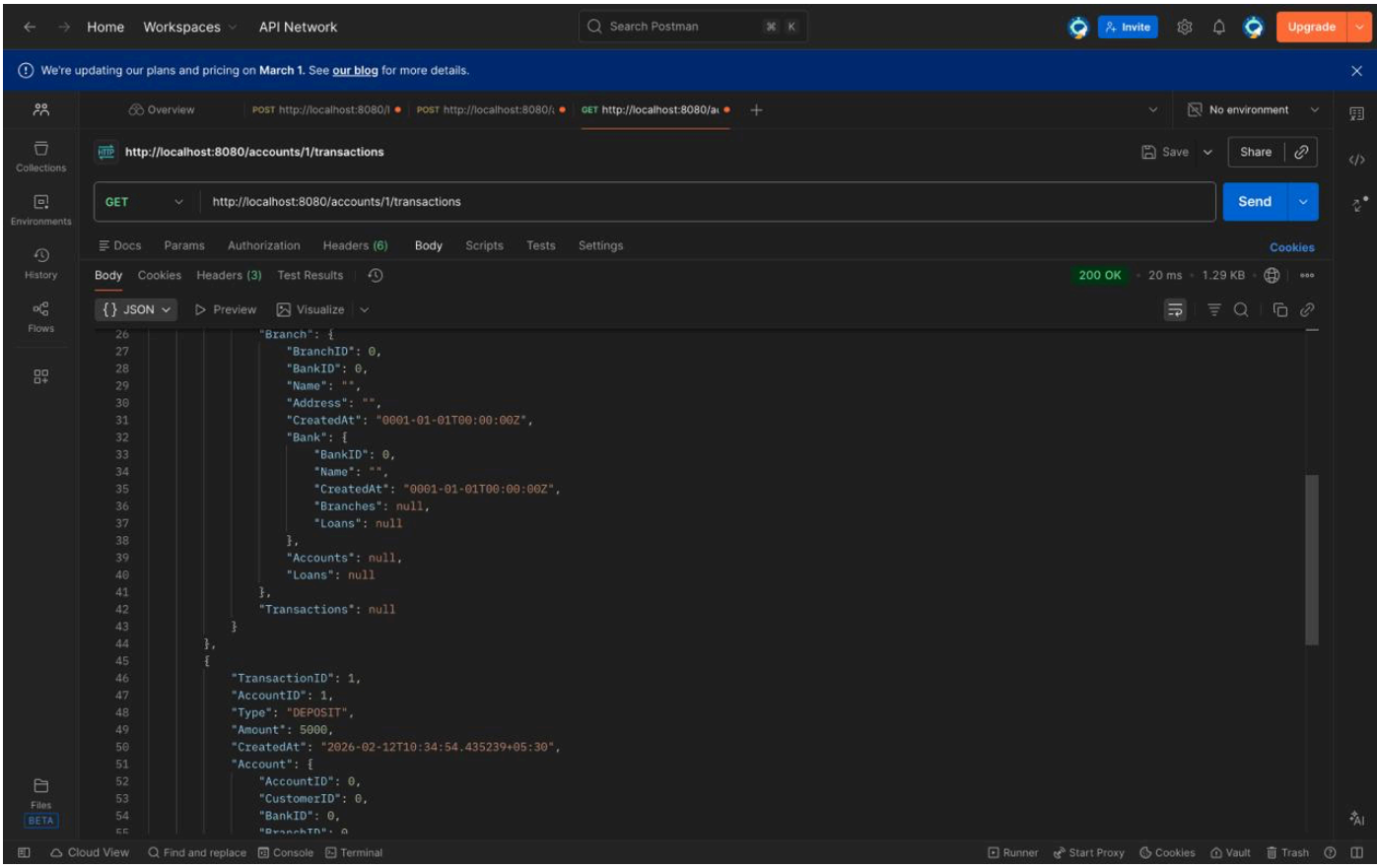
7)Withdraw Money-> SANTOSH WITHDRAWS 1000, NOW HAS 4000

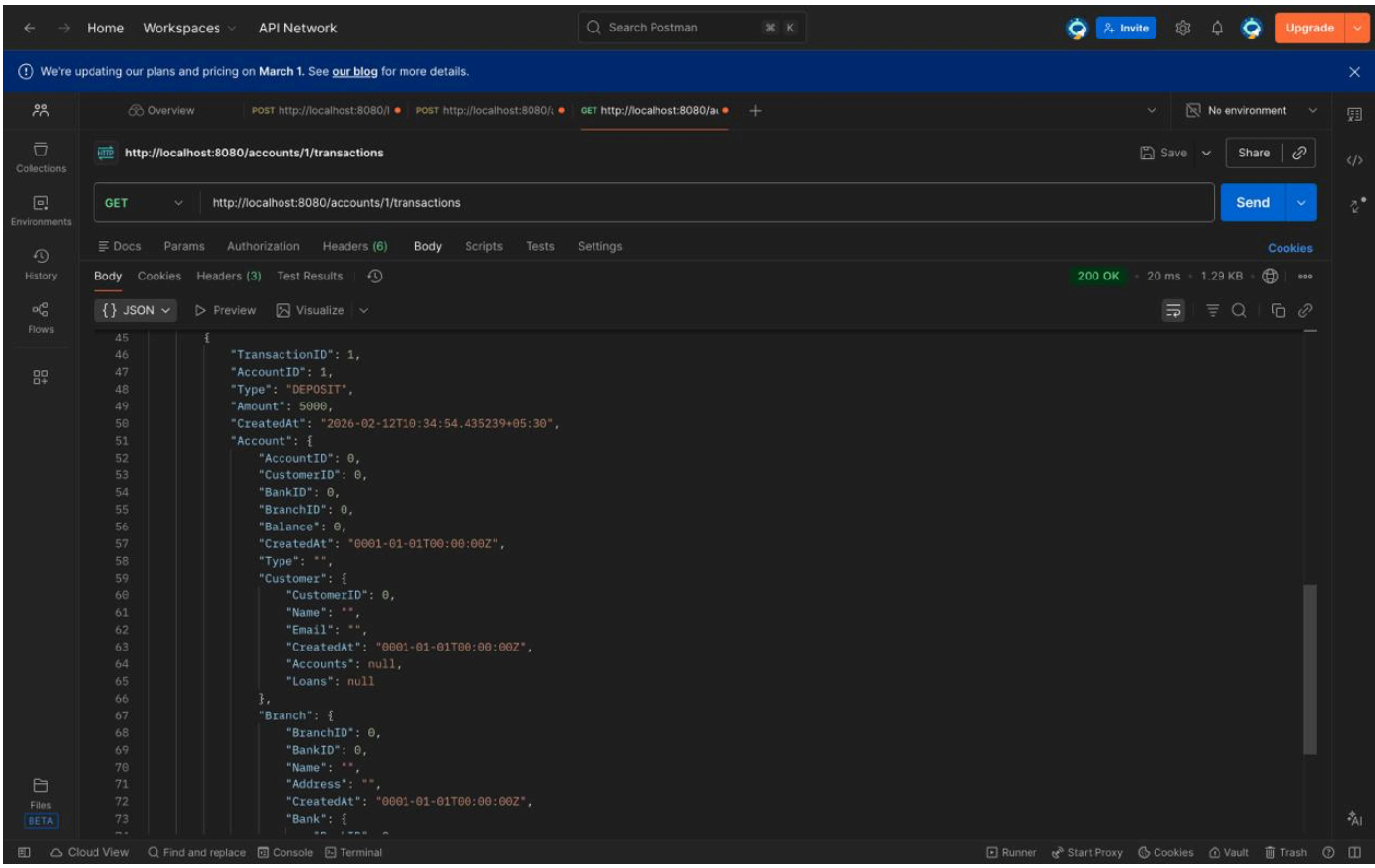


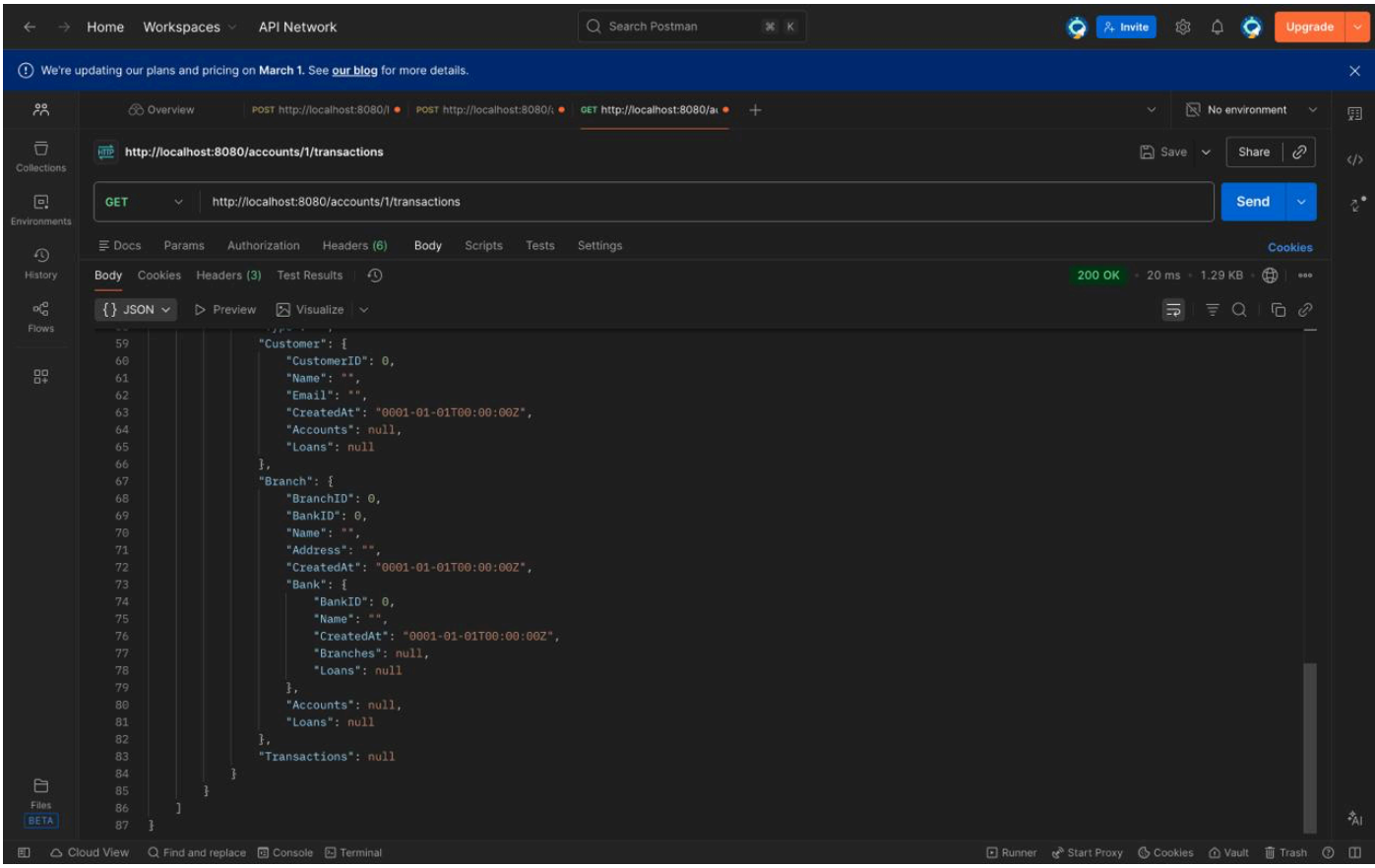


8) NOW TO SEE ALL TRANSACTIONS MADE



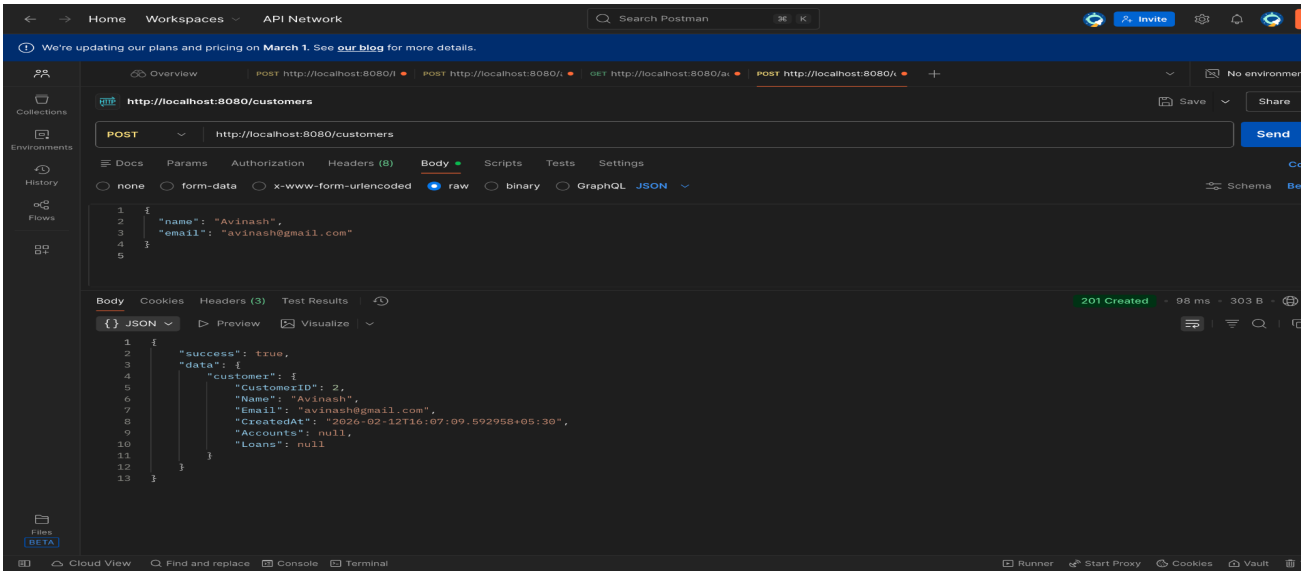

Home
Workspaces
API Network
Search Postman
Invite
Upgrade
We're updating our plans and pricing on March 1. See our blog for more details.
Overview
No environment
http://localhost:8080/accounts/1/transactions
Save
Share
GET
http://localhost:8080/accounts/1/transactions
Send
Docs
Params
Authorization
Headers (6)
Body
Scripts
Tests
Settings
Cookies
Body
Cookies
Headers (3)
Test Results
200 OK
20 ms
1.29 KB
JSON
Preview
Visualize
"Branch": {
"BranchID": 0,
"BankID": 0,
"Name": "",
"Address": "",
"CreatedAt": "0001-01-01T00:00:00Z",
"Bank": {
"BankID": 0,
"Name": "",
"CreatedAt": "0001-01-01T00:00:00Z",
"Branches": null,
"Loans": null
},
"Accounts": null,
"Loans": null
},
"Transactions": null
}
},
{
"TransactionID": 1,
"AccountID": 1,
"Type": "DEPOSIT",
"Amount": 5000,
"CreatedAt": "2026-02-12T10:34:54.435239+05:30",
"Account": {
"AccountID": 0,
"CustomerID": 0,
"BankID": 0,
Files
BETA
Cloud View
Find and replace
Console
Terminal
Runner
Start Proxy
Cookies
Vault
Trash


Home
Workspaces
API Network
Search Postman
Invite
Upgrade
We're updating our plans and pricing on March 1. See our blog for more details.
Overview
No environment
http://localhost:8080/accounts/1/transactions
Save
Share
GET
http://localhost:8080/accounts/1/transactions
Send
Docs
Params
Authorization
Headers (6)
Body
Scripts
Tests
Settings
Cookies
Body
Cookies
Headers (3)
Test Results
200 OK
20 ms
1.29 KB
JSON
Preview
Visualize
{
"TransactionID": 1,
"AccountID": 1,
"Type": "DEPOSIT",
"Amount": 5000,
"CreatedAt": "2026-02-12T10:34:54.435239+05:30",
"Account": {
"AccountID": 0,
"CustomerID": 0,
"BankID": 0,
"BranchID": 0,
"Balance": 0,
"CreatedAt": "0001-01-01T00:00:00Z",
"Type": "",
"Customer": {
"CustomerID": 0,
"Name": "",
"Email": "",
"CreatedAt": "0001-01-01T00:00:00Z",
"Accounts": null,
"Loans": null
},
"Branch": {
"BranchID": 0,
"BankID": 0,
"Name": "",
"Address": "",
"CreatedAt": "0001-01-01T00:00:00Z",
"Bank": {
Files
BETA
Cloud View
Find and replace
Console
Terminal
Runner
Start Proxy
Cookies
Vault
Trash

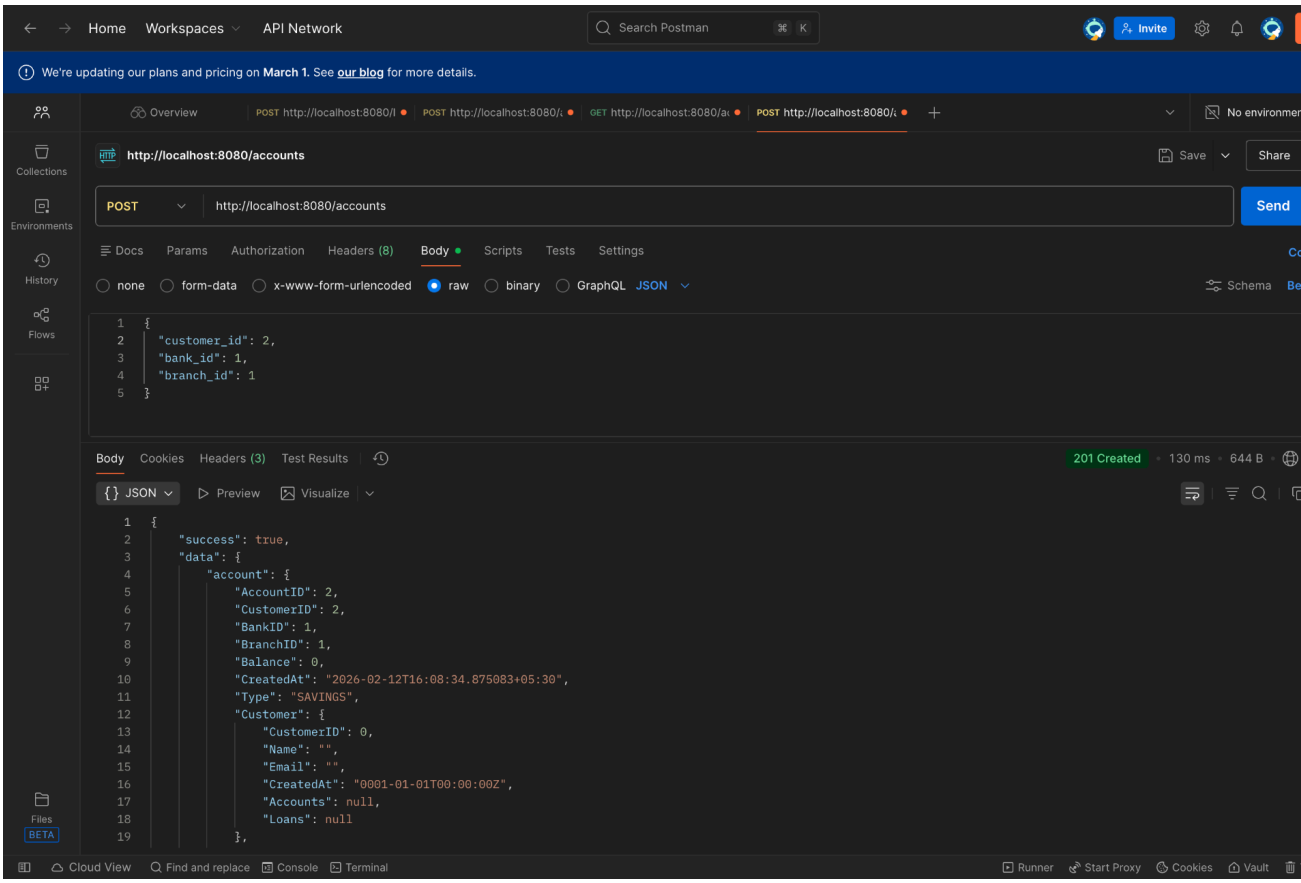


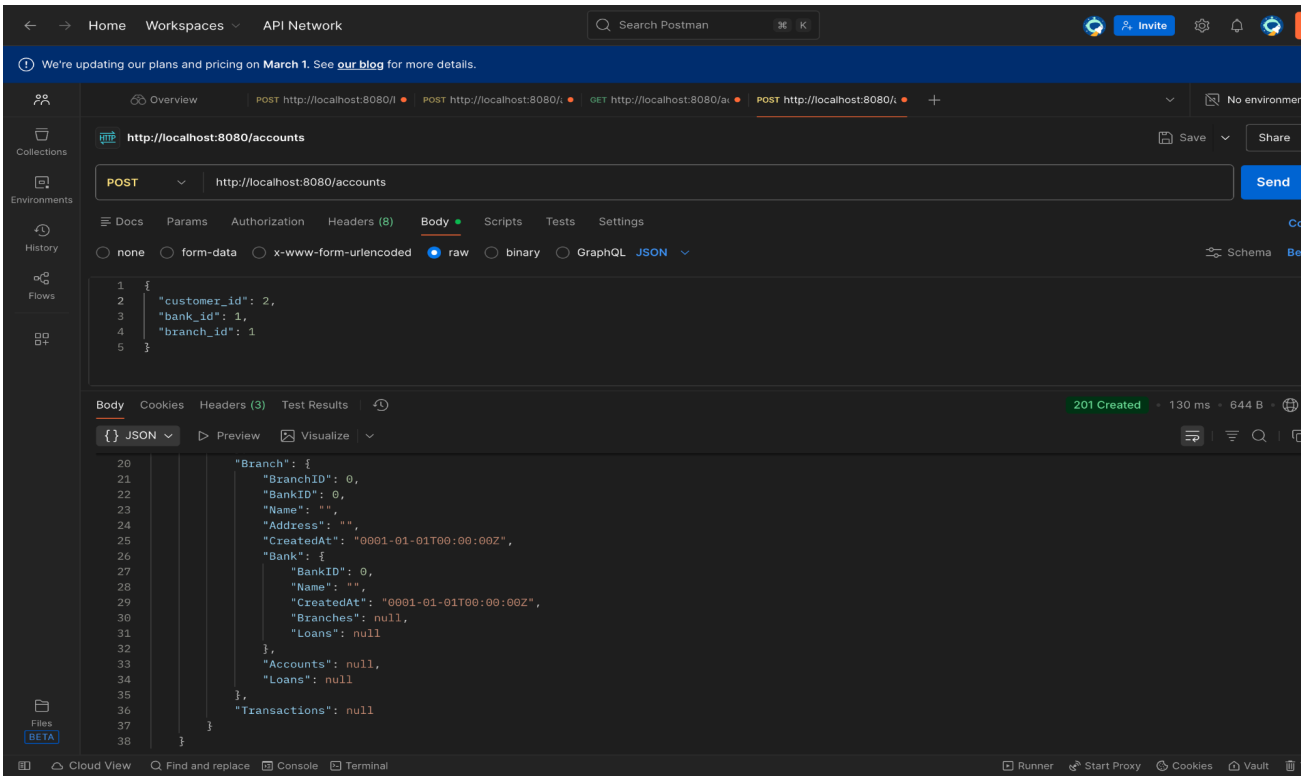
NOW TO TRANSFER BETWEEN 2 ACCOUNTS
WE ALREADY HAVE SANTOSH CUSTOMER , NOW CREATING AVINASH CUSTOMER

A) CREATING AVINASH CUSTOMER



B) Opening Avinash Account in Same Bank And Branch



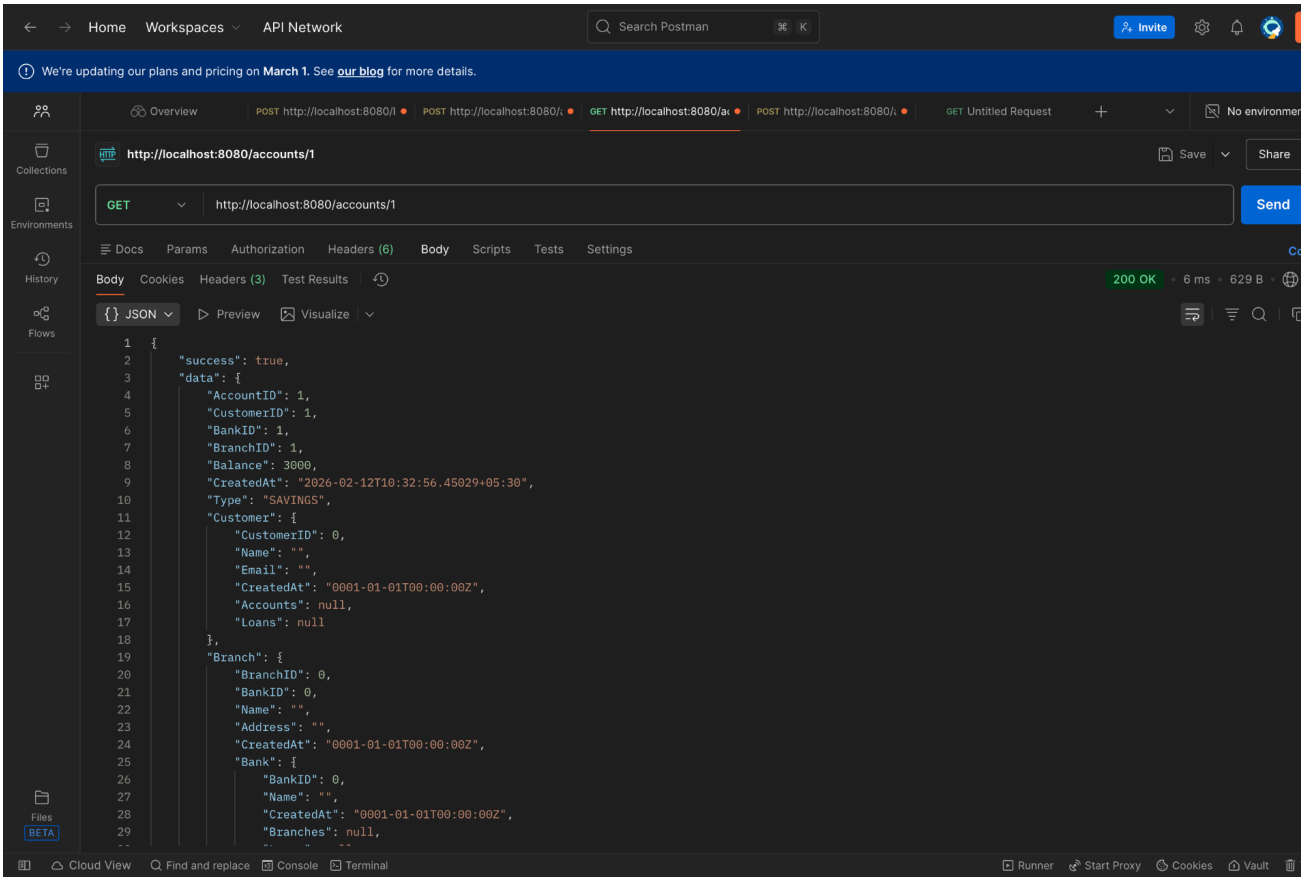


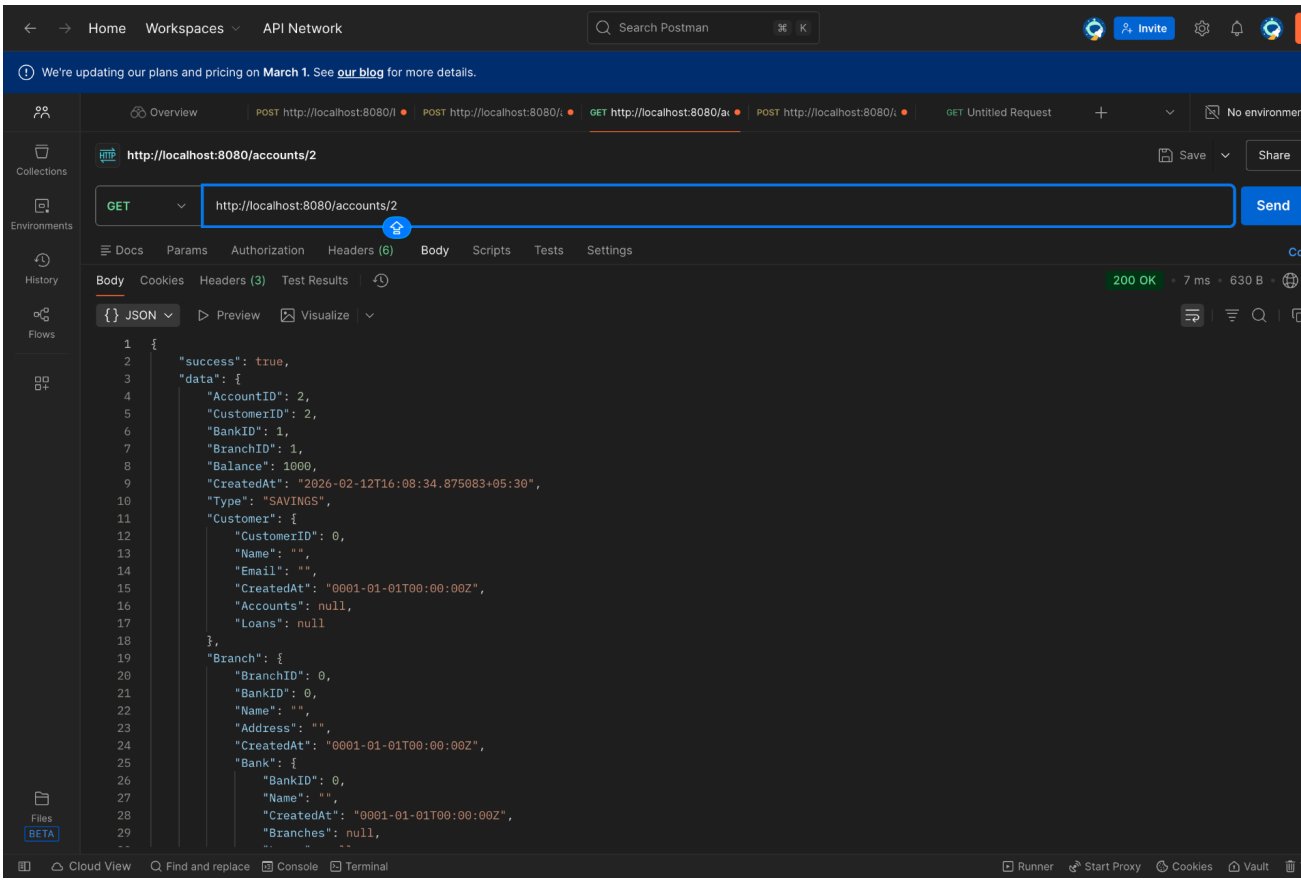
C) NOW SANTOSH TRANSFERS 1000 FROM HIS ACCOUNT TO AVINASH


```
{
   "success": true,
   "data": {
       "amount": 1000,
       "from_account": {
           "AccountID": 1,
           "CustomerID": 1,
           "BankID": 1,
           "BranchID": 1,
           "Balance": 3000,
           "CreatedAt": "2026-02-12T10:32:56.45029+05:30",
           "Type": "SAVINGS",
           "Customer": {
               "CustomerID": 0,
               "Name": "",
               "Email": "",
               "CreatedAt": "0001-01-01T00:00:00Z",
               "Accounts": null,
               "Loans": null
           },
           "Branch": {
               "BranchID": 0,
               "BankID": 0,
               "Name": "",
               "Address": "",
               "CreatedAt": "0001-01-01T00:00:00Z",
               "Bank": {
                   "BankID": 0,
                   "Name": "",
                   "CreatedAt": "0001-01-01T00:00:00Z",
                   "Branches": null,
                   "Loans": null
               },
               "Accounts": null,
               "Loans": null
           },
           "Transactions": null
       },
       "message": "transfer successful",
       "to_account": {
           "AccountID": 2,
           "CustomerID": 2,
           "BankID": 1,
```

```
            "BranchID": 1,
            "Balance": 1000,
            "CreatedAt": "2026-02-12T16:08:34.875083+05:30",
            "Type": "SAVINGS",
            "Customer": {
                "CustomerID": 0,
                "Name": "",
                "Email": "",
                "CreatedAt": "0001-01-01T00:00:00Z",
                "Accounts": null,
                "Loans": null
            },
            "Branch": {
                "BranchID": 0,
                "BankID": 0,
                "Name": "",
                "Address": "",
                "CreatedAt": "0001-01-01T00:00:00Z",
                "Bank": {
                    "BankID": 0,
                    "Name": "",
                    "CreatedAt": "0001-01-01T00:00:00Z",
                    "Branches": null,
                    "Loans": null
                },
                "Accounts": null,
                "Loans": null
            },
            "Transactions": null
        }
    }
}
```

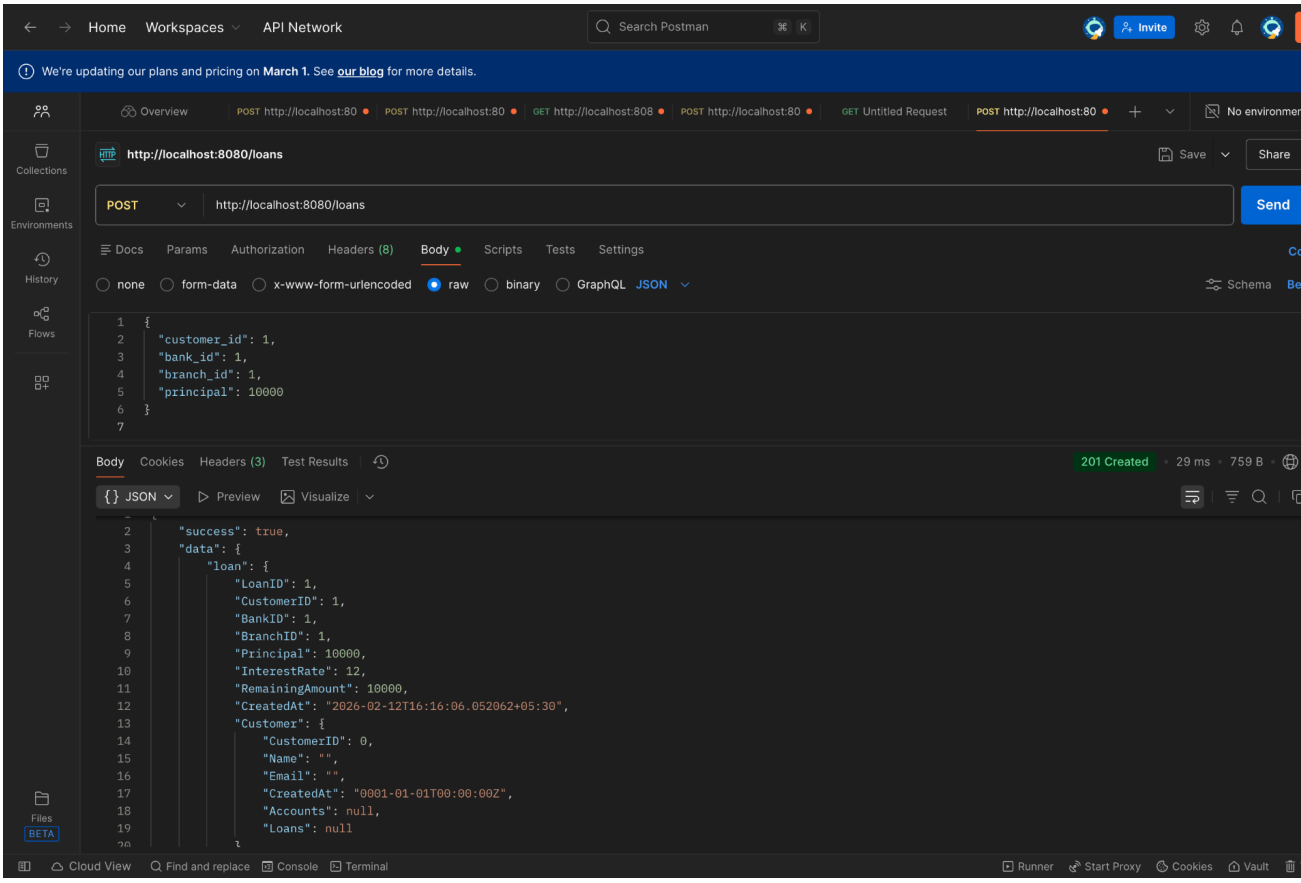

NOW SANTOSH HAS 3000, AVINASH HAS 1000


Home
Workspaces
API Network
Search Postman
Invite
We're updating our plans and pricing on March 1. See our blog for more details.
Overview
GET http://localhost:8080/ac
GET Untitled Request
No environment
Collections
Environments
History
Flows
Files
BETA
http://localhost:8080/accounts/1
Save
Share
GET
http://localhost:8080/accounts/1
Send
Docs
Params
Authorization
Headers (6)
Body
Scripts
Tests
Settings
Body
Cookies
Headers (3)
Test Results
200 OK
6 ms
629 B
{ } JSON
Preview
Visualize
{
"success": true,
"data": {
"AccountID": 1,
"CustomerID": 1,
"BankID": 1,
"BranchID": 1,
"Balance": 3000,
"CreatedAt": "2026-02-12T10:32:56.45029+05:30",
"Type": "SAVINGS",
"Customer": {
"CustomerID": 0,
"Name": "",
"Email": "",
"CreatedAt": "0001-01-01T00:00:00Z",
"Accounts": null,
"Loans": null
},
"Branch": {
"BranchID": 0,
"BankID": 0,
"Name": "",
"Address": "",
"CreatedAt": "0001-01-01T00:00:00Z",
"Bank": {
"BankID": 0,
"Name": "",
"CreatedAt": "0001-01-01T00:00:00Z",
"Branches": null,
Cloud View
Find and replace
Console
Terminal
Runner
Start Proxy
Cookies
Vault

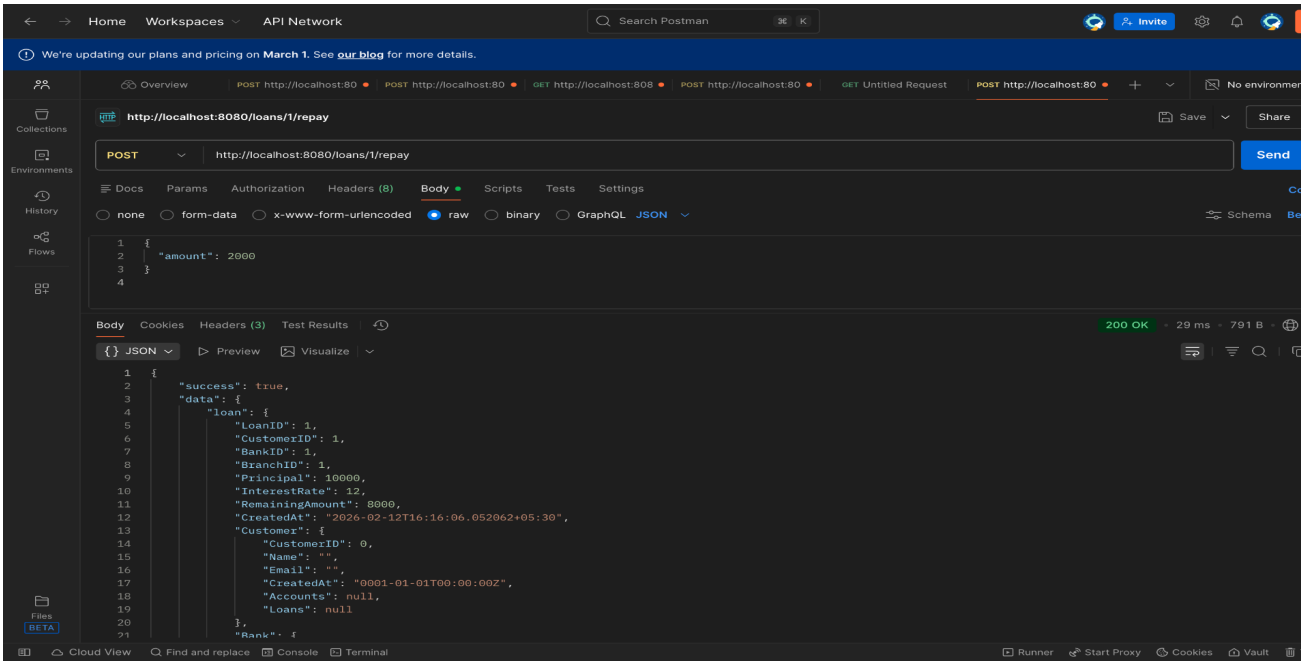

Home
Workspaces
API Network
Search Postman
Invite
We're updating our plans and pricing on March 1. See our blog for more details.
Overview
GET http://localhost:8080/ac
GET Untitled Request
No environment
Collections
Environments
History
Flows
Files
BETA
http://localhost:8080/accounts/2
Save
Share
GET
http://localhost:8080/accounts/2
Send
Docs
Params
Authorization
Headers (6)
Body
Scripts
Tests
Settings
Body
Cookies
Headers (3)
Test Results
200 OK
7 ms
630 B
{ } JSON
Preview
Visualize
{
"success": true,
"data": {
"AccountID": 2,
"CustomerID": 2,
"BankID": 1,
"BranchID": 1,
"Balance": 1000,
"CreatedAt": "2026-02-12T16:08:34.875083+05:30",
"Type": "SAVINGS",
"Customer": {
"CustomerID": 0,
"Name": "",
"Email": "",
"CreatedAt": "0001-01-01T00:00:00Z",
"Accounts": null,
"Loans": null
},
"Branch": {
"BranchID": 0,
"BankID": 0,
"Name": "",
"Address": "",
"CreatedAt": "0001-01-01T00:00:00Z",
"Bank": {
"BankID": 0,
"Name": "",
"CreatedAt": "0001-01-01T00:00:00Z",
"Branches": null,
Cloud View
Find and replace
Console
Terminal
Runner
Start Proxy
Cookies
Vault

NOW TO CREATE LOAN/REPAY LOAN/ GET LOAN DETAILS

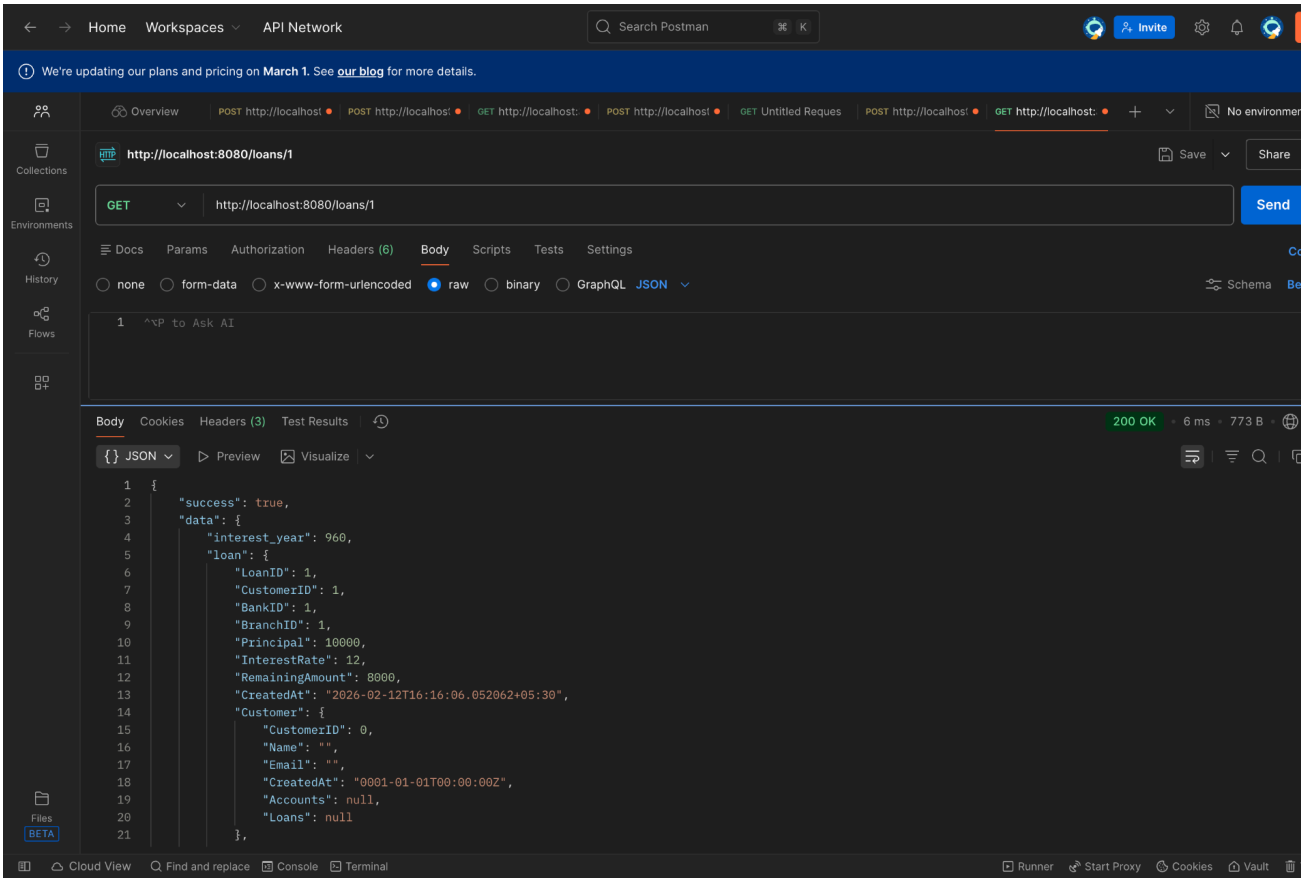
A) SANTOSH TAKES A LOAN OF 10,000 (AUTO 12% RATE APPLIED)



B) SANTOSH REPAYS 2000 OF THAT AMOUNT



C) VIEW LOAN DETAILS



I separated:

- **Account activity (transactions)**
- **Loan lifecycle (loan + repayment history)**

This makes system scalable and normalized.

PRINCIPLE

To create this project I followed **SOLID PRINCIPLES ->**

- **S — Single Responsibility Principle (SRP):**

  Here I designed 4 Layers and each layer only perform one task so only have 1 reason to change

  Handler Layer → Service Layer → Repository Layer → Database

  ❖ ***Handler Layer (API Layer)***

  Purpose:

  - Accept HTTP request
  - Validate input
  - Bind JSON
  - Send response

  No business logic here.

  Because handlers should only deal with HTTP concerns.

  —----------

  ❖ ***Service Layer (Business Logic)***

  This is the brain of the system.

  Examples:

  - Deposit validation
  - Prevent negative withdrawal
  - Check branch belongs to bank
  - Check customer exists before opening account
  - Apply 12% interest on loan creation
  - Use transactions for atomic operations

  It does NOT:

  - Handle HTTP
  - Write raw SQL

❖ *Repository Layer (Database Access)*

Purpose:

- Only DB operations
- No business logic

Examples:

- Create()
- GetByID()
- UpdateBalance()

This makes system testable and clean.

❖ *Database Layer*

PostgreSQL + GORM

- Auto migration
- Foreign key support
- ORM mapping
- Clean struct-to-table mapping

- You change database → only repository changes
- You change business rules → only service changes
- You change API format → only handler changes

Thus there is separation of work.

- **O — Open/Closed Principle (OCP):**

I should be able to add new behavior without breaking old code.

OCP is followed in service layer in interfaces i created like in **account interface** We have **deposit()**,**withdraw()** , In future we can add new behaviors like **FreezeAccount()** etc and also can add various new account type like Current_account,joint_account, we can extend logic in service.

- **L — Liskov Substitution Principle (LSP):**

  If a class implements an interface, it must behave correctly when substituted.

  In my project:
  Example:
  type AccountRepository interface {
  Create()
  GetByID()
  }

  If later I create:
  MockAccountRepository

  For testing.
  It must behave same as real repository.
  Your code already allows this because:
  Services depend on interface, not concrete struct.
  That satisfies LSP.

- **Interface Segregation Principle (ISP)**

  Do not force a class to implement methods it does not use.

  Example:
  Instead of one huge repository:
  BankingRepository

  I created:
  AccountRepository
  LoanRepository
  CustomerRepository
  BranchRepository

  Each interface is small and focused.
  Services only depend on what they need.

Example:
AccountService depends only on:

- AccountRepository
- TransactionRepository
- CustomerRepository

It does NOT depend on LoanRepository.

- ## D — Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules.
Both should depend on abstractions.

In My Project AccountService
depends on
AccountRepository (interface)

Then in main.go:

accountRepo := repositories.NewAccountRepository(dbConn)
accountService := services.NewAccountService(accountRepo, ...)

Instead of direct **Account Service using GORM** there is layer of abstraction called Repository(Interface)

I followed SOLID principles to ensure maintainability and scalability. I separated concerns using a layered architecture where handlers manage HTTP, services manage business rules, and repositories manage persistence. All services depend on abstractions rather than concrete implementations, allowing future extension without modifying core logic. This ensures clean code, easier testing, and better scalability.