**Internal wallet transactional system (API)**

1. **Based on relationships every entity e.g. User, Team, Stock or any other should have their own defined "wallet" to which we could transfer money or withdraw.**

**Explanations:**

My code leverages **polymorphic associations** in the **Wallet** model. Each **User, Team, or Stock** must have its own **wallet**, and money transfers should be possible between these wallets. Below are the parts from the provided code that address this.

*# app/models/wallet.rb*

*class Wallet < ApplicationRecord*

*belongs\_to :owner, polymorphic: true # Allows multiple entities to own a wallet*

*end*

* This means that **every entity** (like User, Team, or Stock) can own a wallet.
* When a Wallet is created, it must specify which entity (owner) it belongs to using the **owner** polymorphic relationship.

Each **User, Team, and Stock** inherits from Entity and can have a **wallet**:

*# app/models/entity.rb*

*class Entity < ApplicationRecord*

*self.abstract\_class = true # Prevents direct instantiation*

*has\_one :wallet, as: :owner, dependent: :destroy # Association to Wallet*

*end*

**Example Entity Models:**

*# app/models/user.rb*

*class User < Entity; end*

*# app/models/team.rb*

*class Team < Entity; end*

*# app/models/stock.rb*

*class Stock < Entity; end*

* The has\_one :wallet, as: :owner association ensures that **each entity has its own wallet**.
* **dependent: :destroy** ensures that if an entity (e.g., a User) is deleted, its wallet will also be removed.

You can try this in the **Rails console** to verify everything works

*wallet1 = Wallet.create(owner: User.create(name: 'Alice'), balance: 1000.0)*

*wallet2 = Wallet.create(owner: User.create(name: 'Bob'), balance: 500.0)*

*transaction = Transaction.create(amount: 100, source\_wallet: wallet1, target\_wallet: wallet2)*

*puts "Wallet 1 Current Balance: #{wallet1.current\_balance}"*

*puts "Wallet 2 Current Balance: #{wallet2.current\_balance}"*

**Create a Wallet for a User CURL Command:**

curl -X POST http://localhost:3000/wallets -H "Content-Type: application/json" -d "{ \"wallet\": { \"balance\": 1000.0, \"owner\_type\": \"User\", \"user\": { \"name\": \"Ryan\" } } }"

curl -X POST http://localhost:3000/wallets -H "Content-Type: application/json" -d "{ \"wallet\": { \"balance\": 2000.0, \"owner\_type\": \"Team\", \"user\": { \"name\": \"Alice\" } } }"

curl -X POST http://localhost:3000/wallets -H "Content-Type: application/json" -d "{ \"wallet\": { \"balance\": 3000.0, \"owner\_type\": \"Stock\", \"user\": { \"name\": \"Bob\" } } }"

**Retrieve a Wallet's Balance CURL Command:**

curl -X GET http://localhost:3000/wallets/1/balance

**Retrieve All Wallets CURL Command:**

curl -X GET http://localhost:3000/wallets

**Create a Transaction (Money Transfer) CURL Command:**

curl -X POST http://localhost:3000/transactions -H "Content-Type: application/json" -d "{ \"transaction\": { \"amount\": 100.0, \"source\_wallet\_id\": 1, \"target\_wallet\_id\": 2 } }"

**Retrieve All Transactions CURL Command:**

curl -X GET http://localhost:3000/transactions

**Delete All Data For Testing Purpose CURL Command:**

curl -X DELETE http://localhost:3000/reset

1. **Every request for credit/debit (deposit or withdraw) should be based on records in database for given model**

The **Transaction model** ensures that every **debit (withdraw)** and **credit (deposit)** operation happens only through a **database record**. You can check my code solutions in this file **app/models/transaction.rb**

* **Every transaction is stored in the transactions table**.
* The **after\_create callback** ensures that wallets are **only updated after a valid transaction** has been saved in the database.
* **Wallet.transaction** ensures that either **all operations succeed** (both wallets are updated) or **none** (rollback on failure).
* The **lock!** method prevents other processes from modifying the wallet balance during the transaction, ensuring **data integrity**.

This migration ensures that **each transaction** is stored in the database with references to the wallets involved, file => 20241018094533\_create\_transactions.rb

*class CreateTransactions < ActiveRecord::Migration[7.2]*

*def change*

*create\_table :transactions do |t|*

*t.decimal :amount, null: false # Amount of the transaction*

*t.references :source\_wallet, null: true, foreign\_key: { to\_table: :wallets }*

*t.references :target\_wallet, null: true, foreign\_key: { to\_table: :wallets }*

*t.timestamps*

*end*

*end*

*end*

1. **Every instance of a single transaction should have proper validations against required fields and their source and targetwallet, e.g. from who we are taking money and transferring to whom? (Credits == source wallet == nil, Debits == targetwallet == nil)**

In app/models/transaction.rb, The **valid\_transaction** method ensures that:

* **Credit transactions** (deposits) must have:

**source\_wallet == nil**

**target\_wallet present**

* **Debit transactions** (withdrawals) must have:

**source\_wallet present**

**target\_wallet == nil**

**Invalid transactions** (both or neither wallets specified) are blocked with appropriate error messages. This implementation ensures that **all transactions are properly validated** before they are recorded in the database

1. **Each record should be created in database transactions to comply with ACID standards**

My code in app/models/transaction.rb uses **ActiveRecord's transaction block** to ensure **ACID compliance** (Atomicity, Consistency, Isolation, and Durability) when handling **wallet balance updates**.

**\*Atomicity**

The transaction will succeed or fail **as a whole**. If any part of the operation fails, **the entire transaction is rolled back**.

*Wallet.transaction do # Atomic operation starts*

*source\_wallet.lock! if source\_wallet*

*target\_wallet.lock! if target\_wallet*

*# Adjust balances*

*source\_wallet.update!(balance: source\_wallet.balance - amount) if source\_wallet*

*target\_wallet.update!(balance: target\_wallet.balance + amount) if target\_wallet*

*end*

**\* Consistency**

The database moves from one consistent state to another after each transaction. Transactions with invalid amounts or both/missing wallets are **not allowed**. Wallet balances are only adjusted within a **valid transaction**.

*validate :valid\_transaction*

*def valid\_transaction*

*errors.add(:source\_wallet, "must be present") if source\_wallet.nil? && target\_wallet.present?*

*errors.add(:target\_wallet, "must be present") if target\_wallet.nil? && source\_wallet.present?*

*end*

**\*Isolation**

Concurrent transactions should not interfere with each other. **lock!** ensures that the wallet rows are **locked for the duration of the transaction**. No other process can modify these wallets until the transaction completes, **preventing race conditions** or inconsistent updates.

*source\_wallet.lock! if source\_wallet*

*target\_wallet.lock! if target\_wallet*

***\**Durability**

ActiveRecord ensures durability, once a transaction is committed, it is **permanently recorded in the database**, even in the case of system failures. Rails relies on the **underlying database’s transaction mechanism** (SQLite, PostgreSQL, etc.) to ensure that **committed transactions are durable**.

1. **Balance for given entity (User, Team, Stock) should be calculated by summing records**

I create the **Wallet** model to dynamically calculate the balance from related **Transaction** records in app/models/wallet.rb file:

*class Wallet < ApplicationRecord*

*belongs\_to :owner, polymorphic: true*

*# Relationship with transactions where this wallet is the source or target*

*has\_many :outgoing\_transactions, class\_name: 'Transaction', foreign\_key: 'source\_wallet\_id'*

*has\_many :incoming\_transactions, class\_name: 'Transaction', foreign\_key: 'target\_wallet\_id'*

*# Calculate the current balance by summing all transactions*

*def current\_balance*

*incoming\_sum = incoming\_transactions.sum(:amount) # Total credits*

*outgoing\_sum = outgoing\_transactions.sum(:amount) # Total debits*

*incoming\_sum - outgoing\_sum # Final balance*

*end*

*end*

**After each transaction**, the **update\_wallets** method updates the wallets involved using the **current\_balance** method. **Credits and Debits** are recorded in the transactions table, and their sum reflects the wallet's balance. The **balance is not just a stored value** but is dynamically calculated by **querying the transactions table**.

1. **Architect generic wallet solution (money manipulation) between entities (User, Stock, Team or any other)**

My **generic wallet solution** consists of:

1. **Polymorphic Wallet Model**, The **belongs\_to :owner, polymorphic: true** in **app/models/wallet.rb**, allows a wallet that can belong to **any type of entity** (User, Stock, Team).
2. **Transaction Model**, Each wallet tracks **incoming and outgoing transactions,** handles **money manipulation** (debit, credit, and transfers) between wallets.
3. **Entity Models (User, Team, Stock, etc.)**, With **has\_one :wallet**, each entity type (User, Stock, or Team) can **own a wallet** and participate in transactions. New entity types can be added (example, Vendor, Customer) without modifying the wallet model.
4. **Create model relationships and validations for achieving proper calculations of every wallet, transactions**

The **relationships** between **Wallets (**app/models/wallet.rb) and **Transactions** (app/models/transaction.rb) ensure that all transactions are tracked accurately.

* **Each wallet can track both outgoing and incoming transactions** through:

**outgoing\_transactions**: Represents **debits** (money taken out).

**incoming\_transactions**: Represents **credits** (money added).

* **Dynamic Balance Calculation**:

The **current\_balance method** sums all transactions related to the wallet to ensure the balance is accurate.

The **validations** in the Transaction model (app/models/transaction.rb) ensure that **only valid transactions** are recorded, preventing issues like missing wallets or negative balances.

* Ensures that only **positive amounts** are allowed in transactions (amount > 0).
* A **valid transaction** must have at least **one wallet (source or target)**.
* Both **source and target wallets cannot be empty or both present**, ensuring clarity in credit or debit operations.

1. **Use STI (or any other design pattern) for proper money manipulation**

My code solution leverages **Single Table Inheritance (STI)** as a design pattern to ensure **proper money manipulation**. This allows different types of entities (such as **User, Team, Stock**) to share a common behavior while keeping the system modular, reusable, and extendable.

The **Entity** class (app/models/entity.rb) serves as the **base class** for **STI**. Subclasses such as **User, Team, or Stock** inherit from it, and all data is stored in a **single entities table**. The **User, Team, and Stock** models inherit from Entity and thus share common behaviors. This structure allows for easy **money manipulation** across different types of entities.

1. **Apply your own sign in (new session solution, no sign up is needed) without any external gem**

The **SessionsController** (app/controllers/sessions\_controller.rb) from my code, implements a **custom sign in solution** without using external gems like Devise. This solution relies on **basic session handling** to create and destroy user sessions.

In **config/routes.rb**, I have define routes for **sign in (login)** and **sign out (logout)**:

*# config/routes.rb*

*Rails.application.routes.draw do*

*post '/login', to: 'sessions#create' # Route for logging in*

*delete '/logout', to: 'sessions#destroy' # Route for logging out*

*end*

* **Login Process:**
* The user sends a **POST request to /login** with their **name** (since no password is required by the requirements).
* If a user with the given name exists, their **ID is stored in the session**.
* The response confirms a successful login.
* **Logout Process:**
  + The user sends a **DELETE request to /logout**.
  + The session is **cleared**, logging the user out.

**Test Login CURL Command:**

curl -X POST "http://localhost:3000/login" -H "Content-Type: application/json" -d "{\"name\": \"Cherry Romandiaz\"}"

**Test Logout CURL Command:**

curl -X DELETE "http://localhost:3000/logout"

1. **Apply Create a LatestStockPrice library (in lib folder in “gem style”) for “price”, “prices” and “price\_all” endpoints - https://rapidapi.com/suneetk92/api/latest-stock-price**

My code solution includes a **custom library** located in the **lib/ folder** with functions that interact with the **Latest Stock Price API** in lib/latest\_stock\_price.rb file.

You can try this in the **Rails console** to verify everything works:

*stock\_price = LatestStockPrice.new*

*puts stock\_price.price('ZOMA.NS')*

*puts stock\_price.prices(['ZOMA.NS', 'TATADVRA.NS'])*

*puts stock\_price.price\_all*

**Single Stock CURL Command:**

curl -X GET "http://localhost:3000/stocks/price?symbol=ZOMA.NS"

**Multiple Stocks CURL Command:**

curl -X GET "http://localhost:3000/stocks/prices?symbols=ZOMA.NS,TATADVRA.NS,GODRCONS.NS"

**All Stocks CURL Command:**

curl -X GET http://localhost:3000/stocks/price\_all

**THANK YOU AND ENJOY MY CODES ^\_^**