DB Desgin

Database Schema Documentation

Users Table

- **Purpose**: Stores information about users.
- **Columns**:
  - `user_id` (SERIAL, PRIMARY KEY): Unique identifier for each user.
  - `email` (VARCHAR(255), UNIQUE): Email address of the user.
  - `firebase_id` (VARCHAR(255), UNIQUE): Firebase ID for user authentication.
  - `name` (VARCHAR(100)): Name of the user.
  - `created_at` (TIMESTAMP): Timestamp of when the user was created.

Groups Table

- **Purpose**: Stores details about groups.
- **Columns**:
  - `group_id` (SERIAL, PRIMARY KEY): Unique identifier for each group.
  - `group_name` (VARCHAR(255)): Name of the group.
  - `created_by` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user who created the group.
  - `created_at` (TIMESTAMP): Timestamp of when the group was created.

Group Members Table

- **Purpose**: Manages the members of each group.
- **Columns**:
  - `group_member_id` (SERIAL, PRIMARY KEY): Unique identifier for each group membership.
  - `user_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user in the group.
  - `group_id` (INT, NOT NULL, FOREIGN KEY): References `groups(group_id)`; the group to which the user belongs.
  - `joined_date` (TIMESTAMP): Timestamp when the user joined the group.

Payment Methods Table

- **Purpose**: Stores information about payment methods used by users.
- **Columns**:
  - `payment_id` (SERIAL, PRIMARY KEY): Unique identifier for each payment method.
  - `user_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user associated with the payment method.
  - `payment_type` (VARCHAR(50), NOT NULL): Type of payment method (e.g., 'UPI', 'Bank Account').
  - `upi_id` (VARCHAR(100)): UPI ID, used if the payment type is 'UPI'.
  - `account_number` (VARCHAR(20)): Account number, used if the payment type is 'Bank Account'.
  - `ifsc_code` (VARCHAR(15)): IFSC code for bank transfers, used if the payment type is 'Bank Account'.
  - `wallet_provider` (VARCHAR(50)): Optional, provider name for UPI wallet, if applicable.
  - `is_primary` (BOOLEAN, DEFAULT FALSE): Indicates whether this is the primary payment method for the user.
  - `created_at` (TIMESTAMP): Timestamp when the payment method was created.

Transactions Table

- **Purpose**: Stores information about monetary transactions between users.
- **Columns**:
  - `transaction_id` (SERIAL, PRIMARY KEY): Unique identifier for each transaction.
  - `lender_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user who is lending money.
  - `borrower_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user borrowing money.
  - `group_id` (INT, NULL, FOREIGN KEY): References `groups(group_id)`; optional, for transactions within groups.
  - `amount` (DECIMAL(10, 2), NOT NULL): The amount of money involved in the transaction.
  - `status` (VARCHAR(50)): The status of the transaction (e.g., 'pending', 'successful', 'failed', 'retrying').

    - `purpose` (VARCHAR(255)): The purpose of the transaction (e.g., 'Loan', 'Settlement').
    - `payment_method_id` (INT, FOREIGN KEY): References `payment_methods(payment_id)`; payment method used for the transaction.
    - `retry_count` (INT, DEFAULT 0): Tracks the number of retry attempts for a failed transaction.
    - `failure_reason` (TEXT): Describes the failure reason, if the transaction failed.

Transaction Splits Table

- **Purpose**: Manages the breakdown of amounts owed by individual participants in a transaction.
- **Columns**:
    - `transaction_split_id` (SERIAL, PRIMARY KEY): Unique identifier for each split.
    - `transaction_id` (INT, NOT NULL, FOREIGN KEY): References `transactions(transaction_id)`; the related transaction.
    - `user_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user responsible for the split.
    - `amount` (DECIMAL(10, 2), NOT NULL): The amount each user owes in the transaction.

Balances Table

- **Purpose**: Tracks the financial balance of each user, including amounts they owe and lent.
- **Columns**:
    - `balance_id` (SERIAL, PRIMARY KEY): Unique identifier for each balance record.
    - `user_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user whose balance is tracked.
    - `group_id` (INT, NULL, FOREIGN KEY): References `groups(group_id)`; optional, for group-specific balances.
    - `owed_amount` (DECIMAL(10, 2), DEFAULT 0): Amount owed by the user.
    - `lent_amount` (DECIMAL(10, 2), DEFAULT 0): Amount the user has lent.

Requests Table

- **Purpose**: Stores requests for money between users.
- **Columns**:
  - `request_id` (SERIAL, PRIMARY KEY): Unique identifier for each request.
  - `sender_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user sending the request.
  - `receiver_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user receiving the request.
  - `group_id` (INT, NULL, FOREIGN KEY): References `groups(group_id)`; optional, for group-specific requests.
  - `amount` (DECIMAL(10, 2), NOT NULL): The amount requested.
  - `status` (VARCHAR(50), DEFAULT 'pending'): The status of the request (e.g., 'pending', 'accepted', 'rejected').
  - `created_at` (TIMESTAMP): Timestamp when the request was created.

Settlements Table

- **Purpose**: Tracks settlements of debts between users.
- **Columns**:
  - `settlement_id` (SERIAL, PRIMARY KEY): Unique identifier for each settlement.
  - `user_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user making the settlement.
  - `counterparty_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user being settled with.
  - `group_id` (INT, NULL, FOREIGN KEY): References `groups(group_id)`; optional, for group-specific settlements.
  - `amount` (DECIMAL(10, 2), NOT NULL): The amount settled.
  - `settlement_date` (TIMESTAMP): Timestamp of the settlement.

Transaction Logs Table

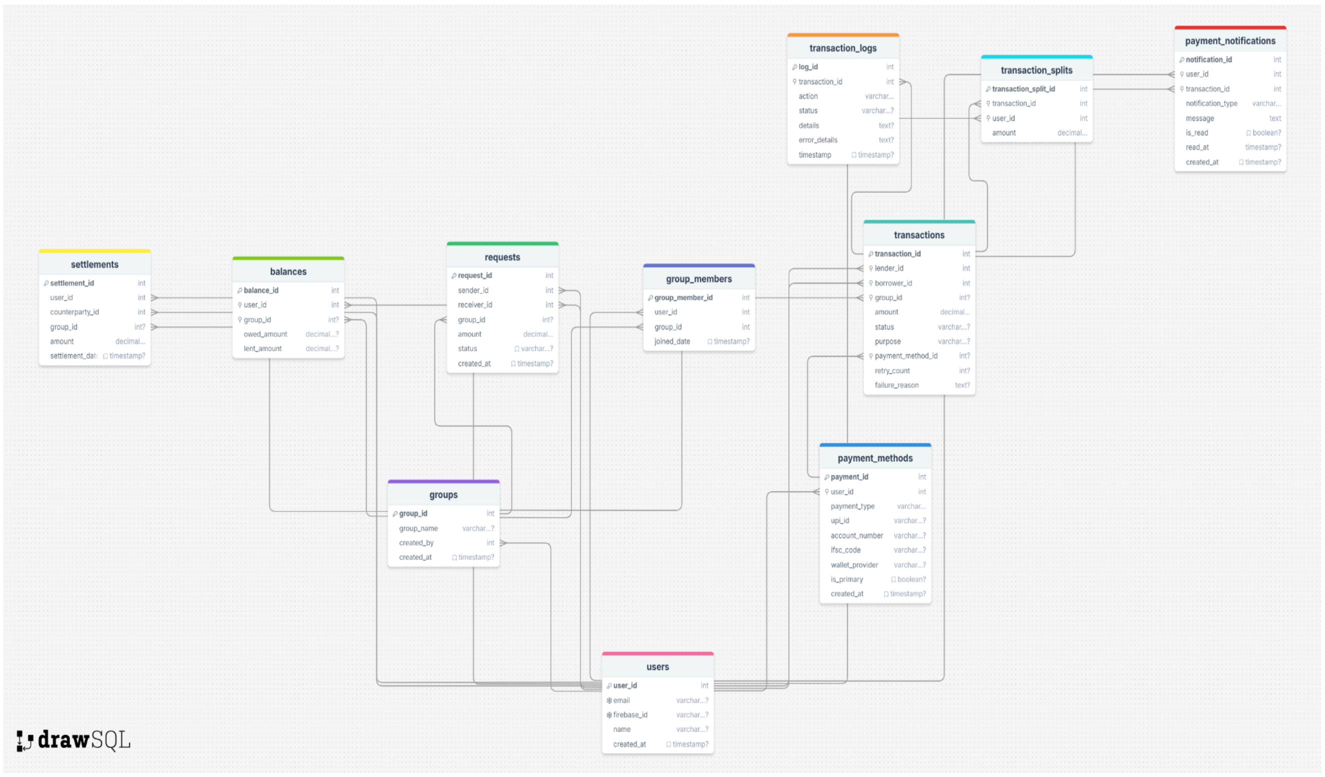
- **Purpose**: Logs actions taken on transactions (e.g., initiated, successful, failed).
- **Columns**:
  - `log_id` (SERIAL, PRIMARY KEY): Unique identifier for each log entry.

- `transaction_id` (INT, NOT NULL, FOREIGN KEY): References `transactions(transaction_id)`; the transaction related to the log entry.
- `action` (VARCHAR(100), NOT NULL): The action taken (e.g., 'Payment Initiated', 'Payment Successful', 'Payment Failed').
- `status` (VARCHAR(50)): The status of the action (e.g., 'Pending', 'Success', 'Failed').
- `details` (TEXT): Additional details related to the action.
- `error_details` (TEXT): Specific error details in case of failure.
- `timestamp` (TIMESTAMP): Timestamp of when the action occurred.

Payment Notifications Table

- **Purpose**: Stores notifications for users about transaction statuses.
- **Columns**:
  - `notification_id` (SERIAL, PRIMARY KEY): Unique identifier for each notification.
  - `user_id` (INT, NOT NULL, FOREIGN KEY): References `users(user_id)`; the user to whom the notification is sent.
  - `transaction_id` (INT, NOT NULL, FOREIGN KEY): References `transactions(transaction_id)`; the related transaction.
  - `notification_type` (VARCHAR(50), NOT NULL): The type of notification (e.g., 'Success', 'Failure', 'Pending').
  - `message` (TEXT, NOT NULL): The content of the notification message.
  - `is_read` (BOOLEAN, DEFAULT FALSE): Indicates if the user has read the notification.
  - `read_at` (TIMESTAMP): Timestamp of when the notification was read.
  - `created_at` (TIMESTAMP): Timestamp when the notification was created.

Image


transaction_logs
log_id int
transaction_id int
action varchar...
status varchar...?
details text?
error_details text?
timestamp timestamp?
transaction_splits
transaction_split_id int
transaction_id int
user_id int
amount decimal...
payment_notifications
notification_id int
user_id int
transaction_id int
notification_type varchar...
message text
is_read boolean?
read_at timestamp?
created_at timestamp?
settlements
settlement_id int
user_id int
counterparty_id int
group_id int?
amount decimal...
settlement_date timestamp?
balances
balance_id int
user_id int
group_id int?
owed_amount decimal...?
lent_amount decimal...?
requests
request_id int
sender_id int
receiver_id int
group_id int?
amount decimal...
status varchar...?
created_at timestamp?
group_members
group_member_id int
user_id int
group_id int
joined_date timestamp?
transactions
transaction_id int
lender_id int
borrower_id int
group_id int?
amount decimal...
status varchar...?
purpose varchar...?
payment_method_id int?
retry_count int?
failure_reason text?
payment_methods
payment_id int
user_id int
payment_type varchar...
upi_id varchar...?
account_number varchar...?
ifsc_code varchar...?
wallet_provider varchar...?
is_primary boolean?
created_at timestamp?
groups
group_id int
group_name varchar...?
created_by int
created_at timestamp?
users
user_id int
email varchar...?
firebase_id varchar...?
name varchar...?
created_at timestamp?
drawSQL

Flow

Step 1: User Creation

The process begins with creating user accounts for Abhinav, X, and Y.

Users Table (each user has unique identifiers and contact details):

- **Abhinav**:
  - `user_id`: 1
  - `email`: abhinav@example.com
  - `firebase_id`: abhinav123
  - `name`: Abhinav
- **X**:
  - `user_id`: 2
  - `email`: x@example.com
  - `firebase_id`: x123
  - `name`: X
- **Y**:
  - `user_id`: 3
  - `email`: y@example.com
  - `firebase_id`: y123
  - `name`: Y

Purpose:

These users are added to the `users` table when they sign up through Firebase or email registration.

Step 2: Adding Payment Methods

Now, let's add payment methods (UPI and Bank details) for each user.

Payment Methods Table:

- **Abhinav**:

  - `user_id`: 1
  - `payment_method_type`: UPI
  - `upi_id`: abhinav@upi
  - `bank_name`: NULL
  - `account_number`: NULL
  - `ifsc_code`: NULL
  - **Abhinav's Bank Details**:
  - `user_id`: 1
  - `payment_method_type`: Bank
  - `upi_id`: NULL
  - `bank_name`: "State Bank of India"
  - `account_number`: "1234567890"
  - `ifsc_code`: "SBIN0001234"
- **X**:
  - `user_id`: 2
  - `payment_method_type`: UPI
  - `upi_id`: x@upi
  - `bank_name`: NULL
  - `account_number`: NULL
  - `ifsc_code`: NULL
- **Y**:
  - `user_id`: 3
  - `payment_method_type`: Bank
  - `upi_id`: NULL
  - `bank_name`: "HDFC Bank"

    - `account_number`: "9876543210"
    - `ifsc_code`: "HDFC0001234"

Step 3: Creating Groups and Adding Members

Next, we create a group where Abhinav, X, and Y will participate.

Groups Table:

- **Group**: "Dinner with Friends"
    - `group_id`: 1
    - `group_name`: "Dinner with Friends"
    - `created_by`: 1 (Abhinav)

Group Members Table:

- **Abhinav**:
    - `group_member_id`: 1
    - `user_id`: 1 (Abhinav)
    - `group_id`: 1
- **X**:
    - `group_member_id`: 2
    - `user_id`: 2 (X)
    - `group_id`: 1
- **Y**:
    - `group_member_id`: 3
    - `user_id`: 3 (Y)
    - `group_id`: 1

Step 4: Recording a Group Transaction

Abhinav decides to pay ₹3,000 for the dinner and wants to split it between the three of them.

Transactions Table:

- **Transaction**: Abhinav pays ₹3,000 for the dinner.
    - `transaction_id`: 2
    - `lender_id`: 1 (Abhinav)

    - `borrower_id`: NULL (Group transaction)
    - `group_id`: 1 ("Dinner with Friends")
    - `amount`: ₹3,000
    - `purpose`: "Dinner"
    - `status`: "Completed"

Transaction Splits Table:

- **Abhinav** (Lender):
    - `transaction_split_id`: 1
    - `transaction_id`: 2
    - `user_id`: 1 (Abhinav)
    - `amount`: ₹0 (since Abhinav paid the entire amount)
- **X** (Owes ₹1,000):
    - `transaction_split_id`: 2
    - `transaction_id`: 2
    - `user_id`: 2 (X)
    - `amount`: ₹1,000
- **Y** (Owes ₹1,000):
    - `transaction_split_id`: 3
    - `transaction_id`: 2
    - `user_id`: 3 (Y)
    - `amount`: ₹1,000

Step 5: Balances Update

After the group transaction is created, balances are updated for each user to track how much they owe Abhinav.

Balances Table:

- **Abhinav's Balance**:
    - `user_id`: 1 (Abhinav)
    - `group_id`: 1 (Dinner group)
    - `owed_amount`: ₹0
    - `lent_amount`: ₹3,000 (because Abhinav paid the full ₹3,000)

- **X's Balance**:
  - `user_id`: 2 (X)
  - `group_id`: 1 (Dinner group)
  - `owed_amount`: ₹1,000 (X owes ₹1,000)
  - `lent_amount`: ₹0
- **Y's Balance**:
  - `user_id`: 3 (Y)
  - `group_id`: 1 (Dinner group)
  - `owed_amount`: ₹1,000 (Y owes ₹1,000)
  - `lent_amount`: ₹0

Step 6: Settling the Transaction

At this point, X and Y decide to pay Abhinav the amounts they owe. These payments can be made through the selected payment methods: UPI or Bank.

Settlements Table:

- **X's Settlement**:
  - `settlement_id`: 1
  - `user_id`: 2 (X)
  - `counterparty_id`: 1 (Abhinav)
  - `group_id`: 1 (Dinner group)
  - `amount`: ₹1,000 (X pays ₹1,000 to Abhinav)
- **Y's Settlement**:
  - `settlement_id`: 2
  - `user_id`: 3 (Y)
  - `counterparty_id`: 1 (Abhinav)
  - `group_id`: 1 (Dinner group)
  - `amount`: ₹1,000 (Y pays ₹1,000 to Abhinav)

After settlement, the balances are updated:

Updated Balances Table:

- **Abhinav's Balance**:
    - `owed_amount`: ₹0
    - `lent_amount`: ₹0 (because the full ₹3,000 is now paid back)
- **X's Balance**:
    - `owed_amount`: ₹0 (X has fully repaid)
    - `lent_amount`: ₹0
- **Y's Balance**:
    - `owed_amount`: ₹0 (Y has fully repaid)
    - `lent_amount`: ₹0

Step 7: Notifications and Transaction Logs

Notifications will be sent to users (Abhinav, X, Y) informing them of the completed transaction and successful payments.

Transaction Logs Table:

Logs can include entries like:

- **Log**: "Dinner transaction created with ₹3,000 paid by Abhinav."
- **Log**: "X settled ₹1,000 with Abhinav."
- **Log**: "Y settled ₹1,000 with Abhinav."

Payment Notifications Table:

- **Abhinav**: Notification of payment received from X and Y.
- **X**: Notification of successful settlement with Abhinav.
- **Y**: Notification of successful settlement with Abhinav.

Step 8: Final Summary of the Entire Process

- **Users**: Abhinav, X, and Y create accounts and add payment methods (UPI/Bank).
- **Groups**: Abhinav creates a group "Dinner with Friends" and adds X and Y.
- **Transactions**: Abhinav pays ₹3,000 for the dinner, splitting the cost with X and Y.
- **Balances**: The balances are updated to reflect how much each user owes.
- **Settlements**: X and Y repay their share of ₹1,000 each to Abhinav.

- **Notifications and Logs**: Users are notified of the transaction and payments, and logs capture the entire flow of the transaction.
