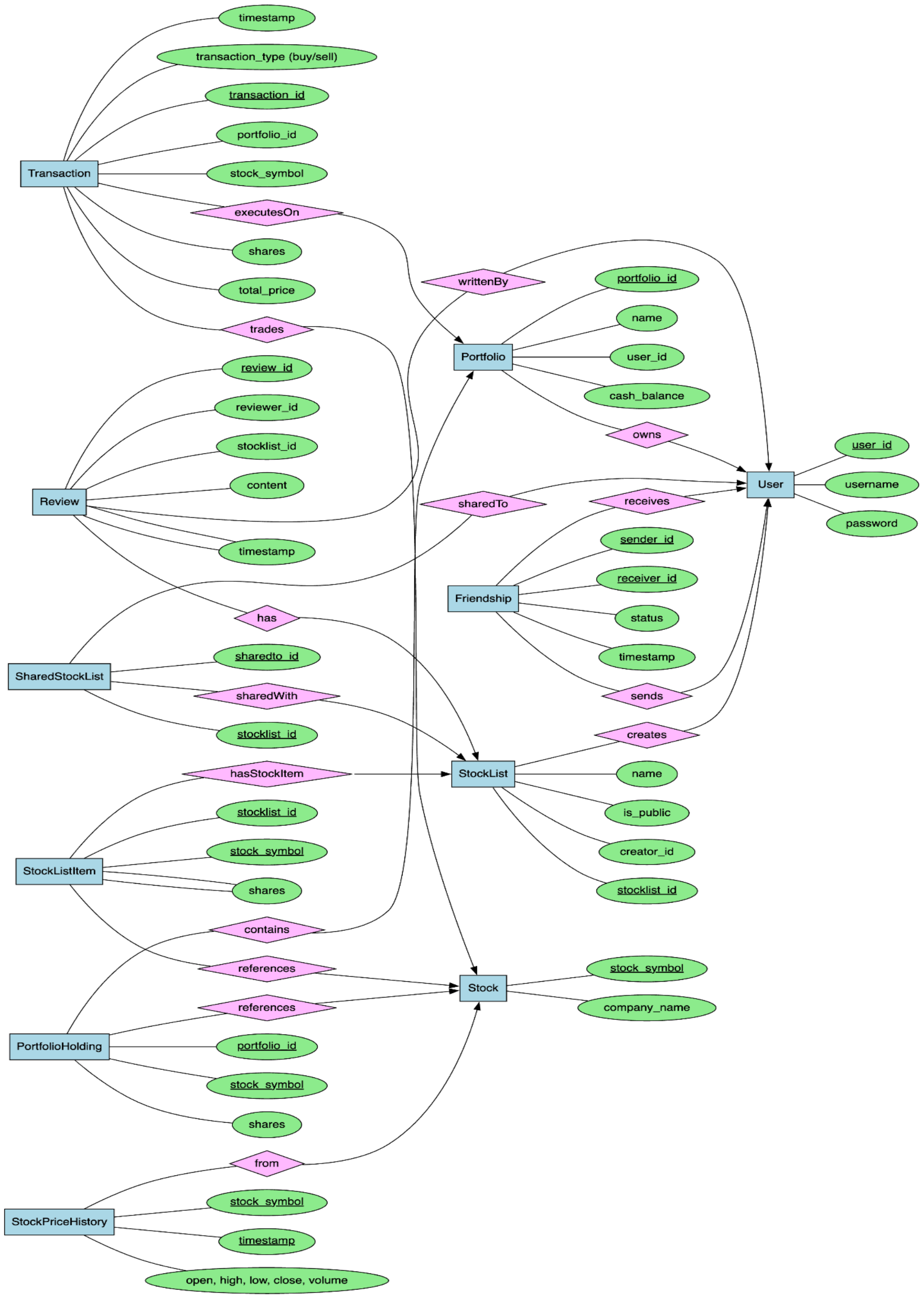
E/R Diagram


timestamp
transaction_type (buy/sell)
transaction_id
portfolio_id
Transaction
stock_symbol
executesOn
shares
writtenBy
total_price
portfolio_id
name
trades
Portfolio
user_id
review_id
cash_balance
reviewer_id
owns
stocklist_id
user_id
content
User
username
Review
sharedTo
receives
password
timestamp
sender_id
receiver_id
Friendship
has
status
sharedto_id
timestamp
SharedStockList
sharedWith
sends
stocklist_id
creates
hasStockItem
StockList
name
stocklist_id
is_public
stock_symbol
StockListItem
creator_id
shares
stocklist_id
contains
references
Stock
stock_symbol
references
company_name
PortfolioHolding
portfolio_id
stock_symbol
shares
from
stock_symbol
StockPriceHistory
timestamp
open, high, low, close, volume

Relational Schema:

users(user_id, username, password)
friendships(sender_id, receiver_id, status, timestamp)
stocklists(stocklist_id, name, is_public, creator_id)
sharedstocklists(stocklist_id, sharedto_id)
reviews(review_id, reviewer_id, stocklist_id, content, timestamp)
stocks(stock_symbol, company_name)
stocklistitems(stocklist_id, stock_symbol, shares)
portfolios(portfolio_id, name, user_id, cash_balance)
portfolioholdings(portfolio_id, stock_symbol, shares)
transactions(transaction_id, portfolio_id, stock_symbol, shares, total_price, timestamp, type)
stockpricehistory(stock_symbol, timestamp, open, high, low, close, volume)

Normalization:

In our relational schema, we have many functional dependencies (FDs) that are straightforward and stem directly from primary keys. For instance, in relations like User or Friendship, we have user_id → username, password and sender_id receiver_id → status, timestamp. The left-hand side of the FDs corresponds to a unique key such that no two tuples will have the same value of. Thus, they trivially determine all other attributes in the relation (because they are the one and only tuple), and so all normal forms, including BCNF and 3NF, are satisfied.

An interesting case arises in the Review relation. Under the assumption that a reviewer can only submit one review per stock list, the composite key we have (reviewer_id, stocklist_id) → content, timestamp. But since we can only have one review, it also implies review_id. Thus, (reviewer_id, stocklist_id) is a superkey, and thus the left hand side of a FD involving it satisfying BCNF. On the other hand, if we instead assume that multiple reviews can be submitted by the same reviewer for a stock list, then (reviewer_id, stocklist_id) is no longer a key. But under this relaxed assumption, the FD (reviewer_id, stocklist_id) → content, timestamp would also not hold, as each review might have different content. Thus, no BCNF violation occurs in either scenario. In both interpretations, the relation remains normalized with no redundant or inconsistent dependencies.

Explanation of SQL Queries:

Throughout the project, we used the declarative SQL language for CRUD (create, read update, delete) operations to our database. Below, we summarize a selection of SQL queries we used for the function of our application.

Login Query:
Validates credentials and returns the user if found:

```
SELECT user_id, username FROM users
WHERE username = $1 AND password = $2 —$1 and $2 are placeholders
```

Register Query
Inserts a new user only if the username doesn't already exist:

```
INSERT INTO users (username, password)
VALUES ($1, $2)
RETURNING user_id, username
```

Preventing Duplicate requests after being deleted or rejected:
Checks existing friendships and uses a 5-minute cooldown:

```
SELECT * FROM friendships
WHERE ((sender_id = $1 AND receiver_id = $2)
    OR (sender_id = $2 AND receiver_id = $1))
  AND status = 'rejected' —this covers the rejected or deleted case
  AND last_timestamp >= NOW() - INTERVAL '5 minutes'
```

Send Request
Deletes reversed rejections before inserting a new request (avoiding duplicates and ensuring consistency):

```
DELETE FROM friendships
WHERE sender_id = $2 AND receiver_id = $1;

INSERT INTO friendships (sender_id, receiver_id, status)
VALUES ($1, $2, 'pending')
ON CONFLICT DO UPDATE
SET status = 'pending', last_timestamp = CURRENT_TIMESTAMP
```

Create and Share Stocklists
Creates a new stocklist with a public boolean, linking it to the creator. Also allowing that list to be shared with others if it is private.

```
INSERT INTO stocklists (name, is_public, creator_id)
VALUES ($1, $2, $3)

INSERT INTO sharedstocklists (stocklist_id, sharedto_id)
```

```
ON CONFLICT DO NOTHING —just allow duplicate shares to go through
```

Add Stocks from List
We add by stock symbol and the number of shares to add. If the stock is already in our database we just augment the number of shares.

```
INSERT INTO stocklistitems (stocklist_id, stock_symbol, shares)
VALUES ($1, $2, $3)
ON CONFLICT (stocklist_id, stock_symbol) DO UPDATE
SET shares = stocklistitems.shares + EXCLUDED.shares
```

Create Review with Duplication Check

```
SELECT 1 FROM reviews
WHERE reviewer_id = $1 AND stocklist_id = $2;

INSERT INTO reviews (reviewer_id, stocklist_id, content)
VALUES ($1, $2, $3)
```

Buy/Sell Logic with Holdings and Cash Constraints

```
SELECT close FROM stockpricehistory
WHERE symbol = $1 ORDER BY timestamp DESC LIMIT 1;

UPDATE portfolios SET cash_balance = cash_balance - $1
WHERE portfolio_id = $2;

UPDATE portfolioholdings SET shares = shares + $1
WHERE portfolio_id = $2 AND stock_symbol = $3;
```

Logging All Transactions

```
INSERT INTO transactions (portfolio_id, stock_symbol, shares,
total_price, the_timestamp, trans_type)
VALUES ($1, $2, $3, $4, CURRENT_TIMESTAMP, $5)
```

Pivoting Time Series Data for Stats
We take this data to compute covariance and correlation matrix.

```
SELECT ph.stock_symbol, sph.the_timestamp, sph.close
```

```
FROM portfolioholdings ph
JOIN stockpricehistory sph ON sph.symbol = ph.stock_symbol
WHERE ph.portfolio_id = $1
```