ISTANBUL TECHNICAL UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

BLG 317E

Database Systems

Stock Market Viewer Project

GROUP MEMBERS:

150220902 : Nahid Aliyev

912400214 : Akbar Bunyadzade

150210906 : Emil Huseynov

2024-2025 FALL

Contents

1 Introduction

Before delving into the project details, you can test the website with its all functionalities as you walk through the sections with the link: *stockviewer.ns1.icu*

1.1 Purpose

The Stock Market Viewer website is designed to provide real-time stock price information and portfolio management capabilities to users. The system integrates live data from BIST100 (Borsa Istanbul) using an external Akbank API (or alternative APIs), allowing users to view market data in Turkish Lira and manage their investments by buying and selling stocks. The system also includes user management features such as account creation, login, and profile editing. In summary, a user can:

- View general stock information and favorite currencies on the Home page (5 main stocks plus currencies).
- Log in with a username and password (stored in plain text as per requirement).
- Sign up with name, username, password, and email.
- Access a Profile page to update personal data.
- Maintain and manage both Favorite stocks **and** Favorite currencies from the Profile page.
- Track the current value of owned stocks, including real-time price updates and gain/loss calculations for each stock in the portfolio.
- Perform transactions (buying and selling stocks) directly from the Portfolio page, with the ability to specify stock quantity and buy/sell price.
- Observe currency exchange rates from an external source to keep track of conversions and maintain a list of favorite currencies.
- Using nested queries and select functions, Sorting and filtering stocks to find desirable stock are accesible from stock finder page.

1.2 Scope

This project focuses on:

- **Web Interface via Django**: purely for routing, view handling, template rendering, and form processing. All database interactions are through raw SQL queries.

- **PostgreSQL Primary Database**: storing all data for users, stocks, currencies, watchlist entries, and portfolio records.
- **Plain-text password storage**: for demonstration purposes in an educational setting.
- A separate `.sql` file contains all schema (DDL) definitions, such as CREATE TABLE statements, along with CSV data imports.
- **Array-based portfolio** structure to demonstrate non-normalized design. Each row in the `portfolio` table can hold arrays of stock IDs, quantities, and prices.
- **Detailed usage of SQL** to handle all CRUD operations: inserts, updates, and deletes for each entity, as well as advanced array operations in the portfolio.

By adhering to a raw SQL workflow, the project demonstrates a traditional database programming approach while still benefiting from Django's modular structure for web development.

2 Database Design

2.1 ER Diagram

The figure below represents the tables and relationships used in our database. Figure 1 shows an ERD snippet with the following entities: `users`, `favorites`, `portfolio`, `stocks`, `currency`, and `favorite_currency` (which extends the system to handle user-specific currency watchlists).

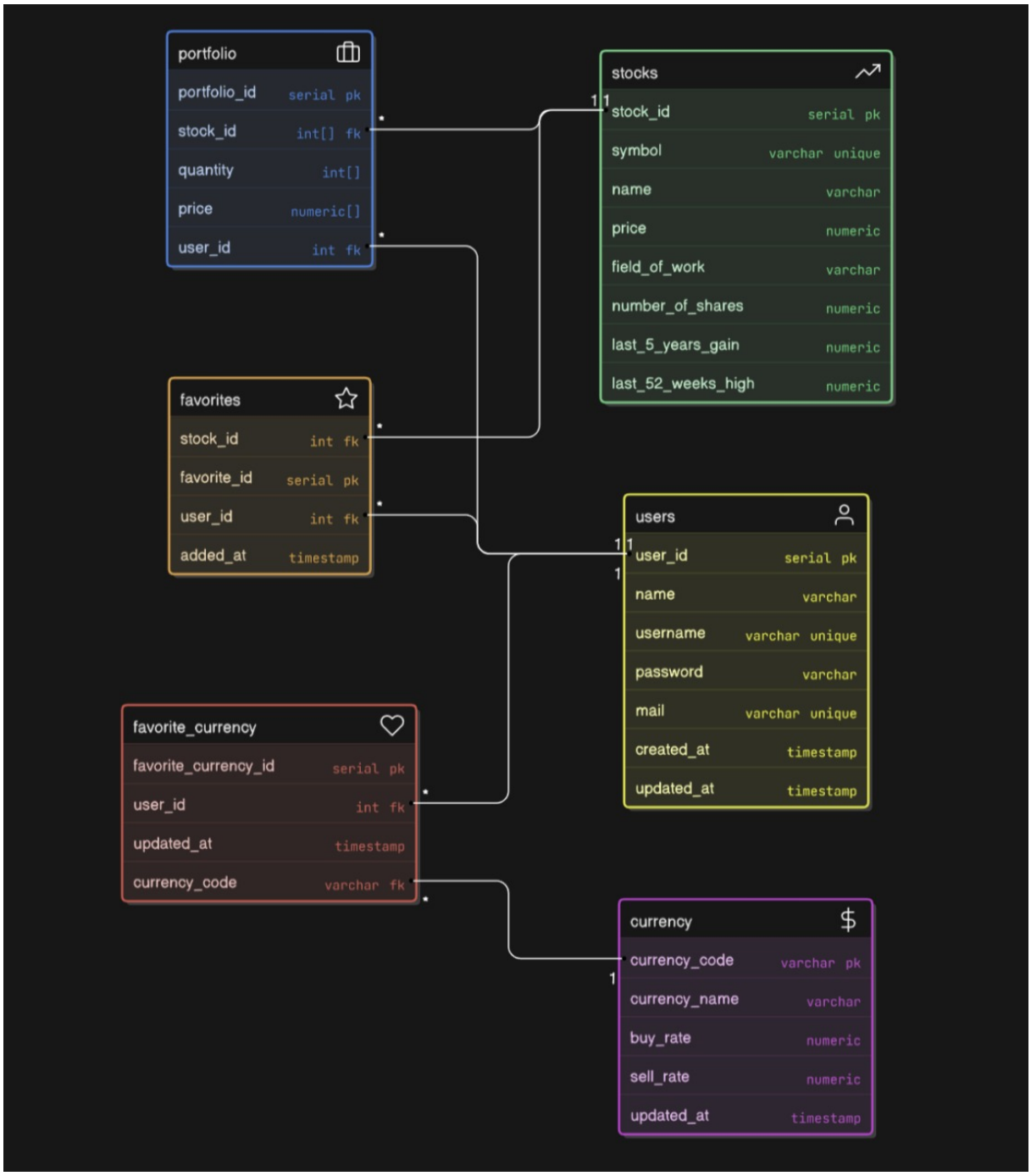

Figure 1: Entity-Relationship Diagram

2.2 SQL DDL (Database Schema)

All table creation commands are stored in a file named `schema.sql`. This file also includes the commands to **load initial BIST100 stock data** from a CSV. Below is the final **updated** schema with arrays in the `portfolio` table and an additional `favorite_currency` table.

Listing 1: DDL for Updated Stock Market DB

```
-- Drop tables if they exist (for clean setup)
DROP TABLE IF EXISTS favorites CASCADE;
DROP TABLE IF EXISTS portfolio CASCADE;
DROP TABLE IF EXISTS users CASCADE;
DROP TABLE IF EXISTS stocks CASCADE;
DROP TABLE IF EXISTS currency CASCADE;
DROP TABLE IF EXISTS favorite_currency CASCADE;

-- Users table
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    name VARCHAR(255),
    username VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL, -- Plain text for demonstration
    mail VARCHAR(255) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT NOW(),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Stocks table
CREATE TABLE stocks (
    stock_id SERIAL PRIMARY KEY,
    symbol VARCHAR(20) UNIQUE NOT NULL,
    name VARCHAR(255),
    price NUMERIC(10,2),
    field_of_work VARCHAR(255),
    number_of_shares NUMERIC(15,2),
    last_5_years_gain NUMERIC(10,2),
    last_52_weeks_high NUMERIC(10,2)
);

-- Favorites table (for stocks)
```

```sql
CREATE TABLE favorites (
    favorite_id SERIAL PRIMARY KEY,
    user_id INT NOT NULL REFERENCES users(user_id) ON DELETE CASCADE,
    stock_id INT NOT NULL REFERENCES stocks(stock_id) ON DELETE CASCADE,
    added_at TIMESTAMP DEFAULT NOW()
);

-- Portfolio table (array-based design)
-- Storing multiple stock_id, quantity, and price values in arrays.
CREATE TABLE portfolio (
    portfolio_id SERIAL PRIMARY KEY,
    user_id INT NOT NULL REFERENCES users(user_id) ON DELETE CASCADE,
    stock_id INT[] NOT NULL,         -- array of stock ids
    quantity INT[] NOT NULL,         -- array of corresponding quantities
    price NUMERIC(10,2)[] NOT NULL   -- array of corresponding buy prices
    -- For demonstration of array usage; not a normalized approach
);

-- Currency table
CREATE TABLE currency (
    currency_code VARCHAR(10) PRIMARY KEY,
    currency_name VARCHAR(255),
    buy_rate NUMERIC(10,4),
    sell_rate NUMERIC(10,4),
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Favorite currency table
CREATE TABLE favorite_currency (
    favorite_currency_id SERIAL PRIMARY KEY,
    user_id INT NOT NULL REFERENCES users(user_id) ON DELETE CASCADE,
    currency_code VARCHAR(10) NOT NULL REFERENCES currency(currency_code) ON
        DELETE CASCADE,
    updated_at TIMESTAMP DEFAULT NOW()
);

-- Load initial BIST100 data into 'stocks' table
\COPY stocks(symbol, name, field_of_work, number_of_shares, last_5_years_gain,
    last_52_weeks_high)
FROM 'app/sql/bist100.csv' DELIMITER ',' CSV HEADER;
```

Key points:

- The `portfolio` table uses **arrays** (`INT[]` and `NUMERIC[]`) to store multiple data points for a single user, demonstrating manual array operations.
- The `favorite_currency` table allows each user to store a list of currencies they want to track, linking to the `currency` table by `currency_code`.
- BIST100 data is **copied** from a CSV file (`bist100.csv`), populating stock entries initially.

3 Implementation Details

3.1 Project Structure (Revisited)

Below is a representative file structure, highlighting where `bist100.csv` is used and how the new `favorite_currency` table is integrated:

```
Database_project/
|-- .gitignore                    # Git ignored files
|-- app/                          # Main application directory
|   |-- __init__.py
|   |-- admin.py                 # Admin interface configuration
|   |-- api_utils.py             # API utilities and helpers
|   |-- apps.py                  # App configuration
|   |-- db.py                    # Raw SQL query helpers
|   |-- management/              # Custom Django commands
|   |   `-- commands/
|   |       |-- update_prices.py
|   |       `-- update_currencies.py
|   |
|   |-- models.py                # (Unused if purely raw SQL, or minimal usage)
|   |-- sql/                     # SQL and data files
|   |   |-- BIST100.csv
|   |   |-- currency.csv
|   |   `-- schema.sql
|   |-- static/                  # Static assets
|   |   `-- app/css/style.css
```

```
|   |-- templates/              # HTML templates
|   |   |-- base.html
|   |   |-- home.html
|   |   |-- login.html
|   |   |-- profile.html
|   |   |-- signup.html
|   |   `-- stock_finder.html
|   |-- tests.py               # Test cases
|   |-- urls.py                # URL routing
|   `-- views/                 # View logic
|       |-- auth_views.py
|       |-- home_views.py
|       |-- portfolio_views.py
|       `-- profile_views.py
|-- manage.py                  # Django CLI utility
`-- stock_viewer/             # Project configuration
    |-- __init__.py
    |-- asgi.py               # ASGI server config
    |-- settings.py           # Project settings
    |-- urls.py               # Root URL config
    `-- wsgi.py               # WSGI server config
```

3.2 Array-Based Portfolio Explanation

Portfolio table with arrays:

- `stock_id INT[]` A list of stock IDs the user owns.
- `quantity INT[]` The corresponding quantities for each stock.
- `price NUMERIC(10,2)[]` The buy prices for each purchased batch of shares.

If a user purchases the same stock multiple times, your code can **append** additional elements to the arrays, or you can identify the matching stock ID and adjust the existing quantity. This approach is flexible for demonstration, though it's *not* normalized. Typically, one would store separate rows in a more normalized design (e.g., a `portfolio_items` table). However, to fulfill the requirement of "storing multiple stocks as a list," this is how we implemented it.

3.3 Favorite Currency Integration

favorite_currency allows a user to track *only* the currencies they care about. For each entry, we store:

- `favorite_currency_id`: Primary key
- `user_id`: References the `users` table
- `currency_code`: Links to `currency.currency_code`
- `updated_at`: Timestamp of the last update

In the UI, a user can **add** or **remove** favorite currencies from their Profile page (similar to how they manage favorite stocks). The home page then displays only the user's favorite currencies (buy and sell rates), alongside other user-specific data.

4 Detailed Use of SQL and Database Operations

In this project, **all** database interactions are handled by **raw SQL** inside Django views or utility functions. Below are highlights of how we construct, parameterize, and execute queries to **Create**, **Read**, **Update**, and **Delete** (CRUD) entries in each table.

4.1 Connection Handling in `db.py`

We use Django's `connection.cursor()` to run queries. A typical pattern is:

Listing 2: execute_query function in db.py

```
from django.db import connection

def execute_query(sql, params=None, fetchone=False, fetchall=False):
    with connection.cursor() as cursor:
        cursor.execute(sql, params or [])
        if fetchone:
            return cursor.fetchone()
        if fetchall:
            return cursor.fetchall()

def execute_insert_or_update(sql, params=None):
    with connection.cursor() as cursor:
        cursor.execute(sql, params or [])
```

By passing `fetchone=True` or `fetchall=True`, we decide whether to retrieve a single row or multiple rows. For inserts or updates, we call `execute_insert_or_update`.

4.2 User Management Queries

4.2.1 Create a New User (Sign Up)

Listing 3: Insert user example

```
INSERT INTO users (name, username, password, mail)
VALUES (%s, %s, %s, %s);
```

In Django, we call `execute_insert_or_update(sql, [name, username, password, mail])`.

4.2.2 Validate Credentials (Login)

Listing 4: Login check example

```
SELECT user_id, username
FROM users
WHERE username = %s AND password = %s;
```

We pass in `(username, password)` from the login form. If a row is returned, the credentials match.

4.2.3 Update User Profile

Listing 5: Update user profile

```
UPDATE users
SET name = %s, mail = %s, updated_at = NOW()
WHERE user_id = %s;
```

If the user also changes password, we include `password = %s` in the SET clause (escaping the percent sign to avoid LaTeX issues).

4.3 Stock Queries

4.3.1 Loading Initial Stocks

The command in `schema.sql` populates `stocks` from `bist100.csv`:

```
\COPY stocks(symbol, name, field_of_work, number_of_shares,
           last_5_years_gain, last_52_weeks_high)
```

```
FROM 'app/sql/bist100.csv' DELIMITER ',' CSV HEADER;
```

4.3.2 Reading Top 5 Stocks

Listing 6: Select top 5 stocks by market value

```
SELECT symbol, name, price
FROM stocks
ORDER BY (number_of_shares * price) DESC
LIMIT 5;
```

We display these on the Home page.

4.3.3 Updating Stock Price

If an external API provides updated share prices, we might do:

Listing 7: Updating a single stock price

```
UPDATE stocks
SET price = %s
WHERE symbol = %s;
```

4.4 Favorite Stocks (favorites table)

4.4.1 Insert a Favorite Stock

Listing 8: Insert favorite stock

```
INSERT INTO favorites (user_id, stock_id)
VALUES (%s, %s);
```

4.4.2 Remove a Favorite Stock

Listing 9: Delete favorite stock

```
DELETE FROM favorites
WHERE user_id = %s
  AND stock_id = %s;
```

4.5 Portfolio Queries (Array-Based)

4.5.1 Create a New Portfolio Row

If a user has no portfolio row yet, we might do:

Listing 10: Create a new portfolio row

```
INSERT INTO portfolio (user_id, stock_id, quantity, price)
VALUES (%s, ARRAY[%s], ARRAY[%s], ARRAY[%s]);
```

We pass the initial stock ID, quantity, and buy price as single-element arrays.

4.5.2 Append to an Existing Array

If a user buys more of the same stock or a new stock, we can:

Listing 11: Append a new stock purchase to existing arrays

```
UPDATE portfolio
SET stock_id = array_append(stock_id, %s),
    quantity = array_append(quantity, %s),
    price    = array_append(price, %s)
WHERE portfolio_id = %s;
```

4.5.3 Partial Sell or Remove

To sell, we might find the index of the matching stock in the arrays, reduce or remove it. E.g.,

Listing 12: Pseudo approach for partial sells

```
-- (Pseudo-step) We'll fetch the current arrays:
SELECT stock_id, quantity, price
FROM portfolio
WHERE user_id = %s;

-- Then in Python, manipulate the arrays (remove index or subtract).
-- Finally, call an UPDATE with the new array values:
UPDATE portfolio
SET stock_id = %s, quantity = %s, price = %s
WHERE portfolio_id = %s;
```

Since we are not using the ORM, all logic to find or remove array elements is done in Python, then re-saved to the database.

4.6 Currency Queries

4.6.1 Insert or Update Currency Exchange Rates

If we fetch live exchange rates, we might run:

Listing 13: Insert or Update currency

```
INSERT INTO currency (currency_code, currency_name, buy_rate, sell_rate)
VALUES (%s, %s, %s, %s)
ON CONFLICT (currency_code)
DO UPDATE
    SET currency_name = EXCLUDED.currency_name,
        buy_rate = EXCLUDED.buy_rate,
        sell_rate = EXCLUDED.sell_rate,
        updated_at = NOW();
```

The `ON CONFLICT` clause ensures we do an *upsert* if the code already exists.

4.6.2 Select Favorite Currencies for a User

Listing 14: Select user's favorite currencies

```
SELECT c.currency_code, c.currency_name, c.buy_rate, c.sell_rate
FROM favorite_currency fc
JOIN currency c ON fc.currency_code = c.currency_code
WHERE fc.user_id = %s;
```

4.6.3 Add / Remove a Favorite Currency

Similar to favorite stocks:

Listing 15: Add favorite currency

```
INSERT INTO favorite_currency (user_id, currency_code)
VALUES (%s, %s);
```

Listing 16: Remove favorite currency

```
DELETE FROM favorite_currency
WHERE user_id = %s
  AND currency_code = %s;
```

5 Screenshots and Demonstration

Below are some screenshots that illustrate the final implementation. The user sees **favorite currencies** on the Home page once logged in, in addition to the top 5 stocks by market value. They can also manage both favorite stocks and currencies on their Profile page, plus maintain an array-based portfolio.

- **Home Page**:

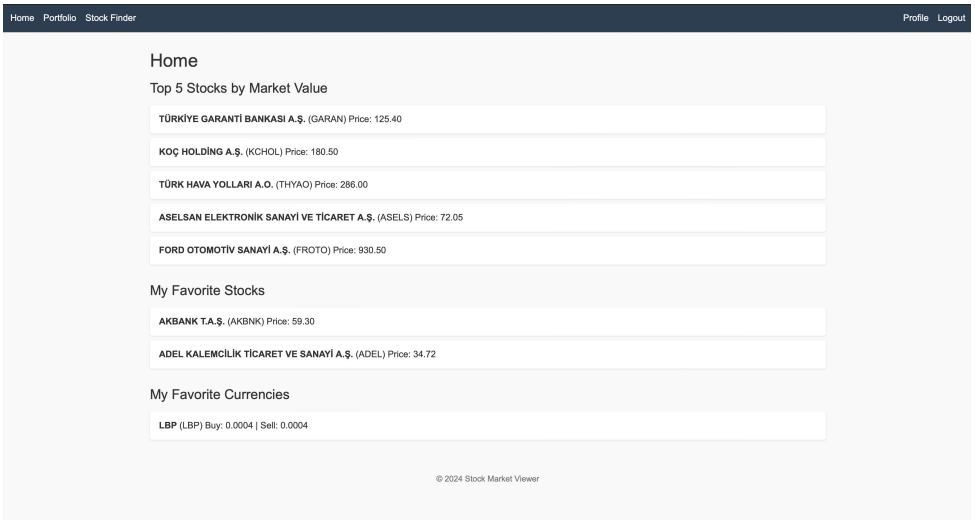

Figure 2: Screenshot of Home page

- **Profile Page**:

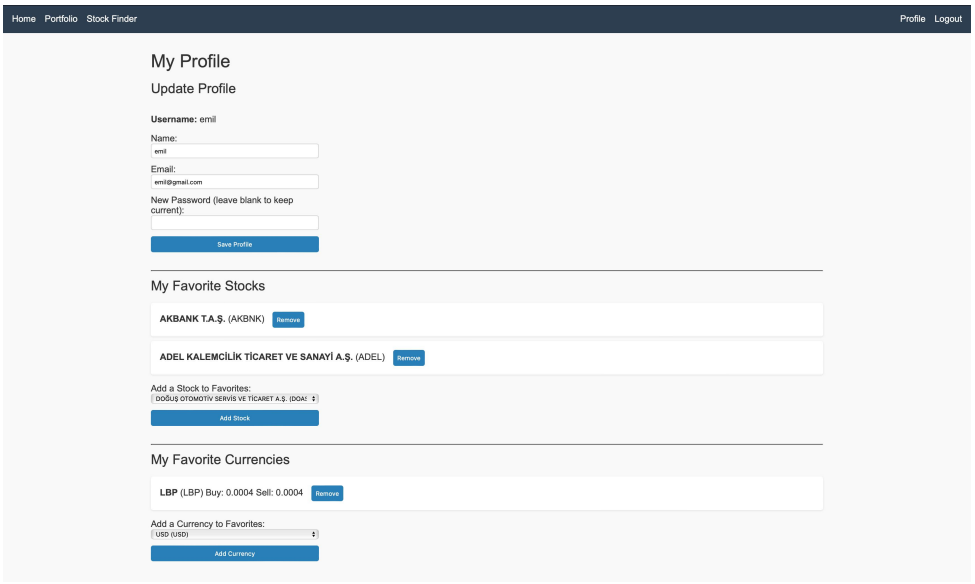

Figure 3: Screenshot Profile page

- **Portfolio Page**:

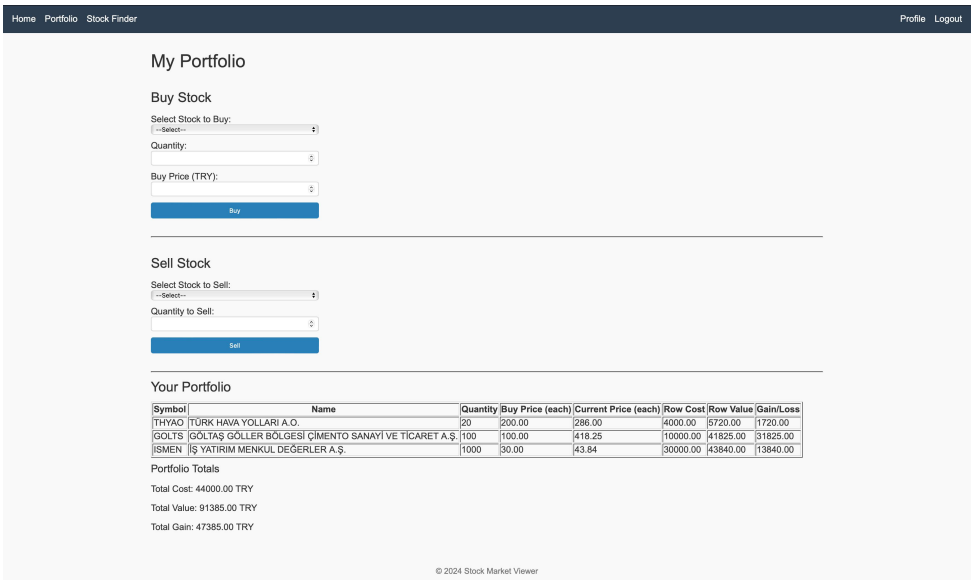

Figure 4: Screenshot of my portfolio page

- **Stock finder Page**:

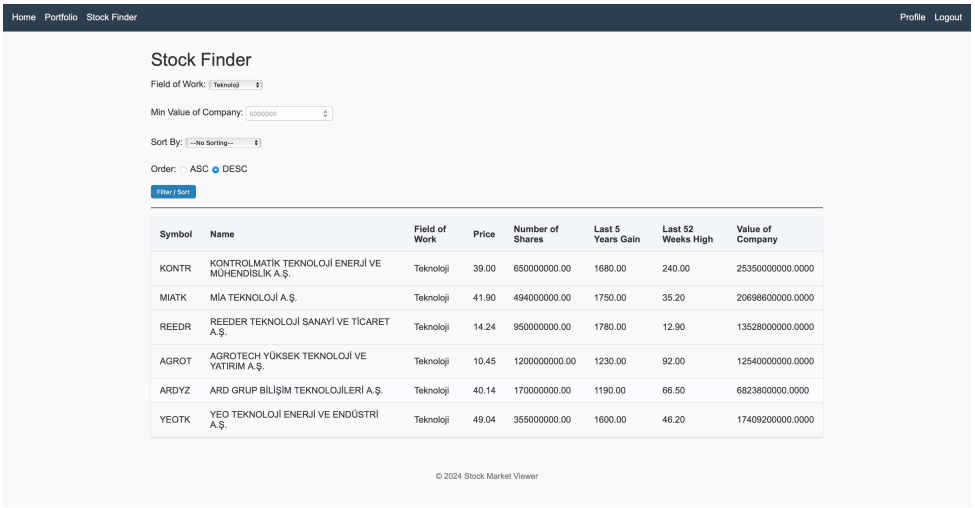

Figure 5: Screenshot of Stock finder page

- **Sign up Page**:

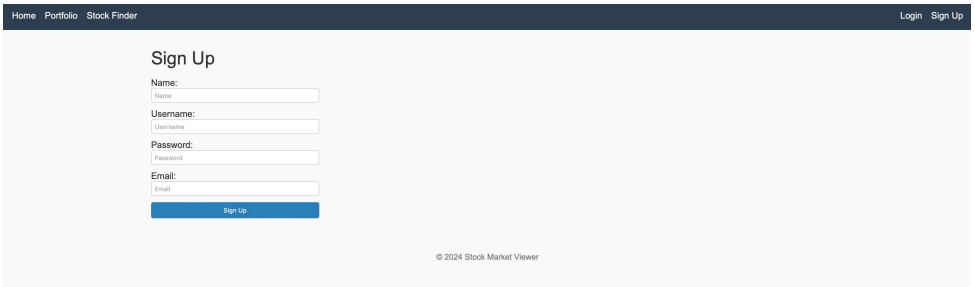

Figure 6: Screenshot of sign up page form

6 Team Responsibilities (Extended)

- **Nahid Aliyev**:
  - Implemented the **front-end pages** (HTML, CSS) for showing favorite currencies on the Home page.
  - Handled all **SQL queries** for the `currency` table, including writing scripts to retrieve data from `doviz.com` or similar APIs for testing purposes.
- **Emil Huseynov**:
  - Designed the **updated** database schema (`schema.sql`) with array-based `portfolio` and `favorite_currency`.
  - Managed forms and user interface flow for portfolio interactions based on arrays (buy / sell).
- **Akbar Bunyadzade**:
  - Implemented the **raw SQL queries** (in `db.py` and `views/*.py`) for adding/removing favorite stocks, updating portfolio arrays.
  - Wrote tests and debugged issues regarding array-based updates in the Portfolio table by figuring out the `favorite_currency` logic in the Profile page.

7 Conclusion

In this project, we demonstrated:

- **PostgreSQL** with **array-based** fields to store multiple stock positions under a single row (though not normalized).
- Employed direct SQL queries (INSERT, SELECT, UPDATE, DELETE, and array-based operations) using connection.cursor() instead of Django's ORM, showcasing low-level control over database interactions.
- Implemented **plain-text credential storage** without hashing to fulfill the specific demonstration requirement, while acknowledg- ing that this approach is not recommended in production.
- Use of an external CSV (`bist100.csv`) for initial population of the `stocks` table.

- **Comprehensive SQL Usage:** Showcased various SQL operations, including upserts (via `ON CONFLICT`), array manipulation (`array_append`), and multi-table joins, forming a robust set of database queries for real-world scenarios.
- **Detailed SQL usage**, from simple inserts/updates to advanced array manipulations in a single row for the `portfolio`.
- **Favorite Currency Feature:** Extended user watchlists to include currency codes, allowing users to store a list of desired currencies in the new `favorite_currency` table and retrieve live buy/sell rates.

While this design meets the project requirements, we note several possible improvements for a production environment in future if necessary:

- **Secure Password Handling:** Transition from plain-text storage to industry-standard cryptographic hashing (e.g., Argon2 or Bcrypt) for all user passwords.
- **Improved Error Handling:** Incorporate robust checks for array insertions, deletions, and partial sells to avoid data inconsistencies or accidental stock misallocations.
- A more advanced mechanism to handle **real-time updates with a minimal lag period** for both stocks and currencies, potentially with streaming data feeds or cron-based tasks.