TANTIV4

PROJECT
ON
STOCK SCRNEER

SUBMITTED

BY

HARSHITHA L

ABSTRACT

The stock screener is a tool designed to quickly identify investment opportunities based on specified criteria, enabling investors to efficiently filter through vast amounts of data and make informed decisions.

INFORMATION

Stock screening has become increasingly crucial in investment decision-making, given the abundance of available data and the need for tools to filter and analyze it efficiently. This introduction provides context on the evolution of stock screeners and their role in empowering investors to navigate the dynamic financial markets effectively.

Frontend:

Html

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Stock Screener</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <div class="container">
      <h1>Stock Screener</h1>
    <div class="filters">
     <label for="priceRange">Price Range:</label>
      <input type="range" id="priceRange" name="priceRange" min="0" max="100" value="50">
        <!-- Add more filter inputs for market cap, dividend yield, etc. -->
      </div>
    <div class="stock-list">          <!-- Stock list will be dynamically populated here -->       </div>
 </div>
<script src="script.js">
</script>
</body>
</html>
```

Explaination

This is the structure of an HTML file for a stock screener web application:

- It starts with the usual HTML5 doc type declaration.
- The <html> element defines the root of the HTML document, with the language attribute set to "en" for English.
- The <head> section contains metadata such as character set, viewport settings, and the page title.
- It links an external CSS file named "styles.css" to style the content.
- The <body> section contains the visible content of the page.
- Inside the <body>, there's a with the class "container" to hold the main content.
- Within the container, there's an <h1> element displaying the title "Stock Screener".
- There's a with the class "filters" to contain filter inputs for the stock screening criteria.
- Currently, it includes a <label> for a price range filter and an <input> element of type "range" to select the price range.
- The comment prompts to add more filter inputs for market cap, dividend yield, etc.
- Beneath the filters, there's a with the class "stock-list" where the list of stocks will be dynamically populated.
- The <script> tag at the end links an external JavaScript file named "script.js" for adding interactivity and functionality to the page.

CSS

```
/* styles.css */
body {    font-family: Arial, sans-serif;
  margin: 0;
  padding: 0;
}
.container {
   max-width: 800px;
    margin: 0 auto;
   padding: 20px;
}
h1 {
   text-align: center;
}
.filters {
  margin-bottom: 20px;
}
.filters label {
   font-weight: bold;
}
.stock-list {
   /* Styling for the stock list */
}
```

Explaination

- body: Sets the font family to Arial or sans-serif for better readability, and removes default margin and padding for a cleaner layout.
- container: Defines a maximum width of 800 pixels, centers it horizontally on the page, and adds padding around its content for spacing.
- h1: Aligns the heading to the center of its container for better presentation.
- filters: Adds a margin at the bottom of the filter section to separate it from other content.
- filters label: Sets the font weight to bold for the labels within the filters section to make them stand out.
- stock-list: This section is currently left empty, indicating that it's ready for future styling specific to the stock list display.

JS

```
// script.js//
 Sample stock data (replace with actual data fetched from backend)
const stocks = [
   { symbol: 'AAPL', name:
 'Apple Inc.', price: 135.37,
marketCap: 2275.02, dividendYield: 0.63 },
   { symbol: 'MSFT', name:
 'Microsoft Corporation', price: 239.91, marketCap: 1809.14, dividendYield: 0.83 },
    { symbol: 'GOOGL', name:
'Alphabet Inc.', price: 2210.88, marketCap: 1475.29, dividendYield: 0 },
 // Add more sample data as needed];
// Function to filter stocks based on criteria
const filterStocks = (criteria) => {
   // Implement filtering logic based on criteria
 // For now, return all stocks    return stocks;};
// Function to render filtered stocksconst renderStocks =
(filteredStocks) => {
   const stockListElement =
 document.querySelector('.stock-list');
   stockListElement.innerHTML
= ''; // Clear previous content
    filteredStocks.forEach(stock => {
    const stockElement = document.createElement('div');
     stockElement.classList.add('stock-item');
      stockElement.innerHTML = `
```

```
 <div class="symbol">${stock.symbol}</div>
  <div class="name">${stock.name}</div>
  <div class="price">$${stock.price}</div>
   <div class="market-cap">$${stock.marketCap}B</div>
  <div class="dividend-yield">${stock.dividendYield}%</div>
     `;       stockListElement.appendChild(stockElement);    });
};
// Event listener for input change (e.g., price
     range)document.getElementById('priceRange').addEventListener('in
     put', (event) => {
   const priceRangeValue = event.target.value;
   // Example: Filter stocks where price is less than or equal to the price
     range value    const filteredStocks = stocks.filter(stock => stock.price
     <= priceRangeValue);
  renderStocks(filteredStocks);
});
// Initially render all stocksrenderStocks(stocks);
```

Explaination

- This JavaScript code is for a simple stock filtering and rendering application.
- Sample Stock Data: The stocks array holds sample data for stocks, including their symbol, name, price, market capitalization, and dividend yield.
- Filtering Function: The filterStocks function takes a criteria parameter (not implemented yet) and is intended to filter stocks based on that criteria. Currently, it returns all stocks.
- Rendering Function: The renderStocks function takes an array of filtered stocks and renders them on the webpage. It clears any previous content and creates HTML elements for each stock, displaying its symbol, name, price, market cap, and dividend yield.
- Event Listener: An event listener is set up for changes in the input with the ID priceRange. When the input value changes (e.g., a price range is selected), it filters the stocks based on the price range and calls the renderStocks function to update the displayed stocks accordingly.
- Initial Rendering: Initially, all stocks are rendered on the webpage by calling renderStocks with the stocks array.

Backend

```
from flask import Flask, request, jsonify
import requests
app = Flask(_name_)
# Define API endpoint for fetching stock data
API_KEY = 'your_api_key_here‘
STOCK_API_URL = 'https://api.example.com/stocks‘
@app.route('/stocks',
methods=['GET'])
def get_stocks():
 # Retrieve filtering criteria from request parameters
min_price = request.args.get('min_price')
 max_price = request.args.get('max_price')
min_market_cap = request.args.get('min_market_cap')
max_market_cap = request.args.get('max_market_cap')
 # Make API request to fetch stock data
 params = {
```

```
 'api_key': API_KEY,
 'min_price': min_price,
 'max_price': max_price,
'min_market_cap': min_market_cap,
 'max_market_cap': max_market_cap
}
 response = requests.get(STOCK_API_URL, params=params)
 data = response.json()
 # Filter stocks based on criteria    filtered_stocks = filter_stocks(data)
return jsonify(filtered_stocks)
def filter_stocks(stocks):
# Implement filtering logic here based on criteria
filtered_stocks = []
 for stock in stocks:
  # Example filtering: price range and market cap range
if float(stock['price']) >= float(min_price) and \
float(stock['price']) <= float(max_price) and \
 float(stock['market_cap']) >= float(min_market_cap) and \
  float(stock['market_cap']) <= float(max_market_cap):
 filtered_stocks.append(stock)
 return filtered_stocks
if_name_ == '_main_':
app.run(debug=True)
```

Explaination

- Imports: We import necessary modules including Flask for creating the web server and handling HTTP requests, and requests for making HTTP requests to external APIs.
- Initialization: We initialize a Flask application.
- Endpoint Definition: We define a route /stocks that handles GET requests. This endpoint expects query parameters for filtering criteria such as min_price, max_price, min_market_cap, and max_market_cap.
- Fetching Stock Data: Inside the endpoint function, we retrieve the filtering criteria from the request parameters and construct a request to a hypothetical stock API. We pass the filtering criteria as query parameters in the request.
- Filtering Stocks: Upon receiving the response from the stock API, we parse the JSON data and filter the stocks based on the provided criteria. The filter_stocks function contains the filtering logic. In this example, we only filter based on price range and market capitalization range.
- Response: Finally, we return the filtered stocks as a JSON response.
- Filtering Function: The filter_stocks function iterates over the list of stocks and applies the filtering logic based on the criteria provided. This function can be expanded to include additional filtering criteria as needed.
- This code provides a basic framework for handling HTTP requests, fetching data from an external API, filtering the data based on specified criteria, and returning the filtered results. It's a starting point that can be expanded and customized further based on specific requirements and the actual stock data API being used.

Database

Connection between C++ and database

```
#include <iostream>
#include <sqlite3.h>
int main() {
sqlite3* db;
int rc = sqlite3_open("stocks.db", &db);
if (rc) {
std::cerr << "Error opening database: " << sqlite3_errmsg(db) << std::endl;
return 1;
 }
  const char* sql = "SELECT * FROM StockData";
 sqlite3_stmt* stmt;
rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
 if (rc != SQLITE_OK) {
std::cerr << "Error preparing statement: " << sqlite3_errmsg(db) << std::endl;
return 1;
}
while (sqlite3_step(stmt) == SQLITE_ROW) {
 // Process each row of the result set
// Example: Retrieve data using sqlite3_column_* functions
 // e.g., int id = sqlite3_column_int(stmt, 0);
}
sqlite3_finalize(stmt);
sqlite3_close(db);
return 0;
}
```

Table

```
CREATE TABLE StockData (
    id INTEGER PRIMARY KEY,
    symbol TEXT NOT NULL,
    date DATE NOT NULL,
    open REAL,
    high REAL,
    low REAL,
    close REAL,
    volume INTEGER,
    dividend_yield REAL,
    market_cap REAL,
 -- Add more fields as needed for additional criteria);
```

Explaination

- Including Headers : The program includes the necessary headers, <iostream> for input/output operations and <sqlite3.h> for SQLite database access.
- Main Function : This is the starting point of the program where execution begins.
- Opening Database Connection : It opens a connection to the SQLite database file named "stocks.db".
- Error Handling for Database Opening : If there's an error opening the database, it prints an error message and exits the program.
- Preparing SQL Statement : It prepares an SQL statement for execution. In this case, it's a SELECT query to retrieve data from a table.
- Error Handling for Statement Preparation : If there's an error preparing the SQL statement, it prints an error message and exits the program.
- Loop Through Result Set : It iterates through the result set obtained by executing the prepared SQL statement. Inside the loop, you would typically process each row of the result set.

THANK YOU