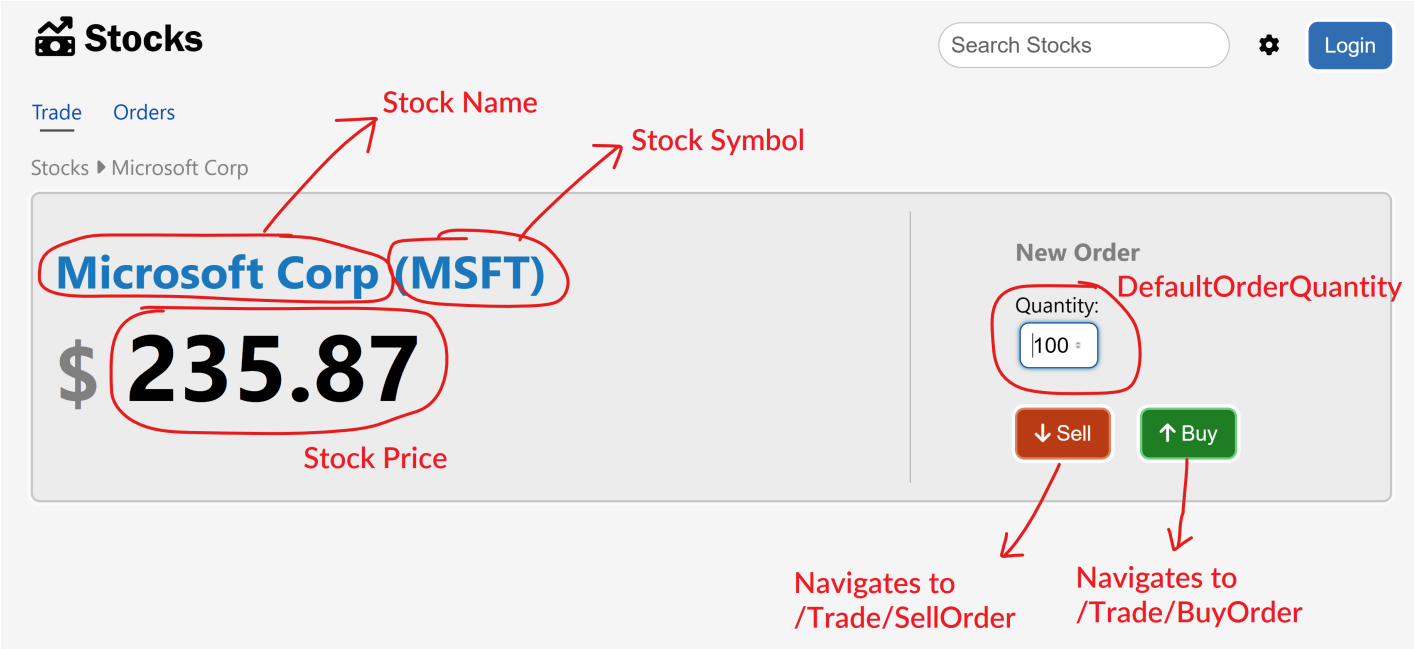
Requirement

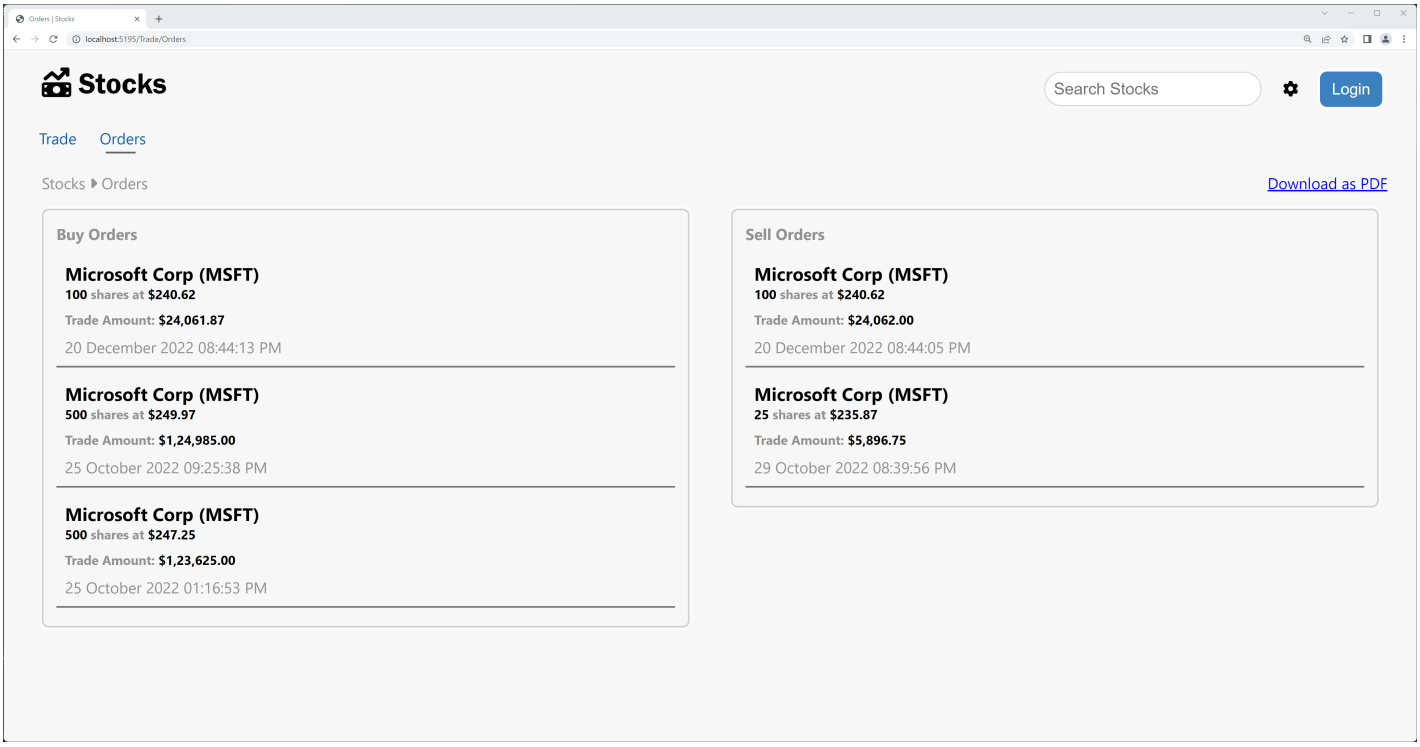
Create an Asp.Net Core Web Application that displays stock price with live updates from "**https://finnhub.io/**".

UI Design

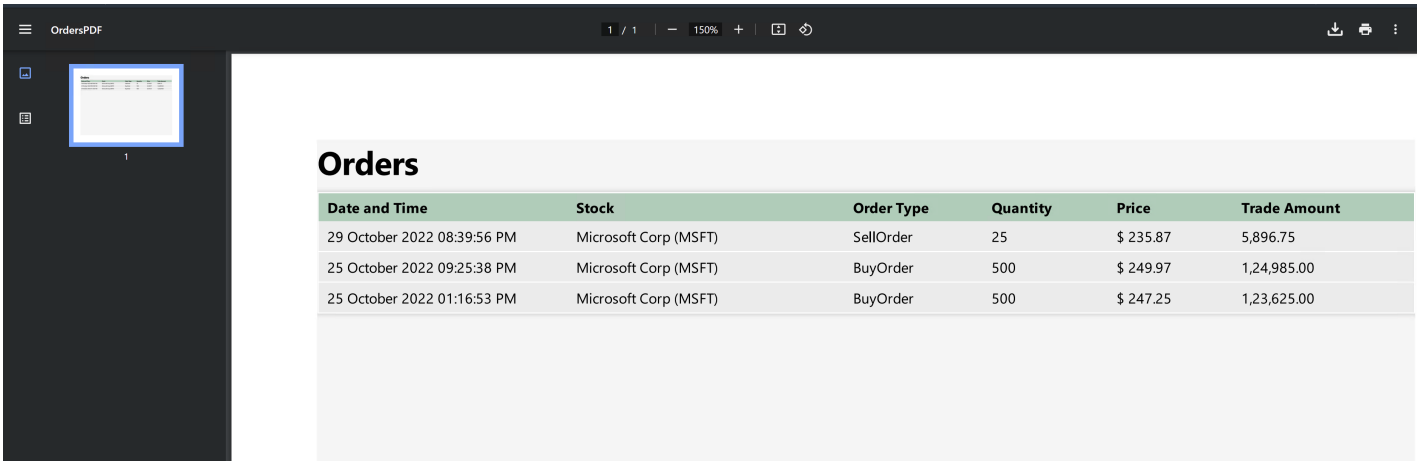
Trade/Index:



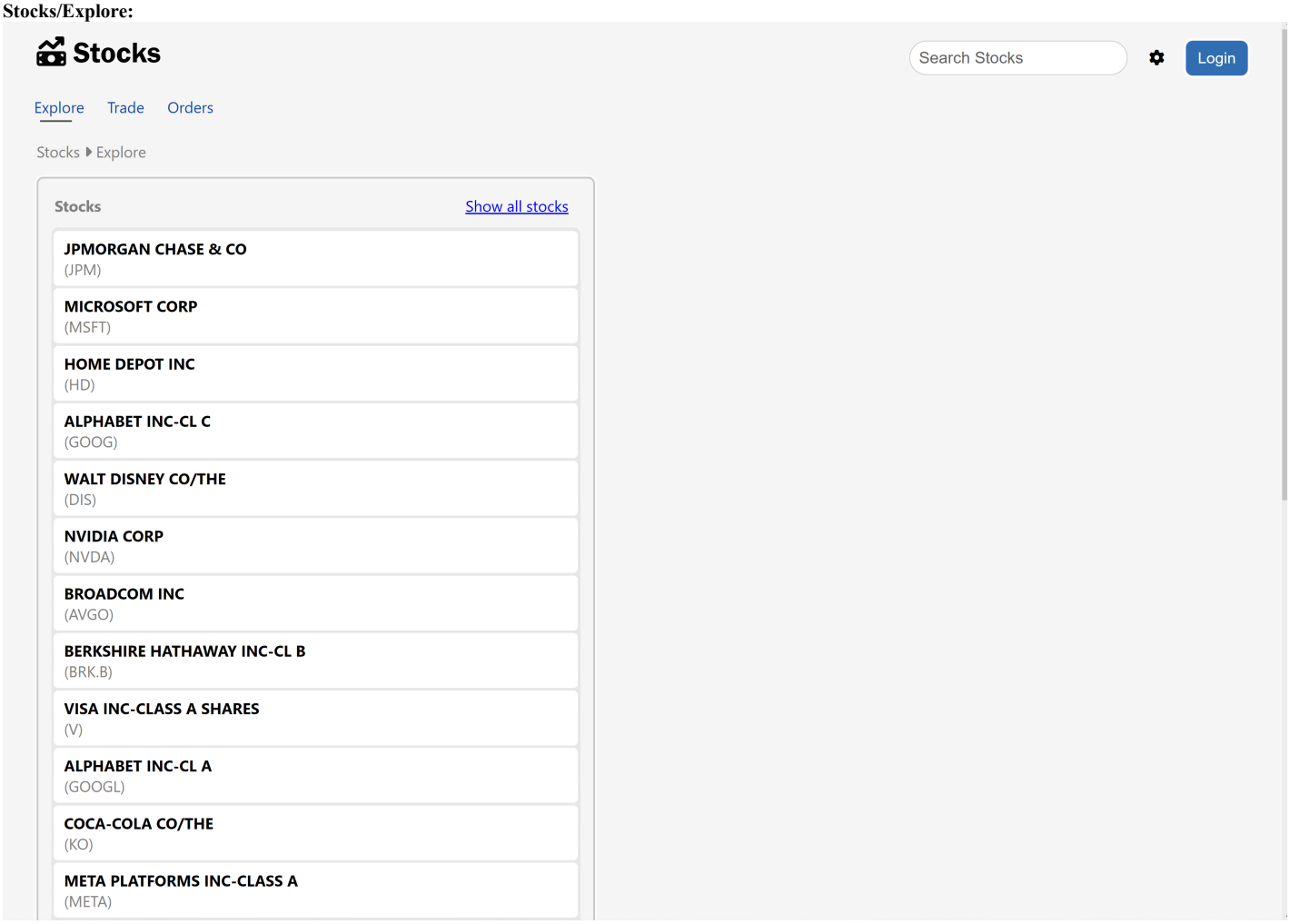
Trade/Orders:



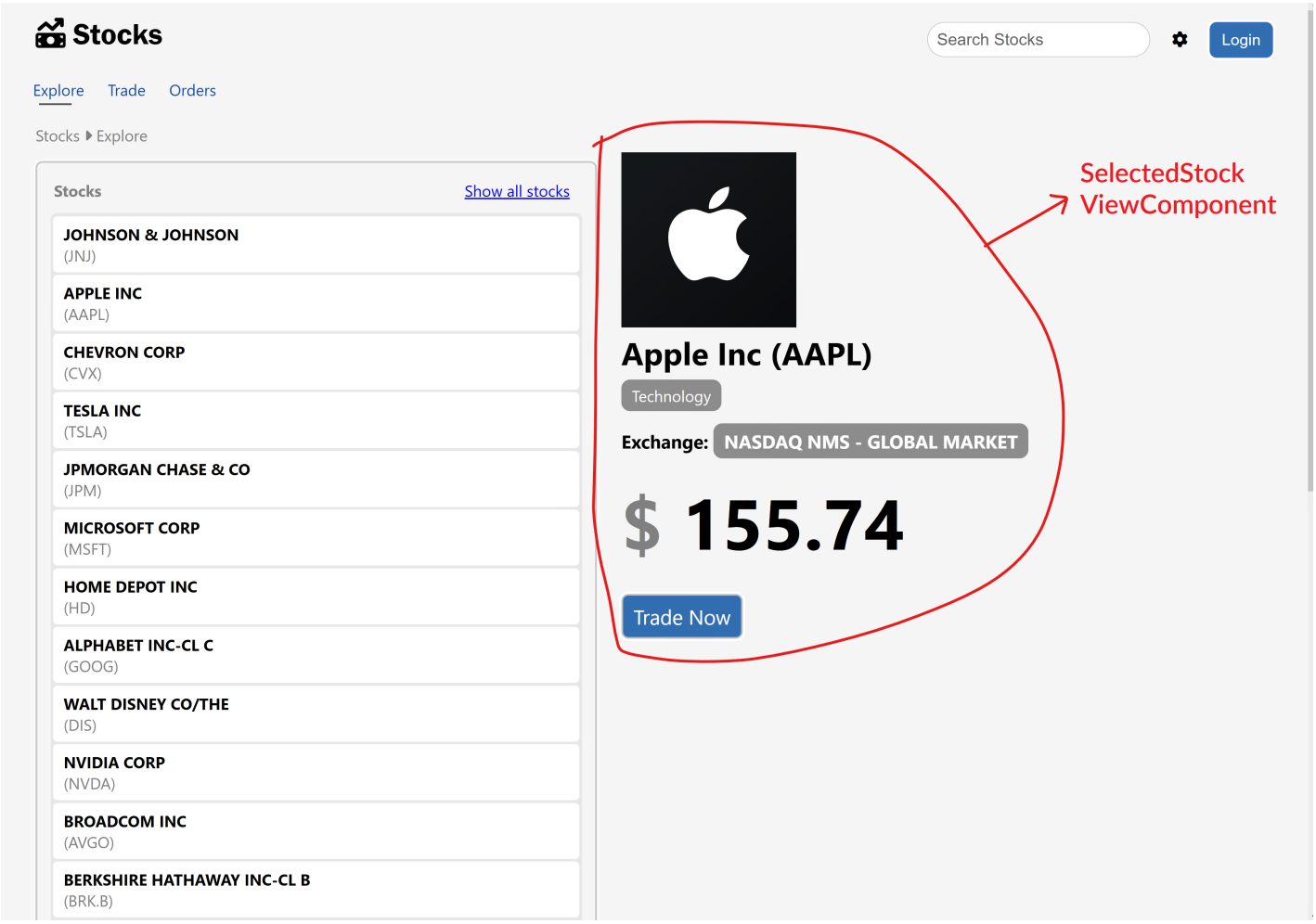
Trade/OrdersPDF:



Stocks/Explore:

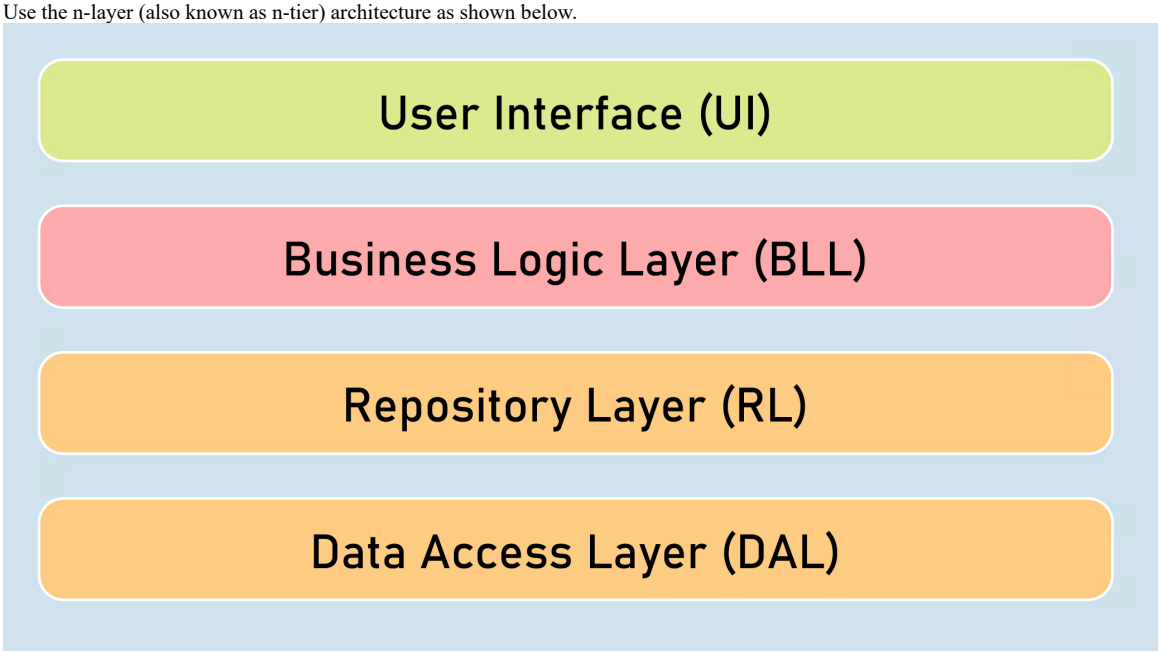


Stocks/Explore (after selecting a stock):



Architecture

Use the n-layer (also known as n-tier) architecture as shown below.



Finnhub.io:

https://finnhub.io is a service provider that provides live stock price information online.

User-Secrets:

- Register in "**https://finnhub.io/login**" to generate your own token [or] use the token "**cc676uaad3i9rj8tb1s0**" to make requests.
- After registration at finnhub, you can find your API Key (token) at "https://finnhub.io/dashboard".
- You need to store this token in user-secrets on your machine.
- You need to attach the token as a part of request url while making requests to "finnhub.io".

appsettings:

You will store the following configuration in appsettings.json:

```
"TradingOptions":
{
  "DefaultOrderQuantity": 100,
  "Top25PopularStocks":
     "AAPL,MSFT,AMZN,TSLA,GOOGL,GOOG,NVDA,BRK.B,META,UNH,JNJ,JPM,V,PG,XOM,HD,CVX,MA,BAC,ABBV,PFE,AV
},
"ConnectionStrings":
{
  "DefaultConnection": "Data Source=(localdb)\\MSSQLLocalDB;Initial
     Catalog=YourDatabaseHere;Integrated Security=True;Connect
     Timeout=30;Encrypt=False;TrustServerCertificate=False;ApplicationIntent=ReadWrite;MultiSubnetF
}
```

- The "DefaultOrderQuantity" specifies default quantity to place an order in "Trade/Index" view.
- The "Top25PopularStocks" represents the list of popular stocks whose prices to be shown in "Trade/Explore" view by default.
- The "DefaultConnection" represents database connection details.

Entity model:

You need to create two entity model classes named "BuyOrder" and "SellOrder".

1. Consider an entity model class called '**BuyOrder**' with following properties, along with corresponding validations:

- Guid BuyOrderID
- string StockSymbol [Mandatory]
- string StockName [Mandatory]
- DateTime DateAndTimeOfOrder
- uint Quantity [Value should be between 1 and 100000]
- double Price [Value should be between 1 and 10000]

2. Consider another entity model class called '**SellOrder**' with following properties, along with corresponding validations:

- Guid SellOrderID
- string StockSymbol [Mandatory]
- string StockName [Mandatory]
- DateTime DateAndTimeOfOrder
- uint Quantity [Range(1, 100000)]
- double Price [Range(1, 10000)]

These two entity classes are used to create list objects to store records in a service class.

DbContext:

You need to create a DbContext called "**StockMarketDbContext**" that connects to the database using the connection string mentioned in the appsettings.json.

It contains the following DbSets'.

- DbSet<BuyOrder> BuyOrders
- DbSet<SellOrder> SellOrders

DTO:

We are implementing the functionality of placing (creating) buy order and sell order.

So you need to create four DTO model classes named "**BuyOrderRequest**", "**BuyOrderResponse**", "**SellOrderRequest**", "**SellOrderResponse**".

As you can guess, the Request model classes (such as BuyOrderRequest and SellOrderRequest) are used as parameters in the service methods to receive information from controller to service class. Alternatively, the Response model classes (such as BuyOrderResponse and SellOrderResponse) are used as return type to return information back to controller, from the service.

1. Consider a DTO model class called '**BuyOrderRequest**' with following properties, along with corresponding validations:

- string StockSymbol [Mandatory]
- string StockName [Mandatory]
- DateTime DateAndTimeOfOrder [Should not be older than Jan 01, 2000]
- uint Quantity [Value should be between 1 and 100000]
- double Price [Value should be between 1 and 10000]

2. Consider another DTO model class called '**SellOrderRequest**' with following properties, along with corresponding validations:

- string StockSymbol [Mandatory]
- string StockName [Mandatory]
- DateTime DateAndTimeOfOrder [Should not be older than Jan 01, 2000]
- uint Quantity [Value should be between 1 and 100000]
- double Price [Value should be between 1 and 10000]

3. Consider another DTO model class called '**BuyOrderResponse**' with following properties:

- Guid BuyOrderID
- string StockSymbol
- string StockName
- DateTime DateAndTimeOfOrder
- uint Quantity
- double Price
- double TradeAmount

4. Consider another DTO model class called '**SellOrderResponse**' with following properties:

- Guid SellOrderID
- string StockSymbol
- string StockName
- DateTime DateAndTimeOfOrder
- uint Quantity
- double Price
- double TradeAmount

FinnhubRepository:

Create a Repository interface called '**IFinnhubRepository**' with following methods:

Task<Dictionary<string, object>?> GetCompanyProfile(string stockSymbol);

Task<Dictionary<string, object>?> GetStockPriceQuote(string stockSymbol);

Task<List<Dictionary<string, string>>?> GetStocks();

Task<Dictionary<string, object>?> SearchStocks(string stockSymbolToSearch);

Implement the above Repository interface called '**IFinnhubRepository**' that sends request to the respective url and returns its response as a Dictionary<string, object>.

GetCompanyProfile: https://finnhub.io/api/v1/stock/profile2?symbol={symbol}&token={token}

GetStockPriceQuote: https://finnhub.io/api/v1/quote?symbol={symbol}&token={token}

GetStocks: https://finnhub.io/api/v1/stock/symbol?exchange=US&token={token}

SearchStocks: https://finnhub.io/api/v1/search?q={stockNameToSearch}&token={token}

The all the repository methods such as GetCompanyProfile(), GetStockPriceQuote(), GetStocks(), SearchStocks() return the response data that was actually returned by finnhub.io/api.

IFinnhubRepository.GetStockPriceQuote():

The returned data from IFinnhubRepository.GetStockPriceQuote() looks like this:

{"c":235.87,"d":9.12,"dp":4.0221,"h":236.6,"l":226.06,"o":226.24,"pc":226.75,"t":1666987204}

Response Attributes:

c

Current price

d

Change

dp

Percent change

h

High price of the day

l

Low price of the day

o

Open price of the day

pc

Previous close price

Reference: https://finnhub.io/docs/api/quote

IFinnhubRepository.GetCompanyProfile():

The returned data from IFinnhubRepository.GetCompanyProfile() looks like this:

{"country":"US","currency":"USD","exchange":"NASDAQ NMS - GLOBAL MARKET","finnhubIndustry":"Technology","ipo":"1986-03-13","logo":"https://static2.finnhub.io/file/publicdatany/finnhubimage/stock_logo/MSFT.svg","marketCapitalization":1758286.5806001066,"name":"Microsoft Corp","phone":"14258828080.0","shareOutstanding":7454.47,"ticker":"MSFT","weburl":"https://www.microsoft.com/en-us"}

Response Attributes:

country

Country of company's headquarter.

currency

Currency used in company filings.

exchange

Listed exchange.

finnhubIndustry

Finnhub industry classification.

ipo

IPO date.

logo

Logo image.

marketCapitalization

Market Capitalization.

name

Company name.

phone

Company phone number.

shareOutstanding

Number of oustanding shares.

ticker

Company symbol/ticker as used on the listed exchange.

weburl

Company website.

Reference: https://finnhub.io/docs/api/company-profile2

IFinnhubRepository.GetStocks():

The returned data from IFinnhubRepository.GetStocks() looks like this:

[{"currency":"USD","description":"NORTECH SYSTEMS INC","displaySymbol":"NSYS","figi":"BBG000BFV339","isin":null,"mic":"XNAS","shareClassFIGI":"BBG001S6XGN6","symbol":"NSYS","symbol2":"","type":"(Stock"}, ...]

Response Attributes:

currency

Price's currency. This might be different from the reporting currency of fundamental data.

description

Symbol description

displaySymbol

Display symbol name.

figi

FIGI identifier.

isin

ISIN. This field is only available for EU stocks and selected Asian markets. Entitlement from Finnhub is required to access this field.

mic

Primary exchange's MIC.

shareClassFIGI

Global Share Class FIGI.

symbol

Unique symbol used to identify this symbol used in /stock/candle endpoint.

symbol2

Alternative ticker for exchanges with multiple tickers for 1 stock such as BSE.

type

Security type.

Reference: https://finnhub.io/docs/api/stock-symbols

IFinnhubRepository.SearchStocks():

The returned data from IFinnhubRepository.SearchStocks() looks like this:

{"count":32,"result":[{"description":"APPLE INC","displaySymbol":"AAPL","symbol":"AAPL","type":"Common Stock"}, ...] }

Response Attributes:

count

Number of results.

result

Array of search results.

description

Symbol description

displaySymbol

Display symbol name.

symbol

Unique symbol used to identify this symbol used in /stock/candle endpoint.

type

Security type.

Reference: https://finnhub.io/docs/api/symbol-search

StocksRepository:

Create a repository interface called '**IStocksRepository**' with following methods:

Task<BuyOrder> CreateBuyOrder(BuyOrder buyOrder);

Task<SellOrder> CreateSellOrder(SellOrder sellOrder);

Task<List<BuyOrder>> GetBuyOrders();

Task<List<SellOrder>> GetSellOrders();

Implement the above repository interface called 'IStocksRepository' that performs the specified operation.

CreateBuyOrder: Inserts a new buy order into the database table called 'BuyOrders'.

CreateSellOrder: Inserts a new sell order into the database table called 'SellOrders'.

GetBuyOrders: Returns the existing list of buy orders retrieved from database table called 'BuyOrders'.

GetSellOrders: Returns the existing list of sell orders retrieved from database table called 'SellOrders'.

FinnhubCompanyProfileService:

Create a service interface called '**IFinnhubCompanyProfileService**' with following methods:

Dictionary<string, object>? GetCompanyProfile(string stockSymbol);

Implement the above service interface called 'IFinnhubCompanyProfileService'; and each method invokes corresponding repository method as mentioned below.

GetCompanyProfile: Invokes FinnhubRepository.GetCompanyProfile() and returns the same returned value.

The service methods return the same response data that was actually returned by the corresponding repository method. Throw a custom exception called FinnhubException in all methods in this service, when an exception occurs while connecting to finnhub/api server.

FinnhubSearchStocksService:

Create a service interface called '**IFinnhubSearchStocksService**' with following methods:

Task<Dictionary<string, object>?> SearchStocks(string stockSymbolToSearch);

Implement the above service interface called '**IFinnhubSearchStocksService**'; and each method invokes corresponding repository method as mentioned below.

SearchStocks: Invokes FinnhubRepository.SearchStocks() and returns the same returned value.

Note: The service methods return the same response data that was actually returned by the corresponding repository method. Throw a custom exception called FinnhubException in all methods in this service, when an exception occurs while connecting to finnhub/api server.

FinnhubStockPriceQuoteService:

Create a service interface called '**IFinnhubStockPriceQuoteService**' with following methods:

Dictionary<string, object>? GetStockPriceQuote(string stockSymbol);

Implement the above service interface called 'IFinnhubStockPriceQuoteService'; and each method invokes corresponding repository method as mentioned below.

GetStockPriceQuote: Invokes FinnhubRepository.GetStockPriceQuote() and returns the same returned value.

The service methods return the same response data that was actually returned by the corresponding repository method. Throw a custom exception called FinnhubException in all methods in this service, when an exception occurs while connecting to finnhub/api server.

FinnhubStocksService:

Create a service interface called 'IFinnhubStocksService' with following methods:

Task<List<Dictionary<string, string>>?> GetStocks();

Implement the above service interface called 'IFinnhubStocksService'; and each method invokes corresponding repository method as mentioned below.

GetStocks: Invokes FinnhubRepository.GetStocks() and returns the same returned value.

The service methods return the same response data that was actually returned by the corresponding repository method. Throw a custom exception called FinnhubException in all methods in this service, when an exception occurs while connecting to finnhub/api server.

StocksService:

Create a service interface called '**IStocksService**' with following methods:

Task<BuyOrder> CreateBuyOrder(BuyOrder buyOrder);

Task<SellOrdere> CreateSellOrder(SellOrder sellOrder);

Task<List<BuyOrder>> GetBuyOrders();

Task<List<SellOrder>> GetSellOrders();

Implement the above service interface called 'IStocksService' that performs the specified operation.

CreateBuyOrder: Invokes the StocksRepository.CreateBuyOrder() and returns the same returned value.

CreateSellOrder: Invokes the StocksRepository.CreateSellOrder() and returns the same returned value.

GetBuyOrders: Invokes the StocksRepository.GetBuyOrders() and returns the same returned value.

GetSellOrders: Invokes the StocksRepository.GetSellOrders() and returns the same returned value.

StockTrade:

You need to create three view model classes called "**StockTrade**", "**Stock**" and "**Orders**".

Create a view model class called "StockTrade" in the Asp.Net Core project with following properties:

string? StockSymbol

string? StockName

double Price

unit Quantity

This class will be used to send model object from controller to the "Trade/Index" view.

Stock:

Create a view model class called "Stocke" in the Asp.Net Core project with following properties:

string? StockSymbol

string? StockName

This class will be used to send model object from controller to the "Stocks/Explore" view.

Orders:

Create a view model class called "Orders" in the Asp.Net Core project with following properties:

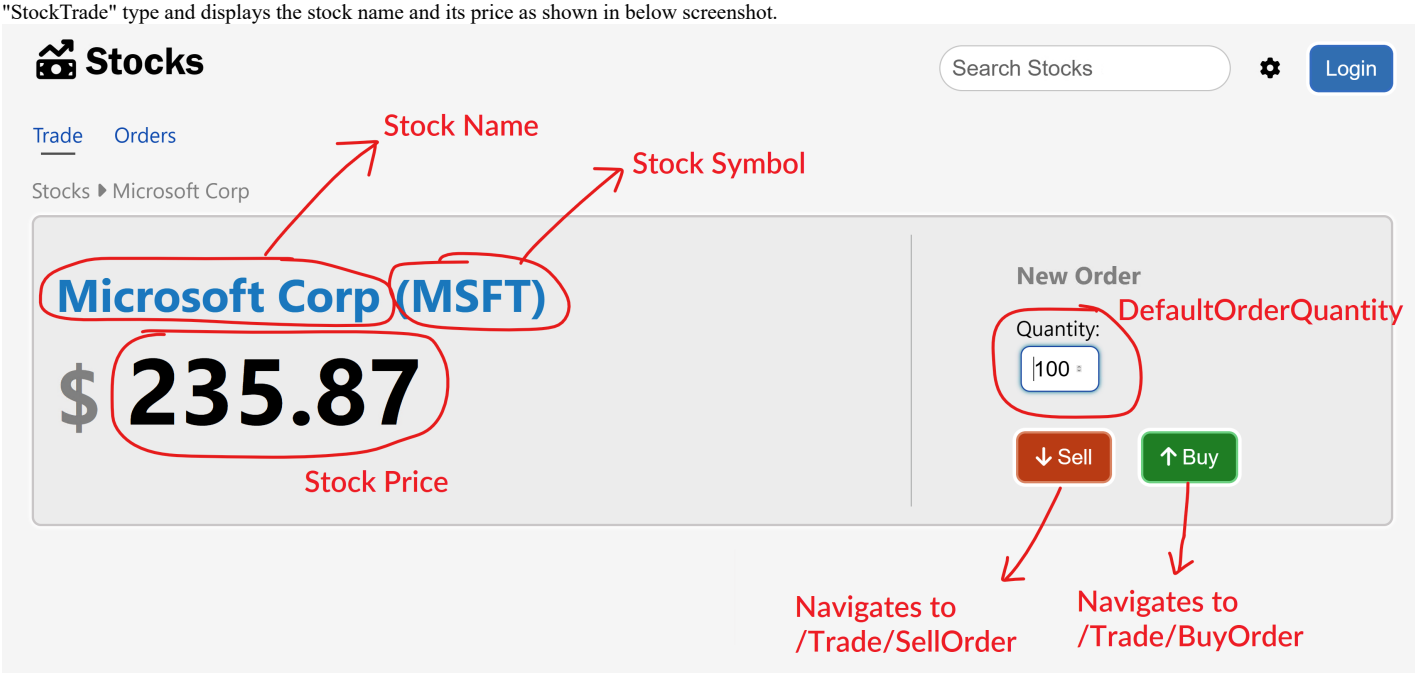
- List<BuyOrderResponse> BuyOrders
- List<SellOrderResponse> SellOrders

This class will be used to send list of buy orders and sell orders - from controller to the "Trade/Orders" view.

"Trade\Index.cshtml":

You need to create four views called "Trade\Index.cshtml", "Stocks\Explore.cshtml", "Trade\Orders.cshtml" and "Trade\OrdersPDF.cshtml".

Create a view called "Index.cshtml" in "Views\Trade" folder - that is strongly typed to the view model class called "StockTrade". So it receives the model object of "StockTrade" type and displays the stock name and its price as shown in below screenshot.



The view should store the stock symbol (Eg: MSFT) as a hidden element. So that, it can be accessible in the javascript code, while connecting finnhub websocket.

So, you need to write javascript code to connect to finnhub websocket at `**wss://ws.finnhub.io?token=${token}**` url. On receiving each message from the socket, you need update the latest stock price in the UI.

The response from finnhub websocket looks like this:

{"data":[{"p":220.89,"s":"MSFT","t":1575526691134,"v":100}, {"p":220.82,"s":"MSFT","t":1575526691167,"v":15}],"type":"trade"}

Response attributes:

type: message type

data: [list of trades]

s: symbol of the company

p: Last price

t: UNIX milliseconds timestamp

v: volume (number of orders)

c: trade conditions (if any)

You can see more than one element in the 'data' array in the above response example. But in the output, we need to display only one. You can display either of the prices (or) the highest price in the output.

When the page is closed, you must unsubscribe from the finnhub websocket to avoid any memory leaks.

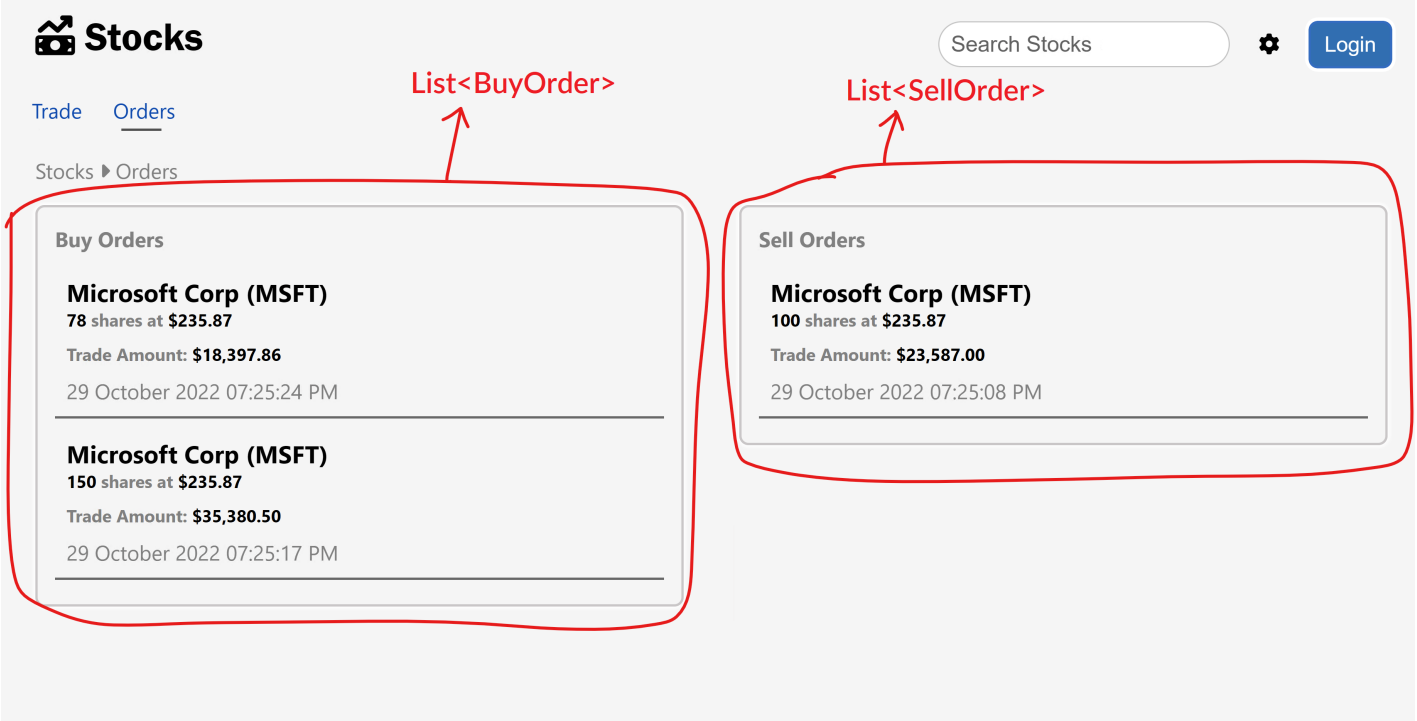
Overall, your application should refresh the updated price of the stock for every one or two seconds as long as the page runs (of course, only when the market is LIVE i.e. usually 09:30 a.m. to 4 p.m. (ET)).

When the user clicks on the "Buy" button, the browser should make a HTTP POST request to "Trade/BuyOrder" with values of "StockSymbol", "StockName", "Quantity" and "Price". You can submit values that are not to be directly displayed, through input type="hidden" fields also.

When the user clicks on the "Sell" button, the browser should make a HTTP POST request to "Trade/SellOrder" with values of "StockSymbol", "StockName", "Quantity" and "Price". You can submit values that are not to be directly displayed, through input type="hidden" fields also.

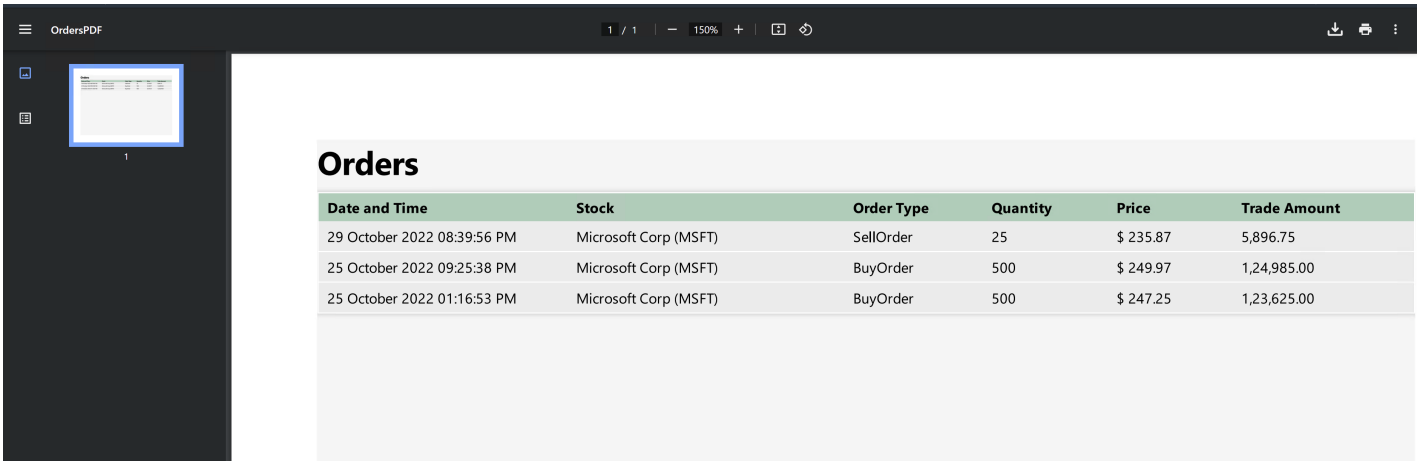
"Trade\Orders.cshtml":

Create a view called "Orders.cshtml" in "Views\Trade" folder - that is strongly typed to the view model class called "Orders". So it receives the model object of "Orders" type from controller and displays the list of both buy orders and sell orders as shown in the below screenshot.



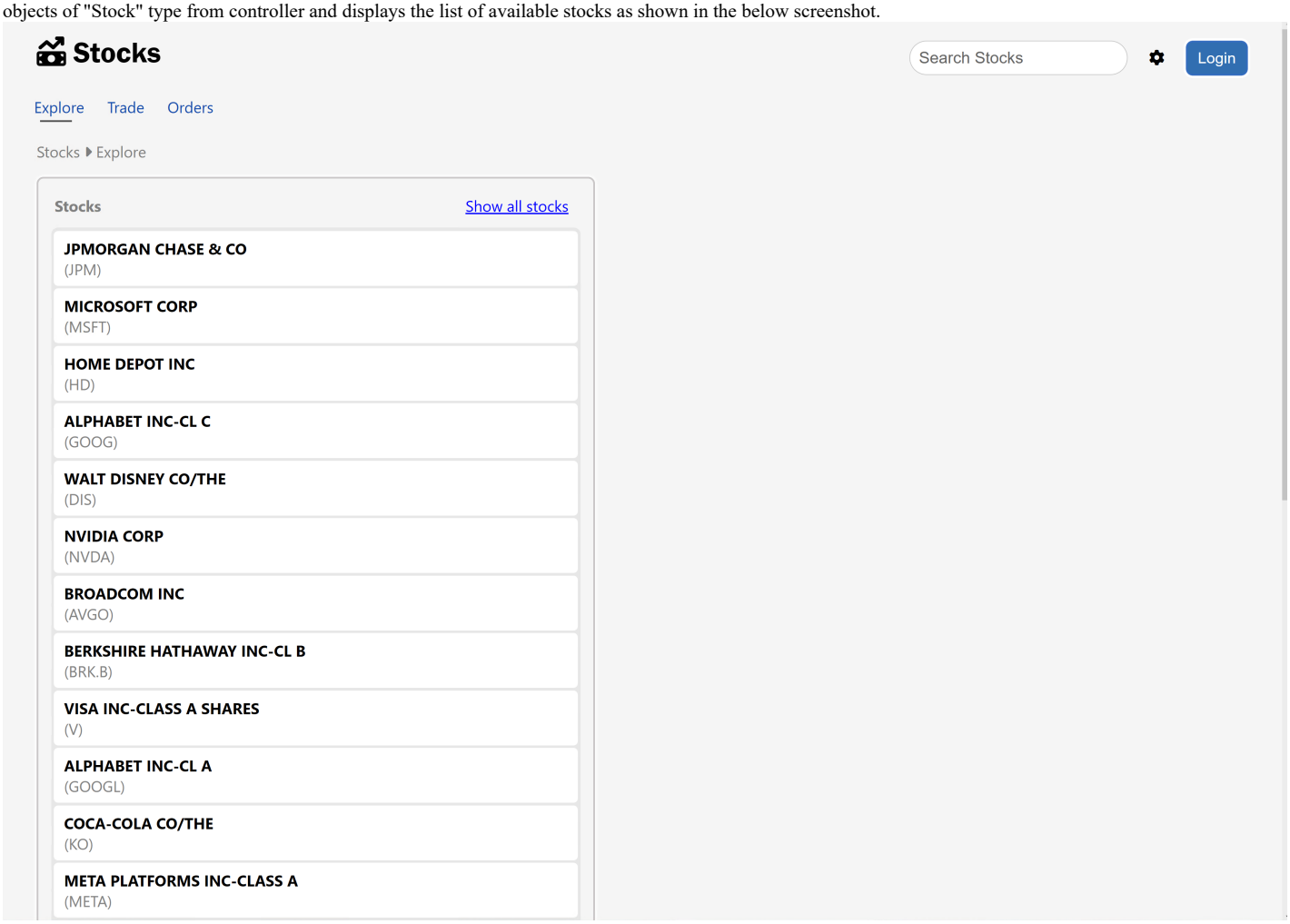
"Trade\OrdersPDF.cshtml":

Create a view called "OrdersPDF.cshtml" in "Views\Trade" folder - that is strongly typed view bound to list of orders. So it receives list of both BuyOrder and SellOrder from controller & displays the list or orders in printable format as shown in the below screenshot.

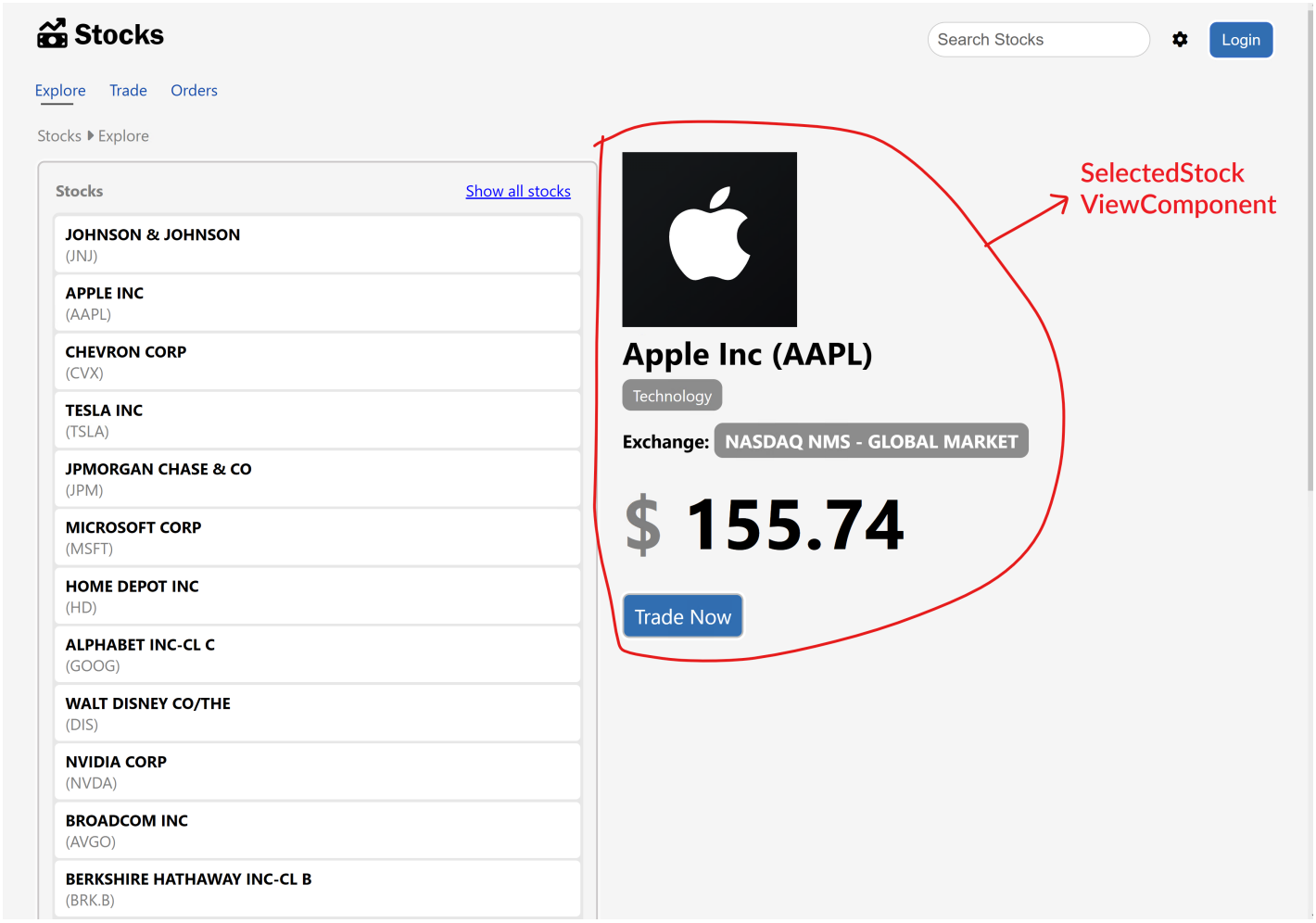


"Stocks\Explore.cshtml":

Create a view called "Explore.cshtml" in "Views\Stocks" folder - that is strongly typed to the view model class of type "List<Stock>". So it receives a list of model objects of "Stock" type from controller and displays the list of available stocks as shown in the below screenshot.



When the user clicks on a specific stock, it should invoke a ViewComponent called "SelectedStockViewComponent" and that should display the selected stock details such as stock name, stock symbol, stock image, price etc., along with same list of availble stocks as mentioned in the below screenshot.



Trade Controller:

Create a controller called TradeController that has five action methods called "**Index**()", "**BuyOrder**()" and "**SellOrder**()", "**Orders**()" and "OrdersPDF()".

The controller has to inject the appsettings called "TradingOptions" (from appsettings.json) and essential services.

TradeController.Index():

- It receives HTTP GET request at route "Trade/Index".

- It first gets the "TradingOptions" from appsettings.json using Options pattern.

- The Index() action method invokes the following methods:

1. FinnhubCompanyProfileService.GetCompanyProfile() to fetch stock name, stock symbol and other details.
2. FinnhubStockPriceQuoteService.GetStockPriceQuote() to fetch current stock price.

- And then it creates an object of "StockTrade" model class and fills essential data such as StockSymbol, StockName, Price and Quantity that are read from the return values of above mentioned service methods i.e. "FinnhubService.GetCompanyProfile()" and "FinnhubService.GetStockPriceQuote()".

- Then it sends the same model object to the view.

TradeController.BuyOrder():

- It receives HTTP POST request at route "Trade/BuyOrder".
- It receives the model object of "BuyOrder" type through model binding.
- It initializes "DateAndTimeOfOrder" into the model object (i.e. buyOrder).

- If model state has no errors, it invokes StocksService.CreateBuyOrder() method. Then it redirects to "Trade/Orders" route to display list of orders.
- Alternatively, in case of validation errors in the model object, it reinvokes the same view, along with same model object.

TradeController.SellOrder():

- It receives HTTP POST request at route "Trade/SellOrder".
- It receives the model object of "SellOrder" type through model binding.
- It initializes "DateAndTimeOfOrder" into the model object (i.e. sellOrder).
- If model state has no errors, it invokes StocksService.CreateSellOrder() method. Then it redirects to "Trade/Orders" route to display list of orders.
- Alternatively, in case of validation errors in the model object, it reinvokes the same view, along with same model object.

TradeController.Orders():

- It receives HTTP GET request at route "Trade/Orders".
- It invokes both the service methods StocksService.GetBuyOrders() and StocksService.GetSellOrders().
- Then it creates an object of the view model class called 'Orders' and initializes both 'BuyOrders' and 'SellOrders' properties with the data returned by the above called service methods.
- It invokes the "Trade/Orders" view to display list of orders.

TradeController.OrdersPDF():

- It receives HTTP GET request at route "Trade/OrdersPDF".
- It invokes both the service methods StocksService.GetBuyOrders() and StocksService.GetSellOrders().
- It invokes the "Trade/OrdersPDF" view and renders it as PDF file using 'Rotativa' package.

Stocks Controller:

Create a controller called StocksController that has an action method called "Explore()".

The controller has to inject the appsettings called "TradingOptions" (from appsettings.json) and essential services.

StocksController.Explore():

- It receives HTTP GET request at route "Stocks/Explore".
- The Explore() action method invokes the FinnhubService.GetStocks() to fetch list of all available stocks in the market.
- And then it filters out unnecessary stocks from the list but keeps only the stocks in "Top25PopularStocks".
- It sends the filtered stocks (only top 25 stocks) to the view as List<Stock>.
- It has a route parameter called "stock". It sends the same stock (stock symbol) to the view through ViewBag. So that, the view can invoke a view component called "SelectedStock" to display selected stock details at right hand side as shown in the screenshot of "Stocks/Explore" view.

CreateOrderActionFilter:

Create an asynchronous action filter called "CreateOrderActionFilter" and apply the same to both "TradeController.BuyOrder()" and "TradeController.SellOrder()" action methods.

Before logic:

- First checks if it has been applied TradeController.

- If so, then it receives the action argument called "orderRequest".
- It perform model level validations on that model object.
- In case of no model errors, it doesn't nothing - just invokes the subsequent filter in the filter pipeline.
- In case if it is has one or more model errors, it has to create object of StockTrade model class with essential data, adds model errors to ViewBag.Errors and then reinvokes the "TradeController.Index" view

ExceptionHandlingMiddleware:

- Create a custom middleware called 'ExceptionHandlingMiddleware' and apply it in earliest position in the middleware pipiline.
- It handles any exception that occur in all subsequent middleware, action method and filters.
- When an exception is caught, it logs that same exception details and rethrow it - so that, the same exception can be caught by UseExceptionHandler() middleware added earlier in the middleware pipeline.
- The built-in UseExceptionHandler() middleware shows user-friendly error message using "/Home/Error" view.

xUnit Test cases:

Write the following unit test cases for testing respective service methods. Use Fluent assertions, Auto fixture, Moq as per the necessity. You need to mock the repositories.

StocksService.CreateBuyOrder():

1. When you supply BuyOrderRequest as null, it should throw ArgumentNullException.

2. When you supply buyOrderQuantity as 0 (as per the specification, minimum is 1), it should throw ArgumentException.

3. When you supply buyOrderQuantity as 100001 (as per the specification, maximum is 100000), it should throw ArgumentException.

4. When you supply buyOrderPrice as 0 (as per the specification, minimum is 1), it should throw ArgumentException.

5. When you supply buyOrderPrice as 10001 (as per the specification, maximum is 10000), it should throw ArgumentException.

6. When you supply stock symbol=null (as per the specification, stock symbol can't be null), it should throw ArgumentException.

7. When you supply dateAndTimeOfOrder as "1999-12-31" (YYYY-MM-DD) - (as per the specification, it should be equal or newer date than 2000-01-01), it should throw ArgumentException.

8. If you supply all valid values, it should be successful and return an object of BuyOrderResponse type with auto-generated BuyOrderID (guid).

StocksService.CreateSellOrder():

1. When you supply SellOrderRequest as null, it should throw ArgumentNullException.

2. When you supply sellOrderQuantity as 0 (as per the specification, minimum is 1), it should throw ArgumentException.

3. When you supply sellOrderQuantity as 100001 (as per the specification, maximum is 100000), it should throw ArgumentException.

4. When you supply sellOrderPrice as 0 (as per the specification, minimum is 1), it should throw ArgumentException.

5. When you supply sellOrderPrice as 10001 (as per the specification, maximum is 10000), it should throw ArgumentException.

6. When you supply stock symbol=null (as per the specification, stock symbol can't be null), it should throw ArgumentException.

7. When you supply dateAndTimeOfOrder as "1999-12-31" (YYYY-MM-DD) - (as per the specification, it should be equal or newer date than 2000-01-01), it should throw ArgumentException.

8. If you supply all valid values, it should be successful and return an object of SellOrderResponse type with auto-generated SellOrderID (guid).

StocksService.GetAllBuyOrders():

1. When you invoke this method, by default, the returned list should be empty.

2. When you first add few buy orders using CreateBuyOrder() method; and then invoke GetAllBuyOrders() method; the returned list should contain all the same buy orders.

StocksService.GetAllSellOrders():

1. When you invoke this method, by default, the returned list should be empty.

2. When you first add few sell orders using CreateSellOrder() method; and then invoke GetAllSellOrders() method; the returned list should contain all the same sell orders.

xUnit Controller Test cases:

Write the following unit test cases for testing respective controller action methods. You need to mock the services.

StocksController.Explore():

1. When you supply null to "stock" parameter, it should return "Explore" view along with List<Stock> as model object.

xUnit Integration Test cases:

Write the following integration test cases for testing respective routes.

"/Trade/Index/MSFT":

When you make a HTTP GET request to the "Trade/Index/MSFT" route, it should return a view result (html response) with an HTML element that has class="price".

Instructions:

1. Create controller(s) with attribute routing.
2. Provide the configuration as service, using Options pattern.
3. Inject the IOptions in the controller.
4. Use CSS styles, layout views, _ViewImports, _ViewStart as per the necessity.
5. The CSS file is provided as downlodable resource for applying essential styles. You can download and use it.
6. Inject essential services such as IFinnhubCompanyProfileService and other services in Controller. Invoke IStocksRepository and IFinnhubRepository in respective service classes.
7. Invoke essential service methods in controller.
8. The Entity model class (BuyOrder and SellOrder) should not be accessed in the controller. They must be used only in the service classes.
9. The DTO model classes (BuyOrderRequest, BuyOrderResponse, SellOrderRequest, SellOrderResponse) should be used as parameter type or return type in the service methods; and can be used in both services and controller.
10. Use appropriate tag helpers such as "asp-controller", "asp-action", "asp-for" etc., in all views wherever necessary.
11. Write appropriate logs using ILogger, in controllers and services.

12. In case of non-Development environment, apply both custom exception handling middleware (i.e. ExceptionHandlingMiddleware) and also built-in error handling middleware called "UseExceptionHandler".
13. Apply SOLID principles such as 'Interface Segregation Principle', 'Single Responsibility Principle', 'Dependency Inversion Principle' and other principles while creating services and other classes.

Changes from previous assignment:

Apply SOLID principles such as 'Interface Segregation Principle', 'Single Responsibility Principle', 'Dependency Inversion Principle' and other principles in the existing code base.