AUTOMATED TRADING SETUP TO MITIGATE MANUAL TRADING

Ashutosh

April 2023

Abstract

Manual trading can be time-consuming and require a significant amount of effort and attention to detail. Traders need to keep up to date with market news, economic indicators, and other relevant information to make informed trading decisions. They also need to monitor their trades in real-time and adjust their strategies as market conditions change. The world is moving to a fast paced automated trading concepts. It is estimated that millions of people around the world engage in trading as a hobby or part-time activity and rely on chart logics for most of the trading. The project is an effort to bridge a gap between a non-professional trader who still sit and trade for hours and the world of automation by creating a framework where traders choose define their logic on a UI and the machine follow the trading principles and trade for them mitigating continuous monitoring for signals.

1. OBJECTIVE

The project is based on algorithmic paper trading using python and its pre-available wrappers/modules.
The entire project is divided into two segments:
1) Trading Logic
2) Trading Bot

1) TRADING LOGIC
We have used an algorithmic designing/trading platform called “QuantConnect” which provides a framework to access all market data for all time frontiers along with ease of API connection with broker apps for paper trading. We are not using real money at this point taking in view that the trading strategy is naïve and very basic.

2) TRADING BOT
We have created a framework where a user can define its own strategy using a UI setup provided by TradeView platform. Alerts are setup on the platform to generate Buy/Sell signals which are then connected to a code hosted on AWS to trade and take positions on Alpaca trading environment.

An attempt has been made to understand the concepts of the model by carrying out the following:

1. Understanding the QuantConnect platform and framework.
2. Understanding and coding different trading logics in python.
3. Learning about day trading manual intensive issues.
4. Getting familiar with automated trading concepts.

2. QUANT CONNECT

QuantConnect is a cloud-based platform that allows users to develop, test, and deploy algorithmic trading strategies using a variety of programming languages, including C#, Python, and F#. The platform provides access to a vast amount of financial data, including historical price data, fundamental data, and alternative data sets such as news and social media sentiment.

QuantConnect's algorithmic trading engine allows users to backtest their trading strategies using historical data, which can help to identify potential trading opportunities and refine the strategy's parameters. The platform also provides a range of backtesting and optimization tools, including a portfolio optimizer, which helps users to optimize their portfolio based on specific risk and return objectives.

Once a strategy has been backtested and optimized, users can deploy their algorithmic trading strategies in live trading environments. QuantConnect offers a range of execution services, including order routing and execution algorithms, to ensure that users' trades are executed quickly and efficiently.

One of the unique features of QuantConnect is its community of users, which includes professional traders, institutional investors, and hobbyist traders. The platform provides a forum where users can share trading ideas and collaborate on the development of trading strategies. The community also provides a repository of open-source trading algorithms that users can use as a starting point for their own strategies.

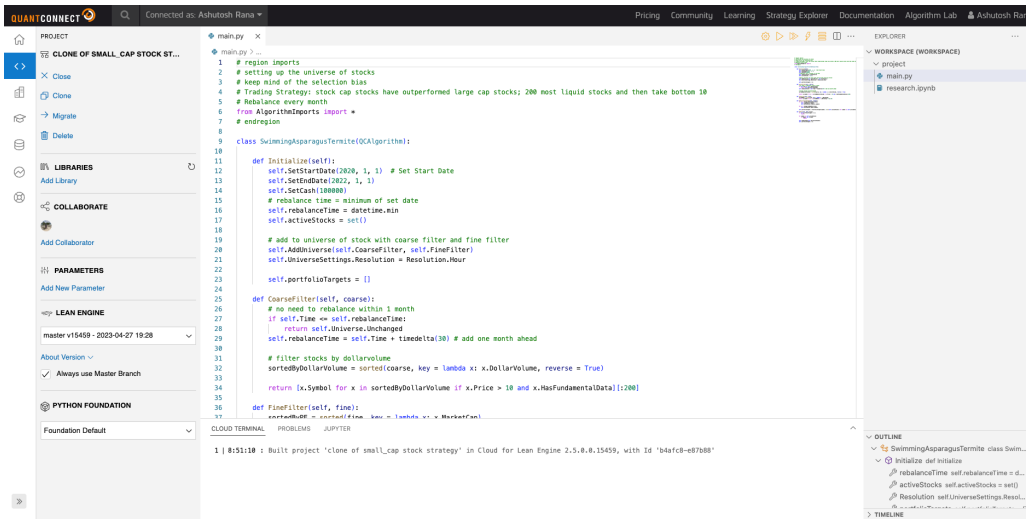


Figure 1: https://www.quantconnect.com/

3. TRADING VIEW

TradingView is a web-based platform that provides real-time data and customizable charts for a wide range of financial instruments. The platform allows users to chart and analyze stocks, futures, forex, cryptocurrencies, and more.

One of the key features of TradingView is its charting capabilities. The platform offers a wide range of chart types, including line charts, bar charts, and candlestick charts. Users can customize their charts by adding technical indicators, drawing tools, and other features. TradingView offers a comprehensive library of over 100 technical indicators, as well as the ability to create custom indicators using its proprietary scripting language, Pine Script.

TradingView also offers a social community where users can share trading ideas, collaborate with other traders, and discuss market trends. Users can follow other traders, view their trading strategies and ideas, and even copy their trades through the platform's auto-trading feature. The platform also includes a chat feature that allows users to communicate in real-time with other traders.

In addition to charting and social features, TradingView offers a range of tools and features designed to help traders make informed trading decisions. These include a watchlist, alerts, and a paper trading mode. The watchlist allows users to track multiple instruments simultaneously and monitor key price levels. Alerts can be set to notify users when price targets are reached or when certain conditions are met, and the paper trading mode allows users to practice trading strategies without risking real money.

TradingView offers both free and paid subscription plans. The free plan includes access to basic charting and analysis tools, as well as access to the social community. The paid plans offer additional features, such as access to more data, advanced technical analysis tools, and custom indicator creation.

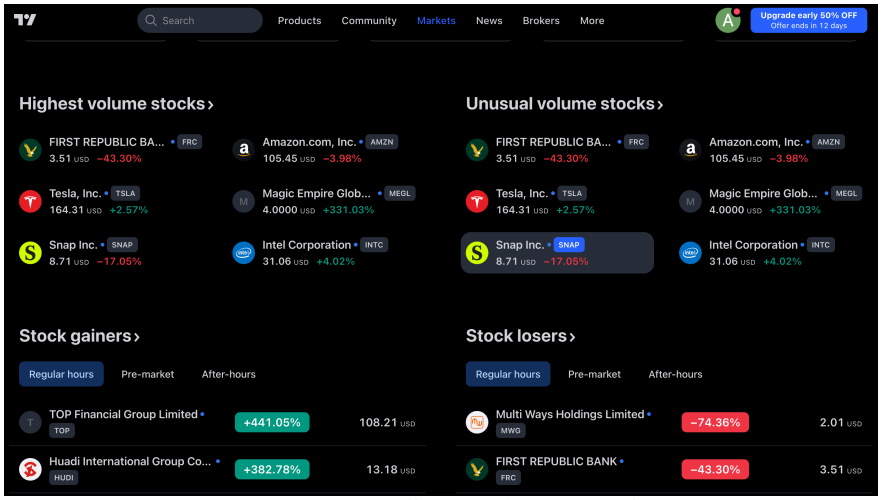


Figure 2: https://www.tradingview.com/

3.1 USER INTERFACE

Charting is an essential aspect of trading as it provides a visual representation of price movements in financial markets. Charting in trading involves using various chart types and technical analysis tools to identify patterns and trends in the data and make informed trading decisions.

There are several different types of charts used in trading, including line charts, bar charts, and candlestick charts. Line charts provide a simple way of displaying the price movements of an asset over time, while bar charts provide more detail by displaying the opening, closing, high, and low prices for each period. Candlestick charts provide a similar level of detail as bar charts but are more visually appealing and easier to interpret.

In addition to chart types, traders use various technical analysis tools to analyze market trends and make trading decisions. These tools include indicators, such as moving averages, relative strength index (RSI), and Bollinger Bands, as well as chart patterns, such as head and shoulders, triangles, and double tops/bottoms. Technical analysis tools help traders to identify potential entry and exit points in the market, as well as to set stop-loss and take-profit levels.

Charting is a vital part of trading as it provides traders with a visual representation of market trends and helps them to make informed trading decisions. By using charting tools and technical analysis, traders can identify potential opportunities in the market and develop effective trading strategies to capitalize on them.

TradingView platform gives users a wide range of charting options to define strategies and trade.

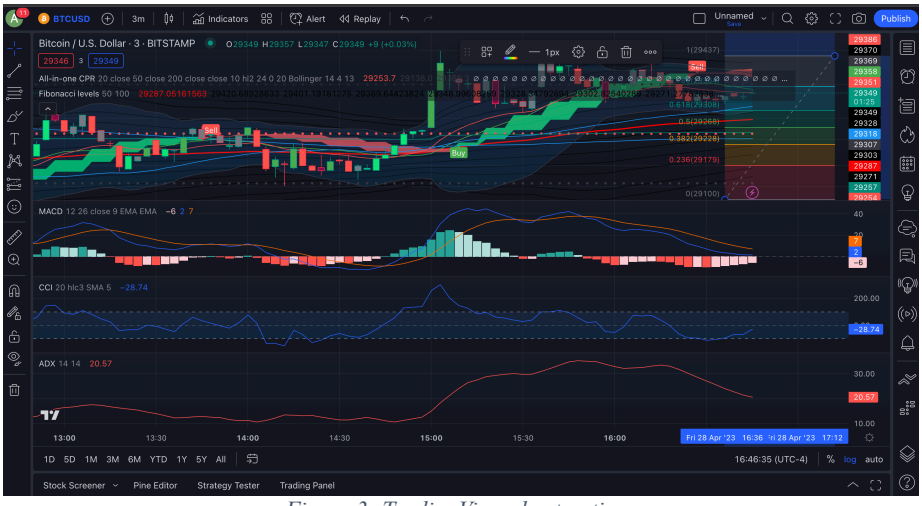


Figure 3: TradingView chart options

3.2 JSON

JSON, which stands for JavaScript Object Notation, is a lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate. It is often used for exchanging data between web servers and clients, as well as between different systems and programming languages.

JSON is based on a subset of the JavaScript programming language, and it uses a simple syntax to represent data in a key-value format. Data in JSON format consists of objects and arrays, which can be nested to represent more complex data structures.

Here's an example of a simple JSON object:

```
{
  "name": "Student1",
  "age": 23,
  "email": "student1@example.com"
}
```

3.2 WEBHOOKUP

Webhooks are a way for web applications to communicate with each other in real-time. They are essentially automated messages or notifications that are triggered by specific events or actions and sent from one application to another via HTTP requests.

When a webhook is set up, the sending application (the "source") will send a POST request to a URL endpoint on the receiving application (the "destination") when a certain event occurs. The data from the event is included in the POST request as JSON or XML data, which the receiving application can then process and use.

Webhooks are commonly used in a variety of scenarios, such as:

Integrating two different web applications, such as a CRM and a marketing automation platform.
Notifying a user or system of an event, such as a new order or a completed payment.
Automatically triggering an action or process, such as generating a new invoice or sending a follow-up email.

To set up a webhook, the receiving application typically provides a URL endpoint and any necessary authentication details to the sending application. The sending application will then make a POST request to that URL whenever the specified event occurs.

Once the webhook is set up, the receiving application can use the data from the POST request to trigger any necessary actions or processes, such as updating a database, sending a notification to a user, or generating a report.

3.4 GENERATION OF ALERTS

TradingView offers a variety of alert options for traders to receive notifications about certain market conditions or events. The alerts can be set up based on a variety of criteria, including price levels, technical indicators, and chart patterns.

To create an alert on TradingView, users can follow these steps:

1. Open the chart that they want to set an alert for.
2. Click on the "Alert" icon on the top toolbar of the chart window.
3. Select the type of alert they want to set up. For example, they can choose "Price," "Indicator," or "Drawing" alert.
4. Set the criteria for the alert. For instance, if they want to set up a price alert, they can choose the currency pair or security they want to monitor and set the price level at which they want to receive the alert.
5. Choose the notification method for the alert. TradingView supports multiple notification methods, including email, SMS, and in-platform notifications.
6. Save the alert.

Once the alert is set up, users will receive a notification when the criteria for the alert are met. They can also view their alerts in the "Alerts" tab on the bottom toolbar of the TradingView platform.

TradingView also offers advanced alert features, including the ability to set multiple conditions for an alert, add notes to an alert, and set up alerts for custom indicators created with Pine Script.

Overall, TradingView's alert system is a useful tool for traders looking to stay informed about market conditions and make timely trading decisions.

We have exploited the feature of Webhook URL feature of alerts to produce required JSON file.

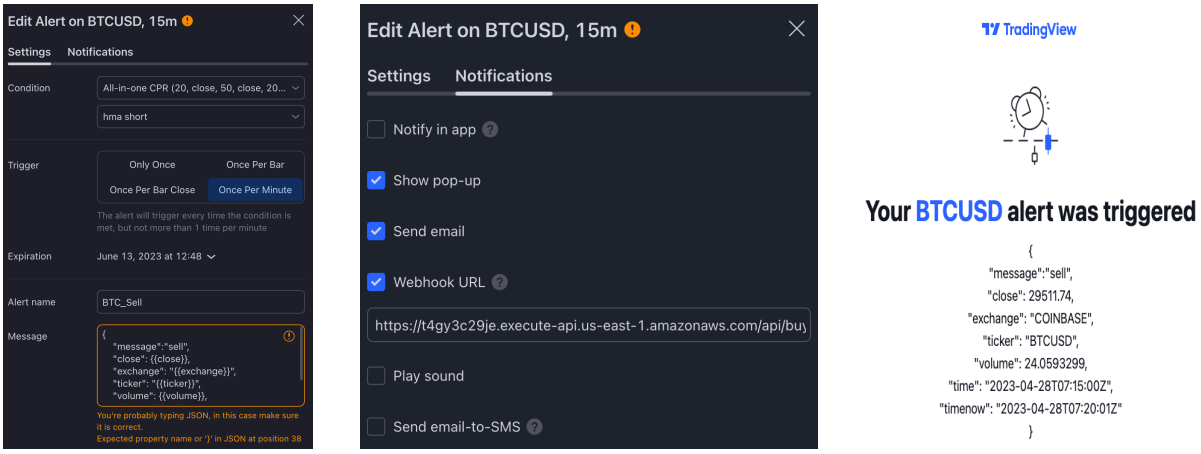


Figure 4: Sell Signal Setup

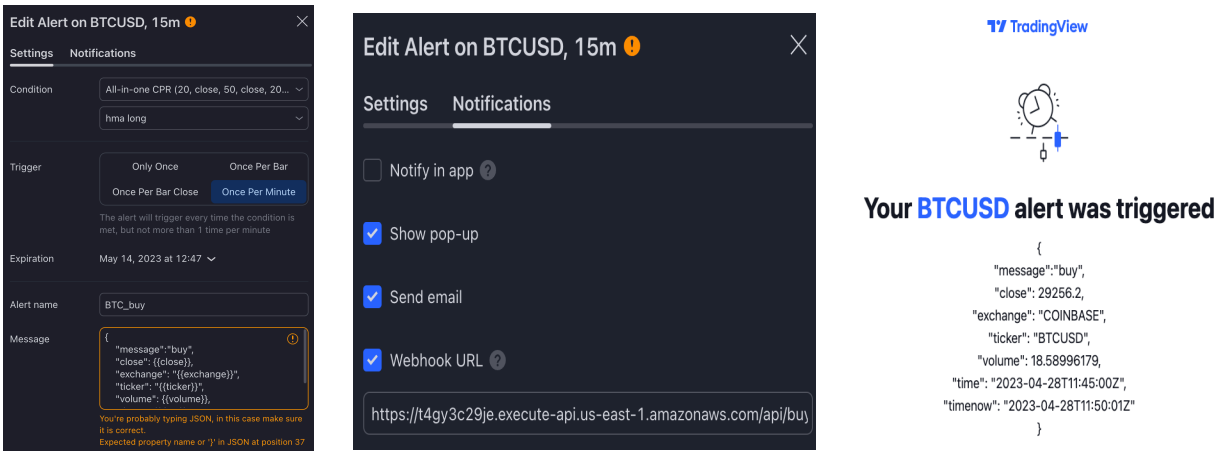


Figure 5: Buy Signal Setup

4. ALPACA API

Alpaca API is a popular REST API that provides programmatic access to various financial markets, including stocks, options, and cryptocurrencies. It is designed for developers who want to build algorithmic trading systems, automated trading bots, and other applications that interact with financial markets.

The Alpaca API is built on top of the OAuth 2.0 authentication protocol, which ensures secure access to user accounts and trading data. The API provides a wide range of functionalities, including account management, market data, order management, and trading.

Here are some of the key features of the Alpaca API:

1. Market data: The API provides real-time and historical market data for a wide range of financial instruments, including stocks, options, and cryptocurrencies.

2. Order management: The API allows users to place, modify, and cancel orders in real-time, using a variety of order types and parameters.

3. Account management: The API provides access to user account information, including balances, positions, and transaction history.

4. Trading algorithms: The API supports the development and deployment of custom trading algorithms, using a variety of programming languages and tools.

5. Integration with other platforms: The API can be integrated with various trading platforms, including TradingView, QuantConnect, and others.

6. Paper trading: The API provides a paper trading environment, which allows users to test their trading strategies in a simulated market environment.

The Alpaca API is widely used by developers, traders, and financial institutions to build a wide range of trading applications and systems. Its ease of use, flexibility, and robustness make it a popular choice for algorithmic trading and other applications that interact with financial markets.

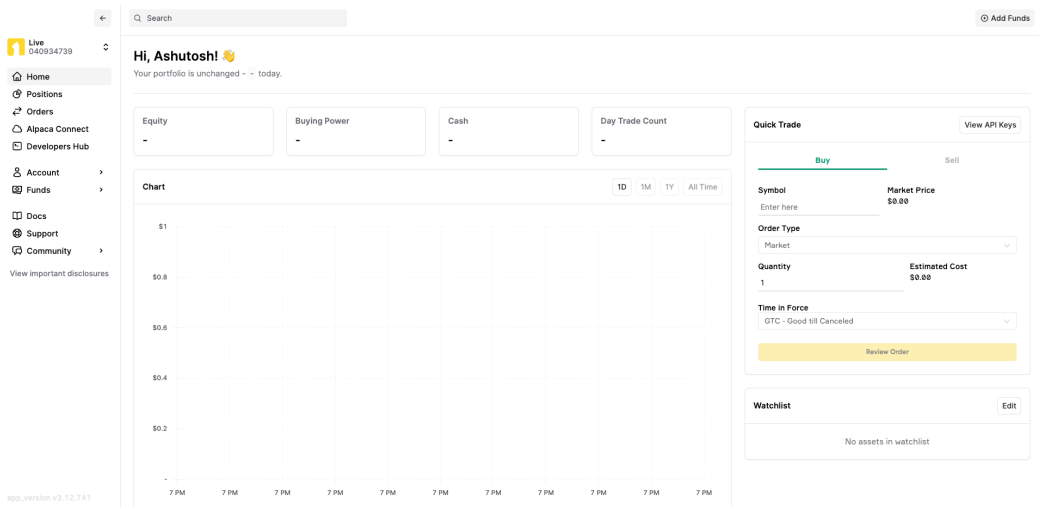


Figure 6: https://alpaca.markets/

5. CODE SETUP

The python code is setup to create a link between TradingView alert generation and Alpaca API order processing. The JSON format sent from webhookup alert is processed and a position request is sent to Alpaca paper trading API to create orders.

```
API_KEY = 'PKA1EL5D294GFMGV8GUD'
SECRET_KEY = 'Na5pJN5itYJpzMBc11i6tcth9fCHsjkzgONkcjPU'

import requests
BASE_URL = "https://paper-api.alpaca.markets"
ACCOUNT_URL = "{}/v2/account".format(BASE_URL)
ORDERS_URL = "{}/v2/orders".format(BASE_URL)
HEADERS = {'APCA-API-KEY-ID':API_KEY, 'APCA-API-SECRET-KEY':SECRET_KEY}


from chalice import Chalice


app = Chalice(app_name='tradingview-webhook-alerts')

def get_account():
  r = requests.get(ACCOUNT_URL, headers = HEADERS)
```

```
@app.route('/buy_stock', methods=['POST'])
def buy_stock():
    request = app.current_request
    webhook_message = request.json_body

    data = {
      "symbol" : webhook_message['ticker'],
      "qty" : 1,
      "side" : webhook_message['message'],
      "type" : "market",
      "time_in_force": "gtc"
    }

    r = requests.post(ORDERS_URL, json=data, headers = HEADERS)

    return {
        'webhook_message': webhook_message,
    }
```

6. AWS CHALICE

The project is then hosted on AWS platform to run throughout the day for automated trading. AWS Chalice is a framework for building serverless applications in Python. It allows developers to quickly create and deploy REST APIs, event-driven applications, and other serverless functions on AWS Lambda, with minimal setup and configuration.

Chalice is built on top of the AWS SDK for Python (Boto3), and provides a simple and intuitive interface for defining AWS Lambda functions, API endpoints, and other serverless resources. It also includes built-in support for AWS services such as Amazon API Gateway, AWS Identity and Access Management (IAM), and Amazon Simple Notification Service (SNS).

Some of the key features of AWS Chalice include:

1. Quick and easy setup: Chalice can be installed and configured with just a few commands, making it easy to get started with serverless development in Python.

2. Flask-like syntax: Chalice uses a Flask-like syntax for defining routes and handling requests, making it easy for developers familiar with Flask to get up and running quickly.

3. Automatic scaling and deployment: Chalice handles automatic scaling and deployment of serverless functions and APIs, allowing developers to focus on writing code and not worry about infrastructure management.

4. Integration with AWS services: Chalice includes built-in support for a wide range of AWS services, including API Gateway, S3, DynamoDB, and more.

5. Debugging and testing: Chalice provides tools for debugging and testing serverless applications locally, before deploying them to the cloud.

AWS Chalice is a popular choice for building serverless applications in Python, due to its ease of use, powerful features, and tight integration with AWS services. It is widely used by developers and organizations to build scalable, reliable, and cost-effective serverless applications and APIs.

```
(base) ashutosh@Ashutoshs-Air tradingmodel % chalice
Usage: chalice [OPTIONS] COMMAND [ARGS]...

Options:
  --version             Show the version and exit.
  --project-dir TEXT    The project directory path (absolute or
                        relative).Defaults to CWD
  --debug / --no-debug  Print debug logs to stderr.
  --help                Show this message and exit.

Commands:
  delete
  deploy
  dev                Development and debugging commands for chalice.
  gen-policy
  generate-models    Generate a model from Chalice routes.
  generate-pipeline  Generate a cloudformation template for a starter CD...
  generate-sdk
  invoke             Invoke the deployed lambda function NAME.
  local
  logs
  new-project
  package
  url
(base) ashutosh@Ashutoshs-Air tradingmodel %
```

Figure 7: Chalice commands

7. OUTPUT

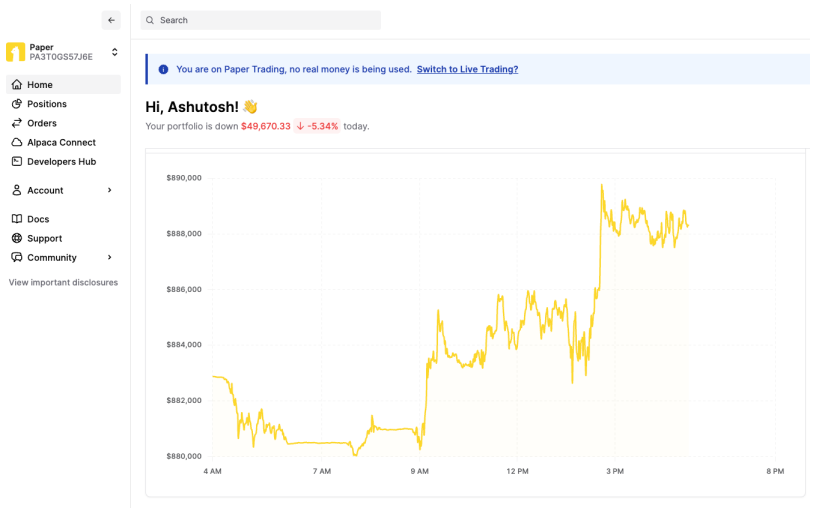


Figure 8: Alpaca Paper Trading

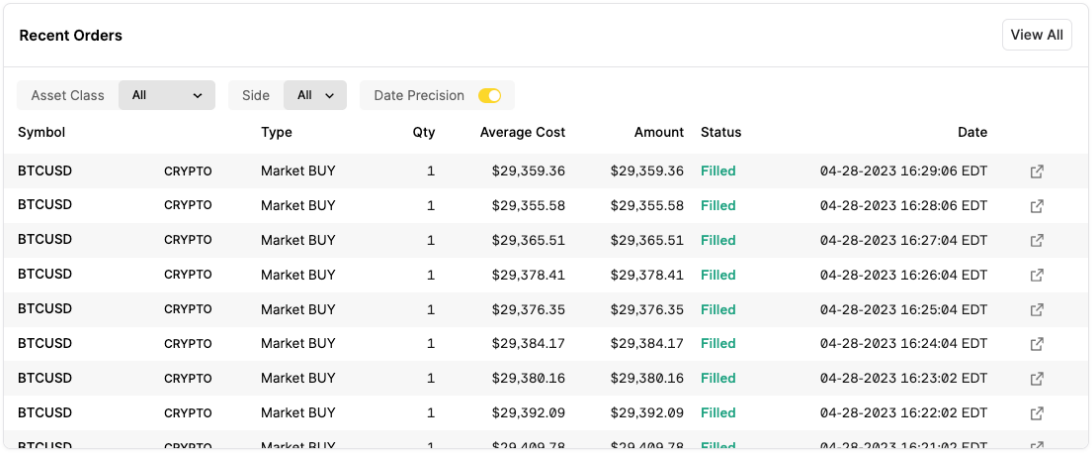


Figure 9: Trade execution

8. RESULT

The project exposes us to varied understandings of algorithmic and automated trading learning and its related concepts.
The project caters to the following learnings:
1. Research and implementation of algorithmic logics.
2. Learning about various methods to modify and generate UI based signals for trade requests.

9. REFERENCES

1. https://www.quantconnect.com/
2. https://www.quantconnect.com/docs/v2
3. https://www.tradingview.com/
4. https://www.json.org/json-en.html
5. https://alpaca.markets/
6. https://aws.github.io/chalice/

APPENDIX

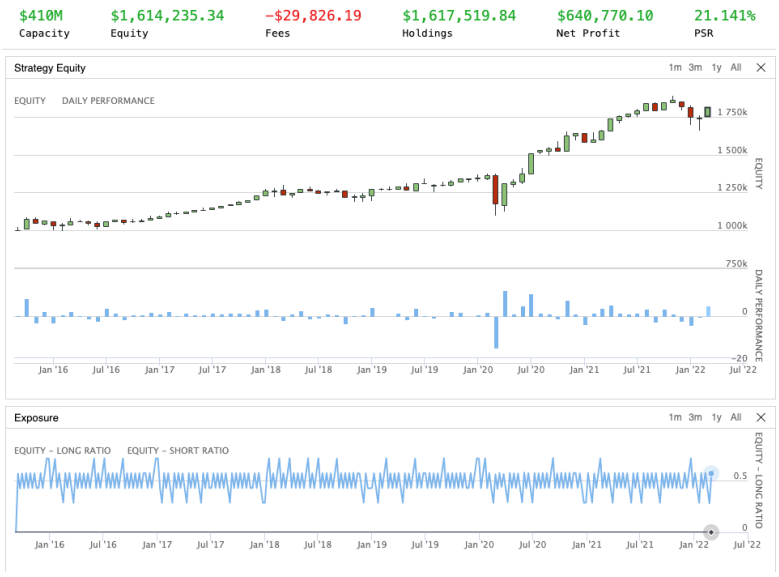
(i) Strategy: Take a position; Take profits if price goes 1% up; Take loss if price goes 1% down

Code:

```
# region imports
from AlgorithmImports import *
# endregion

#strategy = take profit with 1% up and take loss with 1% down (SPY)
class FirstCode(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2015, 9, 12)  # Set Start Date
        self.SetEndDate(2022, 3, 10)    # Set End Date
        self.SetCash(1000000)  # Set Strategy Cash

        spy = self.AddEquity("SPY", Resolution.Daily)

        spy.SetDataNormalizationMode(DataNormalizationMode.TotalReturn)


        self.spy = spy.Symbol
        self.SetBenchmark("SPY")
        self.SetBrokerageModel(BrokerageName.InteractiveBrokersBrokerage,
AccountType.Margin)

        self.entryPrice = 0
        self.period = timedelta(1)
        self.nextEntryTime = self.Time


    #OnData -> lets us set historic data and the time frontier
    def OnData(self, data: Slice):
        if not self.spy in data:
            return

        price = data.Bars[self.spy].Close

        if not self.Portfolio.Invested:
            if self.nextEntryTime <= self.Time:
                self.SetHoldings(self.spy,1)
                self.Log("BUY SPY @"+str(price))
```

```
elif self.entryPrice*1.1 < price or self.entryPrice*0.9 > price:
    self.Liquidate()
    self.Log("SELL SPY @" + str(price))
    self.nextEntryTime = self.Time + self.period
```

Back test results:



Note: The reason to choose QuantConnect is that it provided code logs and position data for the trading logics which enabled us to manually check for the strategy logic and the positions that the strategy indicated.

References:

```
2015-09-12 00:00:00    Launching analysis for 21975be182545fd1fb086371c0f33a7c with LEAN Engine v2.5.0.0.15288
:
2015-09-12 00:00:00    BUY SPY @196.74
:
2015-09-15 00:00:00    Warning: all market orders sent using daily data, or market orders sent after hours are automatically converted into MarketOnOpen orders.
:
2015-09-15 00:00:00    SELL SPY @196.01
:
2015-09-16 00:00:00    BUY SPY @198.46
:
2015-09-17 00:00:00    SELL SPY @200.18
:
2015-09-18 00:00:00    BUY SPY @199.73
:
2015-09-19 00:00:00    SELL SPY @196.49
:
2015-09-22 00:00:00    BUY SPY @197.5
:
2015-09-23 00:00:00    SELL SPY @194.95
:
2015-09-24 00:00:00    BUY SPY @194.64
:
2015-09-25 00:00:00    SELL SPY @193.94
:
2015-09-26 00:00:00    BUY SPY @193.89
:
2015-09-29 00:00:00    SELL SPY @189.05
:
2015-09-30 00:00:00    BUY SPY @189.16
:
2015-10-01 00:00:00    SELL SPY @192.67
:
2015-10-02 00:00:00    BUY SPY @193.17
```

(ii) Strategy: We combined two technical indicators: (A) SMA (B) Trend; Take long position when if near high and trend is up; Take short position when near low and trend is down

Code:

```
#region imports
from AlgorithmImports import *
#endregion
from collections import deque

class AdaptableSkyBlueHornet(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2021, 1, 1)
        self.SetEndDate(202, 1, 1)
        self.SetCash(100000)
        self.spy = self.AddEquity("SPY", Resolution.Daily).Symbol

    #    self.sma = self.SMA(self.spy, 30, Resolution.Daily)

        # History warm up for shortcut helper SMA indicator
    #    closing_prices = self.History(self.spy, 30, Resolution.Daily)["close"]
    #    for time, price in closing_prices.loc[self.spy].items():
    #        self.sma.Update(time, price)

        # Custom SMA indicator
        self.sma = CustomSimpleMovingAverage("CustomSMA", 30)
        self.RegisterIndicator(self.spy, self.sma, Resolution.Daily)


    def OnData(self, data):
        if not self.sma.IsReady:
            return

        # Save high, low, and current price
        hist = self.History(self.spy, timedelta(365), Resolution.Daily)
        low = min(hist["low"])
        high = max(hist["high"])

        price = self.Securities[self.spy].Price

        # Go long if near high and uptrending
        if price * 1.05 >= high and self.sma.Current.Value < price:
            if not self.Portfolio[self.spy].IsLong:
                self.SetHoldings(self.spy, 1)
```

```
        # Go short if near low and downtrending
        elif price * 0.95 <= low and self.sma.Current.Value > price:
            if not self.Portfolio[self.spy].IsShort:
                self.SetHoldings(self.spy, -1)

        # Otherwise, go flat
        else:
            self.Liquidate()

        self.Plot("Benchmark", "52w-High", high)
        self.Plot("Benchmark", "52w-Low", low)
        self.Plot("Benchmark", "SMA", self.sma.Current.Value)


class CustomSimpleMovingAverage(PythonIndicator):

    def __init__(self, name, period):
        self.Name = name
        self.Time = datetime.min
        self.Value = 0
        self.queue = deque(maxlen=period)

    def Update(self, input):
        self.queue.appendleft(input.Close)
        self.Time = input.EndTime
        count = len(self.queue)
        self.Value = sum(self.queue) / count
        # returns true if ready
        return (count == self.queue.maxlen)
```

Back test results:

(iii) Setting up a universe of stocks using fundamental ratios: We built a stock portfolio of 10 small cap companies which get added/removed from the portfolio on a monthly basis based on the top 10 small cap companies.

Code:

```
# region imports
# setting up the universe of stocks
# keep mind of the selection bias
# Trading Strategy: stock cap stocks have outperformed large cap stocks; 200 most
liquid stocks and then take bottom 10
# Rebalance every month
from AlgorithmImports import *
# endregion

class SwimmingAsparagusTermite(QCAlgorithm):

    def Initialize(self):
        self.SetStartDate(2020, 1, 1)  # Set Start Date
        self.SetEndDate(2022, 1, 1)
        self.SetCash(1000000)
        # rebalance time = minimum of set date
        self.rebalanceTime = datetime.min
        self.activeStocks = set()

        # add to universe of stock with coarse filter and fine filter
        self.AddUniverse(self.CoarseFilter, self.FineFilter)
        self.UniverseSettings.Resolution = Resolution.Hour
```

```
        self.portfolioTargets = []

    def CoarseFilter(self, coarse):
        # no need to rebalance within 1 month
        if self.Time <= self.rebalanceTime:
            return self.Universe.Unchanged
        self.rebalanceTime = self.Time + timedelta(30) # add one week ahead

        # filter stocks by dollarvolume
        sortedByDollarVolume = sorted(coarse, key = lambda x: x.DollarVolume, reverse
= True)
        return [x.Symbol for x in sortedByDollarVolume if x.Price > 10 and
x.HasFundamentalData][:200]

    def FineFilter(self, fine):
        sortedByPE = sorted(fine, key = lambda x: x.MarketCap)
        return [x.Symbol for x in sortedByPE if x.MarketCap > 0][:20]

    def OnSecuritiesChanged(self, changes):
        for x in changes.RemovedSecurities:
            self.Liquidate(x.Symbol)
            self.activeStocks.remove(x.Symbol)

        for x in changes.AddedSecurities:
            self.activeStocks.add(x.Symbol)

        self.portfolioTargets = [PortfolioTarget(symbol, 1/len(self.activeStocks)) for
symbol in self.activeStocks]

    def OnData(self, data: Slice):
        if self.portfolioTargets == []:
            return

        for symbol in self.activeStocks:
            if symbol not in data:
                return

        self.SetHoldings(self.portfolioTargets)
        self.portfolioTargets = []
```

Back test results:

$2.4M	$138,292.82	−$945.19	$137,625.59	$54,508.15	18.762%	38.29 %	$−16,249.42
Capacity	Equity	Fees	Holdings	Net Profit	PSR	Return	Unrealized