# Rell Trader: Server Documentation

## Introduction

Rell Trader is an advanced trading system designed to monitor trade signals from multiple symbols in the forex market, place trades, and manage user funds efficiently. The system incorporates user authentication, dynamic symbol management, trade history tracking, and real-time updates to a mobile app. This documentation provides an in-depth look at the development process, challenges faced, and the solutions implemented.

## Development Process

### Setting Up the Django Environment

1. **Project Initialization:**

The project was initialized using Django, setting up a virtual environment and installing necessary packages such as Django Rest Framework (DRF), Celery, Redis, and other dependencies. This ensured a clean and isolated environment for development and easy dependency management.

1. **App Structure:**

The project was divided into several Django apps for better modularity, making the code-base easier to manage and scale:

1. users: Manages user authentication and profiles. It handles user registration, login, password management, and profile updates.
2. trading: Handles trade signals, trade placements, and trade history. It processes incoming trade signals, executes trades, and logs each trade's details.
3. symbols: Manages the tradable symbols. It allows users to add, update, or remove symbols dynamically and maintains a list of all available symbols.
4. notifications: Sends real-time updates to the mobile app. It ensures that users receive instant notifications about trade activities and system alerts.

### Core Features

1. **User Authentication:**

Implemented using a custom built authentication system, ensuring a secure and robust user authentication process. The user model was extended to include additional fields relevant to trading, such as account balance and trading preferences. Integration with Django Rest Framework allowed for API-based token authentication, enabling secure communication between the server and client applications.

1. **Dynamic Symbol Management:**

Created models and views to manage tradeable symbols dynamically. Users can add new symbols, update existing ones, or remove them from the database. This feature ensures that the system can adapt to changing market conditions and user preferences without requiring code changes or server restarts.

1. **Trade Signal Processing and Trade Placement:**

Designed a robust system to process trade signals and manage trade placements. The system listens for incoming trade signals, evaluates them based on predefined criteria, and executes trades accordingly. Each trade's data, including entry price, exit price, profit/loss, and timestamp, is saved and retrievable for every user, providing a comprehensive trade history.

1. **Real-time Updates:**

Utilized Django Channels to handle WebSocket connections and google firebase notification system for real-time updates. This allows the server to push updates to the mobile app instantly, ensuring users receive the latest information about their trades and account status without having to refresh the app manually. The notification system was designed to handle high volumes of messages efficiently, ensuring timely delivery of updates.

## Challenges and Solutions

### Challenge: Parallel Processing of Trade Signals

**Initial Approach:**

Threading was initially used to run parallel workers for checking signals and placing trades. However, this approach presented significant limitations:

* **Unreliable Execution:** Threads would sometimes stop unexpectedly, causing missed trade opportunities and potential financial losses.
* **Resource Contention:** Threads competed for system resources, leading to performance issues and bottlenecks.
* **Complexity:** Managing thread life-cycle and synchronization was complex and error-prone, increasing the risk of bugs and system instability.

**Solution: Celery and Redis**

To address these challenges, Celery, a powerful background worker built with python to perform heavy duty tasks and Redis, a very fast database, message broker that uses the ram as primary storage, were adopted for background task processing. This approach provided a more reliable, scalable, and efficient solution for handling parallel processing of trade signals and trade placements.

**Celery Setup:**

Celery was configured with Redis as the message broker to manage task queues and ensure reliable task execution. This setup allowed tasks to be distributed across multiple worker processes, ensuring efficient use of system resources and high availability.

**Task Definition and Scheduling:**

Defined tasks for processing trade signals and placing trades, ensuring each task is handled independently and efficiently. Celery was used for parallel processing, allowing the system to check for trade signals and execute trades promptly.

### Benefits of the Solution

* **Reliability:** Redis ensures reliable queuing and execution of tasks, reducing the risk of missed trade opportunities and system failures.
* **Scalability:** Celery allows for easy scaling of workers to handle increased load efficiently. As the number of users and trade volume grows, additional worker processes can be added to maintain performance.
* **Efficiency:** Background tasks run independently of the main application thread, enhancing overall system performance. This ensures the main application remains responsive, even under high load.

**Rell Trader Money Management source code**

### Imports and Initial Setup

import asyncio

from datetime import datetime, timezone, timedelta

import pandas as pd

import vectorbt as vbt

import MetaTrader5 as mt5

import logging

from channels.db import database\_sync\_to\_async

from channels.layers import get\_channel\_layer

from functions.notifications import send\_notification\_sync

from Generate\_signals.models import Trade\_History

from signals\_auth.models import MT5Account

logger = logging.getLogger(\_\_name\_\_)

**Function:**

* Imports necessary modules and libraries.
* Sets up logging to monitor and debug the application.

### Premium\_Trade Class Initialization

class Premium\_Trade:

def \_\_init\_\_(self):

self.channel\_layer = get\_channel\_layer()

**Function:**

* Initializes the Premium\_Trade class.
* Sets up the WebSocket channel layer for real-time communication.

### get\_buy\_or\_sell\_signal Method

async def get\_buy\_or\_sell\_signal(self):

try:

bars = mt5.copy\_rates\_from(self.symbol, mt5.TIMEFRAME\_M1, datetime.now(timezone.utc), 365)

df = pd.DataFrame(bars)

df["time"] = pd.to\_datetime(df["time"], unit="s")

df = df.set\_index("time")

current\_price = df["close"].iloc[-1]

ma14 = vbt.MA.run(df["close"], 14)

ma50 = vbt.MA.run(df["close"], 50)

ma365 = vbt.MA.run(df["close"], 365)

rsi = vbt.RSI.run(df["close"], 14)

if (not (ma14.ma.iloc[-1] > ma50.ma.iloc[-1] > ma365.ma.iloc[-1] and rsi.rsi.iloc[-1] < 40)

and not (ma14.ma.iloc[-1] < ma50.ma.iloc[-1] < ma365.ma.iloc[-1] and rsi.rsi.iloc[-1] > 59)):

data = {

"status": False,

"message": "checking for signal...",

"data": {

"symbol": self.symbol,

}

}

await self.channel\_layer.group\_send(

self.room,

{

"type": "trade.format",

\*\*data

}

)

return data

elif (ma14.ma.iloc[-1] > ma50.ma.iloc[-1] > ma365.ma.iloc[-1] and rsi.rsi.iloc[-1] < 40):

data = {

"status": True,

"condition":"BUY",

"RSI":rsi.rsi.iloc[-1],

"14 SMA": ma14.ma.iloc[-1],

"Current Price": current\_price,

}

await self.channel\_layer.group\_send(

self.room,

{

"type": "trade.format",

"status": True,

"message":"BUY",

"data": {

"symbol": self.symbol,

}

}

)

return data

elif (ma14.ma.iloc[-1] < ma50.ma.iloc[-1] < ma365.ma.iloc[-1] and rsi.rsi.iloc[-1] > 59):

data = {

"status": True,

"condition":"SELL",

"RSI":rsi.rsi.iloc[-1],

"14 SMA": ma14.ma.iloc[-1],

"Current Price": current\_price

}

await self.channel\_layer.group\_send(

self.room,

{

"type": "trade.format",

"status": True,

"message":"SELL",

"data": {

"symbol": self.symbol,

}

}

)

return data

except Exception as e:

logger.error(f"Error in WebSocket task: {e}")

**Function:**

* Fetches the latest market data and converts it to a DataFrame.
* Calculates moving averages (14, 50, 365) and RSI (14) indicators.
* Checks for buy or sell signals based on the conditions of the indicators.
* Sends real-time updates via WebSocket and returns the signal data.

### check\_profit\_or\_loss Method

async def check\_profit\_or\_loss(self, initial\_balance):

account\_info = mt5.account\_info()

new\_balance = account\_info.balance

if initial\_balance < new\_balance:

return "profit"

elif initial\_balance > new\_balance:

return "loss"

else:

return "no change"

**Function:**

* Compares the initial balance with the current balance to determine if there was a profit, loss, or no change.

### check\_open\_positions Method

async def check\_open\_positions(self):

open\_positions = mt5.positions\_get()

symbol\_positions = [position for position in open\_positions if position.symbol == self.symbol]

for position in symbol\_positions:

if position.comment.lower().startswith("signal"):

self.open\_position = position

return True

return False

**Function:**

* Checks if there are any open positions for the specified symbol.
* Returns True if an open position with a comment starting with "signal" is found, and sets the open\_position attribute.

### check\_trade\_history Method

async def check\_trade\_history(self):

end\_time = datetime.now()

start\_time = end\_time - timedelta(days=30)

deals = mt5.history\_deals\_get(start\_time, end\_time)

if deals is None:

return False

self.last\_closed\_trade = None

for deal in reversed(deals):

if deal.symbol == self.symbol and deal.comment.lower().startswith("signal"):

self.last\_closed\_trade = deal

return True

return False

**Function:**

* Retrieves the trade history for the last 30 days.
* Checks if there are any closed trades with a comment starting with "signal".
* Sets the last\_closed\_trade attribute and returns True if a match is found.

### wait\_for\_trade\_close Method

async def wait\_for\_trade\_close(self, order\_id):

while True:

if not await self.check\_open\_positions():

return True

await self.channel\_layer.group\_send(

self.room,

{

"type": "trade.format",

"status": True,

"message": "active trade in progress",

"data": {

"symbol": self.open\_position.symbol,

"trade\_type": "BUY" if self.open\_position.type == 0 else "SELL",

"stop\_loss": self.open\_position.sl,

"take\_profit": self.open\_position.tp,

"open\_price": self.open\_position.price\_open,

"curent\_price": self.open\_position.price\_current

}

}

)

await asyncio.sleep(1)

**Function:**

* Waits for a trade to close by continuously checking for open positions.
* Sends real-time updates via WebSocket about the active trade in progress.

### place\_trade Method

async def place\_trade(self, symbol, volume, sl, tp, trade\_type, price):

try:

request = {

"action": mt5.TRADE\_ACTION\_DEAL,

"symbol": symbol,

"volume": volume,

"sl": sl,

"tp": tp,

"price": price,

"deviation": 200,

"magic": 123456,

"type": mt5.ORDER\_TYPE\_BUY if trade\_type == "BUY" else mt5.ORDER\_TYPE\_SELL,

"type\_filling": mt5.ORDER\_FILLING\_FOK,

"comment": f"signal.{self.current\_phase}.{self.current\_step}"

}

result = mt5.order\_send(request)

return result

except Exception as e:

logger.error(f"place trade error: {e}")

**Function:**

* Places a trade order on MetaTrader 5.
* Configures the trade request parameters including symbol, volume, stop loss, take profit, price, and trade type.
* Sends the trade request and returns the result.

### login\_and\_place\_trade\_slave Method

async def login\_and\_place\_trade\_slave(self, user, symbol, trade\_type, price):

account = await database\_sync\_to\_async(MT5Account.objects.get)(user=user)

login = await self.login\_to\_mt5(account.account, account.password, account.server)

if login:

sl, tp, lot\_size = await self.get\_sl\_tp\_lot\_size(trade\_type, price)

while True:

result = await self.place\_trade(symbol, lot\_size, sl, tp, trade\_type, price)

if result.retcode != mt5.TRADE\_RETCODE\_DONE:

logger.info("trade error")

else:

logger.info(f"trade placed for user id - {user.id}, name {user.fullname}")

break

**Function:**

* Logs into MetaTrader 5 using the account details of a slave user.
* Places a trade for the slave user based on the given symbol, trade type, and price.

### place\_buy\_or\_sell\_trade Method

async def place\_buy\_or\_sell\_trade(self, trade\_type, price):

sl, tp, lot\_size = await self.get\_sl\_tp\_lot\_size(trade\_type, price)

while True:

result = await self.place\_trade(self.symbol, lot\_size, sl, tp, trade\_type, price)

if result.retcode != mt5.TRADE\_RETCODE\_DONE:

logger.info("trade error")

else:

logger.info(f"trade placed with order\_id - {result.order}")

break

await self.place\_trade\_slave\_accounts(self.symbol, trade\_type, price, result)

return result

**Function:**

* Places a buy or sell trade for the master account.
* Configures the trade parameters including stop loss, take profit, and lot size.
* Places the trade and then places the same trade for all slave accounts.

### get\_price Method

async def get\_price(self, trade\_type):

if trade\_type == "BUY":

price = mt5.symbol\_info\_tick(self.symbol).ask

else:

price = mt5.symbol\_info\_tick(self.symbol).bid

return price

**Function:**

* Retrieves the current price for a given trade type (buy or sell).

### convert\_pips\_to\_price Method

async def convert\_pips\_to\_price(self, pips):

digits = mt5.symbol\_info(self.symbol).digits

pip\_value = 0.1 \*\* digits

return pips \* pip\_value

**Function:**

* Converts pips to a price value based on the symbol's decimal places.

### calculate\_initial\_lot\_size Method

async def calculate\_initial\_lot\_size(self, balance):

risk\_per\_trade = balance \* 0.01

symbol\_info = mt5.symbol\_info(self.symbol)

lot\_size = risk\_per\_trade / (symbol\_info.point \* symbol\_info.trade\_tick\_value)

return round(lot\_size, 2)

**Function:**

* Calculates the initial lot size based on the account balance and risk management.

### PHASES\_DATA Method

async def PHASES\_DATA(self):

initial\_lot\_size = await self.calculate\_initial\_lot\_size(mt5.account\_info().balance)

logger.info(initial\_lot\_size)

phases = {

phase: [(phase \* initial\_lot\_size, 250, 750),

(phase \* initial\_lot\_size, 250, 750),

(phase \* initial\_lot\_size, 500, 1500),

(phase \* initial\_lot\_size, 1000, 3000)]

for phase in range(1, 13)

}

return phases

**Function:**

* Generates data for different phases of trading, including lot sizes, stop loss, and take profit levels.

### get\_sl\_tp\_lot\_size Method

async def get\_sl\_tp\_lot\_size(self, trade\_type, price):

phases\_data = await self.PHASES\_DATA()

current\_phase\_data = phases\_data[self.current\_phase - 1]

sl\_pips = current\_phase\_data["sl\_pips"]

tp\_pips = current\_phase\_data["tp\_pips"]

if trade\_type == "BUY":

sl = price - await self.convert\_pips\_to\_price(sl\_pips)

tp = price + await self.convert\_pips\_to\_price(tp\_pips)

else:

sl = price + await self.convert\_pips\_to\_price(sl\_pips)

tp = price - await self.convert\_pips\_to\_price(tp\_pips)

account\_info = mt5.account\_info()

lot\_size = await self.calculate\_initial\_lot\_size(account\_info.balance)

return sl, tp, lot\_size

**Function:**

* Gets the stop loss, take profit, and lot size for the current phase and step based on the trade type and price.

### convert\_to\_two\_decimal\_places Method

async def convert\_to\_two\_decimal\_places(self, value):

return round(value, 2)

**Function:**

* Rounds a value to two decimal places if needed.

### save\_to\_db Method

async def save\_to\_db(self, user, symbol, trade\_type, price, sl, tp, lot\_size):

await database\_sync\_to\_async(Trade\_History.objects.create)(

user=user,

symbol=symbol,

order=trade\_type,

volume=lot\_size,

tp=tp,

sl=sl,

price=price,

type=trade\_type,

retcode=mt5.TRADE\_RETCODE\_DONE,

comment=f"signal.{self.current\_phase}.{self.current\_step}",

)

**Function:**

* Saves trade details to the database.

### adjust\_phases\_and\_steps Method

async def adjust\_phases\_and\_steps(self, profit\_or\_loss):

phases\_data = await self.PHASES\_DATA()

current\_phase\_data = phases\_data[self.current\_phase - 1]

if profit\_or\_loss == "profit":

if self.current\_step < current\_phase\_data["steps"]:

self.current\_step += 1

else:

self.current\_phase += 1

self.current\_step = 1

else:

if self.current\_step > 1:

self.current\_step -= 1

else:

if self.current\_phase > 1:

self.current\_phase -= 1

self.current\_step = phases\_data[self.current\_phase - 1]["steps"]

else:

self.current\_step = 1

**Function:**

* Adjusts the current phase and step based on the trade outcome (profit or loss).

### shutdown\_mt5 Method

async def shutdown\_mt5(self):

mt5.shutdown()

**Function:**

* Shuts down the MetaTrader 5 terminal.

### login\_to\_mt5 Method

async def login\_to\_mt5(self, account, password, server):

if not mt5.initialize():

logger.error("initialize() failed")

return False

if not mt5.login(account, password=password, server=server):

logger.error("login() failed")

return False

return True

**Function:**

* Logs into the MetaTrader 5 terminal with the provided account details.

### money\_management Method

async def money\_management(self, symbol, room):

self.symbol = symbol

self.room = room

self.current\_phase = 1

self.current\_step = 1

initial\_balance = mt5.account\_info().balance

while True:

await self.check\_open\_positions()

signal = await self.get\_buy\_or\_sell\_signal()

if not signal["status"]:

await asyncio.sleep(10)

continue

trade\_type = signal["condition"]

price = await self.get\_price(trade\_type)

if not await self.check\_open\_positions():

master\_result = await self.place\_buy\_or\_sell\_trade(trade\_type, price)

await self.wait\_for\_trade\_close(master\_result.order)

await self.check\_trade\_history()

profit\_or\_loss = await self.check\_profit\_or\_loss(initial\_balance)

await self.adjust\_phases\_and\_steps(profit\_or\_loss)

initial\_balance = mt5.account\_info().balance

await asyncio.sleep(1)

**Function:**

* The main method that manages the trading process.
* Continuously checks for open positions, places trades, and adjusts phases and steps based on trade outcomes.
* Runs in a loop to keep the trading system active.

### initiate\_system Method

def initiate\_system(self, \*\*kwargs):

asyncio.run(self.money\_management(\*\*kwargs))

**Function:**

* Initiates the trading system by running the money\_management method asynchronously.

This breakdown provides a detailed explanation of each block of code and its functions, making it easier to understand the automated trading system's structure and logic.

## Conclusion

Building Rell Trader involved tackling complex challenges related to parallel processing, real-time updates, and dynamic data management. Leveraging Django's robust framework, MetaTrader 5 feature packed library and integrating Celery with Redis resulted in an efficient and scalable trading system. This documentation provides a comprehensive overview of the development process, key features, and the solutions implemented to overcome challenges.

### Future Improvements

While the current implementation of Rell Trader is robust, there are always areas for improvement and enhancement:

1. **Advanced Trade Signal Algorithms:**

Incorporate more sophisticated trade signal algorithms and machine learning models to improve trade accuracy and profitability.

1. **User Interface Enhancements:**

Improve the user interface of the mobile app to provide a more intuitive and seamless user experience.

Add more customization options for users to tailor the system to their specific trading strategies and preferences.

1. **Integration with Additional Financial Markets:**

Expand the system to support trading in additional financial markets, such as stocks, commodities, and cryptocurrencies.

1. **Enhanced Security Measures:**

Implement additional security measures to protect user data and ensure the integrity of trade operations.

By continually refining and enhancing the system, Rell Trader can remain at the forefront of automated trading technology, providing users with a powerful and reliable tool for managing their investments.