Data Retriever.py

```
from datetime import datetime, timedelta
from flask import Flask, jsonify, request
import sqlite3

# Path to the file containing the date
date_file_path = "last_date.txt"

def read_date_file():
    """
    Reads the last date stored in the 'last_date.txt' file.
    Converts the date string to a Python date object.

    Returns:
        datetime.date: The date object parsed from the file.
    """
    with open(date_file_path, "r") as file:
        date_string = file.read().strip()  # Read and strip extra spaces/newlines
        return datetime.strptime(date_string, "%m/%d/%Y").date()  # Parse to date format

def get_start_end_dates():
    """
    Calculates the start and end dates based on the last date in the file.
    Adds 364 days to the last date to compute the end date to take dates within a one year interval due to
limitation of the website from which the data is scraped.

    Returns:
        tuple: A tuple containing the start date and end date as date objects.
    """
    with open(date_file_path, "r") as file:
        date_from = read_date_file()  # Read start date
        return date_from, date_from + timedelta(days=364)  # Calculate end date

def get_rows_columns():
    """
    Fetches stock data for a given symbol from the SQLite database.
    Also supports filtering based on an optional 'interval' parameter (default is 1 day).

    Query parameters (passed via HTTP GET request):
        symbol (str): The stock symbol to filter by (required).
        interval (str): The time interval for filtering data (default is '1d').

    Returns:
        tuple: A tuple containing rows (data) and columns (field names) if successful.
        OR
        Flask response: JSON error message with HTTP status code in case of failure.
    """
    symbol = request.args.get('symbol')  # Get 'symbol' parameter from the request
    interval = request.args.get('interval', '1d')  # Default interval to '1d' if not provided

    if not symbol:
        # Return an error response if 'symbol' parameter is missing
        return jsonify({"error": "Symbol parameter is required"}), 400

    try:
        # Connect to the SQLite database
```

```
    conn = sqlite3.connect('stocks_history.db')
    cursor = conn.cursor()

    # SQL query to fetch stock history for the given symbol, sorted by date in descending order
    query = "SELECT * FROM stocks_history WHERE Symbol = ? ORDER BY Date DESC"
    cursor.execute(query, (symbol,))  # Execute query with the provided symbol

    # Fetch all rows from the query
    rows = cursor.fetchall()
    conn.close()  # Close the database connection

    # Define the column names for the returned data
    columns = ["Symbol", "Date", "Last_Trade_Price", "Max", "Min", "Average_Price",
               "Change", "Volume", "Best_Turnover", "Total_Turnover"]

    # Return rows and columns as a tuple
    return rows, columns
except Exception as e:
    # Handle exceptions and return a JSON error response with status code 500
    return jsonify({"error": str(e)}), 500
```

Data scraper.py

```
import requests
from bs4 import BeautifulSoup
from datetime import datetime, timedelta

def get_symbols():
    """
    Fetches a list of stock symbols from the Macedonian Stock Exchange website.
    Parses the dropdown menu to extract valid stock symbols (ignoring symbols containing digits).

    Returns:
        list: A list of valid stock symbols (strings).
    """
    url = "https://www.mse.mk/en/stats/symbolhistory/TEL"  # URL to fetch symbols
    response = requests.get(url)  # Perform a GET request

    if response.status_code == 200:  # Check if the request was successful
        soup = BeautifulSoup(response.text, "html.parser")  # Parse the HTML content
        dropdown = soup.find("select", {"id": "Code"})  # Find the dropdown menu by its ID
        symbols = []

        # Loop through all options in the dropdown menu
        for option in dropdown.find_all("option"):
            symbol = option.get("value")  # Get the value of the option
            # Filter out symbols containing any digits
            if not any(char.isdigit() for char in symbol):
                symbols.append(symbol)

        return symbols
    else:
        print("Cannot reach site.")  # Print an error message if the site is unreachable
        return []  # Return an empty list

def get_stock_data(symbol, start_date, end_date):
    """
    Fetches historical stock data for a specific symbol from the Macedonian Stock Exchange website.
    Parses the data from an HTML table and formats it into a list of dictionaries.

    Args:
        symbol (str): The stock symbol to fetch data for.
        start_date (str): The start date in the format 'mm/dd/yyyy'.
        end_date (str): The end date in the format 'mm/dd/yyyy'.

    Returns:
        list: A list of dictionaries containing stock data for each row in the table.
    """
    url = f"https://www.mse.mk/en/stats/symbolhistory/{symbol}"  # URL for the stock's history page
    headers = {
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/86.0.4240.183 Safari/537.36",
        "Content-Type": "application/x-www-form-urlencoded",
    }
    # Form data to send with the request
    form_data = {"FromDate": start_date, "ToDate": end_date, "Code": symbol}
    response = requests.get(url, headers=headers, data=form_data)  # Perform a GET request with
headers and form data
    data = []  # Initialize an empty list to store stock data
```

```
    if response.status_code == 200:  # Check if the request was successful
        soup = BeautifulSoup(response.text, "html.parser")  # Parse the HTML content
        table = soup.find("table", {"id": "resultsTable"})  # Find the table by its ID
        if table:
            # Loop through each row in the table, skipping the header row
            for row in table.find_all("tr")[1:]:
                entry_data = row.find_all("td")  # Get all cells in the row

                # Create a dictionary for each row's data
                entry = {"date": datetime.strptime(entry_data[0].text.strip(), "%m/%d/%Y").strftime("%Y-
%m-%d")}  # Convert date format
                prices = ["last_price", "max", "min", "avg", "change", "volume", "best_turnover",
"total_turnover"]
                for i, key in zip(range(1, 9), prices):  # Iterate over column indices and corresponding keys
                    if entry_data[i].text.strip():  # If the cell is not empty
                        # Convert text to a standardized numeric format
                        num_text = entry_data[i].text.replace(',', 'X')  # Temporarily replace commas with 'X'
                        num_text = num_text.replace('.', ',')  # Replace dots with commas
                        num_text = num_text.replace('X', '.')  # Restore original dots for decimal values
                        entry[key] = num_text  # Assign the formatted value to the dictionary
                    else:
                        entry[key] = "0,00"  # Default value for empty cells

                data.append(entry)  # Add the row's data to the list
                print(f"{symbol}: {entry}")  # Print the data for debugging purposes
    else:
        print("Cannot reach site.")  # Print an error message if the site is unreachable

    return data  # Return the list of stock data
```

Database main pipeline.py

```
import os
import threading
import sqlite3
from datetime import datetime, timedelta
from concurrent.futures import ThreadPoolExecutor, wait

# Importing functions and variables from other modules
from data_retriever import date_file_path, get_start_end_dates
from data_scraper import get_symbols
from database_updater import get_and_store_data, update_last_date

def main_pipeline():
    """
    Main pipeline function to manage the retrieval, processing, and storage of stock data.
    Utilizes threading and a thread pool to handle data retrieval efficiently.

    Steps:
    1. Create and initialize the SQLite database and table if not already present.
    2. Ensure the date file exists and retrieve the start and end dates.
    3. Fetch stock symbols from the data scraper.
    4. Use a thread pool to concurrently fetch and store stock data for the given date range.
    5. Update the last processed date after completing each iteration.
    """
    write_lock = threading.Lock()  # Lock to ensure thread-safe operations on shared resources
    today = datetime.now().date()  # Get the current date
    conn = sqlite3.connect('stocks_history.db')  # Connect to the SQLite database
    cursor = conn.cursor()

    # Create the stocks_history table if it doesn't already exist
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS stocks_history (
            Symbol TEXT,
            Date TEXT,
            Last_Trade_Price REAL,
            Max REAL,
            Min REAL,
            Average_Price REAL,
            Change REAL,
            Volume REAL,
            Best_Turnover REAL,
            Total_Turnover REAL,
            PRIMARY KEY (Symbol, Date)
    )''')
    conn.commit()  # Save the table creation

    # If the date file doesn't exist, initialize it with today's date
    if not os.path.exists(date_file_path):
        update_last_date()

    # Retrieve the start and end dates for data retrieval
    start_date, end_date = get_start_end_dates()
    # Fetch the list of stock symbols
    symbols = get_symbols()
    futures = []  # List to keep track of futures for concurrent tasks

    start_time = datetime.now()  # Record the start time for performance tracking
```

```
    # Use a thread pool for concurrent execution of tasks
    with ThreadPoolExecutor() as executor:
        while start_date < today:  # Loop until all required dates are processed
            with write_lock:  # Lock to ensure thread-safe updates
                print(f"Starting thread for {start_date}")
                # Submit a task to fetch and store data for the current date range
                futures.append(executor.submit(get_and_store_data, start_date, end_date, symbols))
                update_last_date()  # Update the last processed date
                start_date, end_date = get_start_end_dates()  # Get the next date range
                print(f"Current Start Date: {start_date}")

    # Wait for all futures (tasks) to complete
    wait(futures)

    # Print the total time taken for database updates
    print(f"Time needed to create/adjust database: {(datetime.now() - start_time).total_seconds()}
seconds")
    conn.close()  # Close the database connection
```

Database updater.py

```
import sqlite3
import os
from datetime import datetime, timedelta
from data_scraper import get_stock_data  # Function to scrape stock data from the website
from data_retriever import date_file_path, read_date_file  # Utilities for handling date files


def store_data(issuer_code, cleaned_data):
    """
    Store the cleaned stock data into the SQLite database.

    Args:
        issuer_code (str): The stock symbol/issuer code.
        cleaned_data (list): List of dictionaries containing stock data for the issuer.
    """
    conn_store = sqlite3.connect('stocks_history.db')  # Connect to the SQLite database
    cursor_store = conn_store.cursor()

    # Define the column names corresponding to the cleaned data
    columns = ["date", "last_price", "max", "min", "avg", "change", "volume", "best_turnover",
"total_turnover"]

    # Iterate through each record and insert it into the database
    for record in cleaned_data:
        values = [issuer_code] + [record[column] for column in columns]
        cursor_store.execute('''
            INSERT OR IGNORE INTO stocks_history
            (Symbol, Date, Last_Trade_Price, Max, Min, Average_Price, Change, Volume, Best_Turnover,
Total_Turnover)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)''', values)

    conn_store.commit()  # Save changes to the database


def update_last_date():
    """
    Update the `last_date` in the date file, used to track the last processed date.

    Behavior:
        - If the file doesn't exist, it initializes the last date as 11 years ago.
        - If the difference between today and the last date is less than 365 days, set the last date to today.
        - Otherwise, increment the last date by 364 days.
    """
    today = datetime.now().date()  # Get the current date

    # Determine the last processed date
    if not os.path.exists(date_file_path):
        last_date = today.replace(year=today.year - 11)  # Initialize to 11 years ago
    else:
        last_date = read_date_file()

    # Update the last date based on the difference from today
    if (today - last_date).days < 365:
        last_date = today
    else:
        last_date = last_date + timedelta(days=364)
```

```
    # Write the updated last date back to the file
    with open(date_file_path, "w") as file:
        file.write(last_date.strftime("%m/%d/%Y"))


def get_and_store_data(start_date, end_date, symbols):
    """
    Fetch stock data for a given date range and list of symbols, then store it in the database.

    Args:
        start_date (datetime.date): The start date for fetching stock data.
        end_date (datetime.date): The end date for fetching stock data.
        symbols (list): List of stock symbols to process.
    """
    for symbol in symbols:
        # Fetch raw stock data for the given symbol and date range
        raw_data = get_stock_data(symbol, start_date, end_date)

        # If data is retrieved, store it in the database
        if raw_data:
            store_data(symbol, raw_data)

        # Log progress for the current symbol
        print(f"CONTINUE: {start_date} for {symbol}")
```

Tehnical analysis.py

```
import ta
from ta import momentum, trend
import pandas as pd


def preprocess_data(data):
    """
    Preprocess the input data for technical analysis.

    Steps:
        1. Convert the input data into a Pandas DataFrame.
        2. Parse the 'Date' column into a datetime format.
        3. Convert numeric columns from string to appropriate float types.
        4. Handle commas and periods in the numeric strings for proper conversion.
        5. Sort the data by date and reset the index.

    Args:
        data (list or DataFrame): Input data with required columns.

    Returns:
        DataFrame: Processed and cleaned data.
    """
    df = pd.DataFrame(data)
    df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d')

    # Convert numeric columns to floats with appropriate formatting adjustments
    for col in ['Last_Trade_Price', 'Max', 'Min', 'Average_Price', 'Change']:
        df[col] = df[col].str.replace('.', '', regex=False).str.replace(',', '.', regex=False).astype(float)

    # Convert other numeric columns with error coercion
    df['Volume'] = pd.to_numeric(df['Volume'], errors='coerce')
    df['Best_Turnover'] = pd.to_numeric(df['Best_Turnover'], errors='coerce')
    df['Total_Turnover'] = pd.to_numeric(df['Total_Turnover'], errors='coerce')

    # Sort data by date for time-series analysis
    df.sort_values(by='Date', inplace=True)
    df.reset_index(drop=True, inplace=True)

    print("Data Preprocessed.")
    return df


def calculate_indicators(df):
    """
    Calculate technical indicators for the given stock data.

    Indicators included:
        - RSI (Relative Strength Index)
        - Stochastic Oscillator
        - MACD (Moving Average Convergence Divergence)
        - Momentum
        - CCI (Commodity Channel Index)
        - SMA (Simple Moving Average)
        - EMA (Exponential Moving Average)
        - WMA (Weighted Moving Average)
        - MAE_upper and MAE_lower (SMA bands at 2% deviation)
```

```
    - HMA (Hull Moving Average)

    Args:
        df (DataFrame): Preprocessed data.

    Returns:
        DataFrame: Data with added indicator columns.
    """
    df['RSI'] = ta.momentum.RSIIndicator(close=df['Last_Trade_Price']).rsi()
    df['Stochastic'] = ta.momentum.StochasticOscillator(
        high=df['Max'], low=df['Min'], close=df['Last_Trade_Price']
    ).stoch()
    df['MACD'] = ta.trend.MACD(close=df['Last_Trade_Price']).macd()
    df['Momentum'] = df['Last_Trade_Price'].diff()
    df['CCI'] = ta.trend.CCIIndicator(
        high=df['Max'], low=df['Min'], close=df['Last_Trade_Price']
    ).cci()
    df['SMA'] = ta.trend.SMAIndicator(close=df['Last_Trade_Price'], window=14).sma_indicator()
    df['EMA'] = ta.trend.EMAIndicator(close=df['Last_Trade_Price'], window=14).ema_indicator()

    # Calculate Weighted Moving Average (WMA) manually
    df['WMA'] = df['Last_Trade_Price'].rolling(window=14).apply(
        lambda x: (x * range(1, len(x) + 1)).sum() / sum(range(1, len(x) + 1))
    )

    # Define upper and lower bands around SMA
    df['MAE_upper'], df['MAE_lower'] = df['SMA'] * 1.02, df['SMA'] * 0.98

    # Calculate Hull Moving Average (HMA)
    df['HMA'] = ta.trend.WMAIndicator(close=df['Last_Trade_Price'], window=14).wma()

    print("Indicators Calculated.")
    return df


def generate_signals(df):
    """
    Generate trading signals based on technical indicators.

    Signals generated:
        - RSI_signal: Buy if RSI < 30, Sell if RSI > 70, Hold otherwise.
        - MA_signal: Buy if SMA < EMA, Sell if SMA > EMA, Hold otherwise.

    Args:
        df (DataFrame): Data with calculated indicators.

    Returns:
        DataFrame: Data with added signal columns.
    """
    df['RSI_signal'] = df['RSI'].apply(lambda x: 'Buy' if x < 30 else 'Sell' if x > 70 else 'Hold')
    df['MA_signal'] = df.apply(
        lambda row: 'Buy' if row['SMA'] < row['EMA'] else 'Sell' if row['SMA'] > row['EMA'] else 'Hold',
axis=1
    )

    print("Signals Generated.")
    return df


def aggregate_signals(df):
```

```
    """
    Aggregate multiple signals into a single overall trading signal.

    The overall signal is determined by the mode (most frequent) of individual signals.

    Args:
        df (DataFrame): Data with individual signals.

    Returns:
        DataFrame: Data with the added overall signal column.
    """
    signals = ['RSI_signal', 'MA_signal']
    df['Overall_signal'] = df[signals].mode(axis=1)[0]  # Aggregate using mode (most common value)

    print("Signals Aggregated.")
    return df


def get_signal_dataframe(data):
    """
    Main function to preprocess data, calculate indicators, generate signals, and aggregate them.

    Args:
        data (list or DataFrame): Input stock data.

    Returns:
        DataFrame: Final DataFrame with signals and indicators.
    """
    return aggregate_signals(generate_signals(calculate_indicators(preprocess_data(data))))
```

Natural language analysis.py

```
import requests
from bs4 import BeautifulSoup
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
# nltk.download("vader_lexicon")  # Ова го активира само ако `vader_lexicon` не е веќе инсталиран


def get_news_sentiment(company_name):
    """
    Fetches the latest news articles, analyzes their sentiment, and provides a recommendation
    for the specified company based on positive, negative, and neutral sentiments.

    Args:
        company_name (str): Name of the company to analyze news sentiment for.

    Returns:
        str: A recommendation based on the sentiment analysis ("Buy", "Sell", or "Hold").
    """
    # List of URLs to scrape news from (pagination support)
    urls = [
        "https://www.mse.mk/mk/news/latest/1",
        "https://www.mse.mk/mk/news/latest/2"
    ]

    # Step 1: Fetch all news links
    news_links = []
    for url in urls:
        response = requests.get(url)
        if response.status_code != 200:
            print(f"Failed to fetch the page: {url}")
            continue

        # Parse the page and extract news links
        soup = BeautifulSoup(response.content, "html.parser")
        panel_body = soup.find("div", class_="panel-body")
        if panel_body:
            links = panel_body.find_all("a", href=True)
            news_links.extend([link["href"] for link in links])

    if not news_links:
        print("No news links found.")
        return

    # Step 2: Initialize Sentiment Analyzer
    sia = SentimentIntensityAnalyzer()

    # Sentiment tracking for the given company
    company_sentiment = {"positive": 0, "negative": 0, "neutral": 0}
    company_news_found = False

    # Step 3: Analyze sentiment for each news article
    for link in news_links:
        # Fix relative URLs if needed
        if not link.startswith("http"):
            link = f"https://www.mse.mk{link}"
```

```
    # Fetch the content of the news article
    response = requests.get(link)
    if response.status_code != 200:
        print(f"Failed to fetch news article: {link}")
        continue

    # Extract the text content of the news article
    soup = BeautifulSoup(response.content, "html.parser")
    news_text = soup.get_text()

    # Check if the company name is mentioned in the article
    if company_name.lower() in news_text.lower():
        company_news_found = True

        # Perform sentiment analysis on the article text
        sentiment_score = sia.polarity_scores(news_text)
        if sentiment_score["compound"] > 0.05:
            company_sentiment["positive"] += 1
        elif sentiment_score["compound"] < -0.05:
            company_sentiment["negative"] += 1
        else:
            company_sentiment["neutral"] += 1

# Step 4: Handle cases where no relevant news is found
if not company_news_found:
    print("No information.")
    return

# Step 5: Summarize and return recommendation
positive = company_sentiment["positive"]
negative = company_sentiment["negative"]
neutral = company_sentiment["neutral"]

print(f"Sentiment Analysis for {company_name}:")
print(f"Positive news: {positive}")
print(f"Negative news: {negative}")
print(f"Neutral news: {neutral}")

# Provide recommendation based on the sentiment
if positive > negative:
    print("Recommendation: Buy stocks.")
    return "Buy"
elif negative > positive:
    print("Recommendation: Sell stocks.")
    return "Sell"
else:
    print("Recommendation: Hold stocks.")
    return "Hold"
```

Stock price predictor.py

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from keras.models import Sequential
from keras.layers import LSTM, Dense


# Function to create sequences of input features and corresponding targets
def create_sequences(data, target_col, sequence_length=10):
    """
    Creates sequences of features and corresponding target values for time-series prediction.

    Args:
        data (pd.DataFrame): The input DataFrame with features and the target column.
        target_col (str): The name of the target column.
        sequence_length (int): Number of time steps in each sequence.

    Returns:
        tuple: Numpy arrays for sequences and their respective target values.
    """
    sequences = []
    targets = []

    # Check if the input is a Pandas DataFrame
    if isinstance(data, np.ndarray):
        raise TypeError("Expected a Pandas DataFrame, but got a NumPy array.")

    # Generate sequences and corresponding targets
    for i in range(len(data) - sequence_length):
        seq = data.iloc[i:i + sequence_length].drop(columns=[target_col]).values
        target = data.iloc[i + sequence_length][target_col]
        sequences.append(seq)
        targets.append(target)

    return np.array(sequences), np.array(targets)

# Main function to predict prices
def predict_prices(df):
    """
    Prepares the dataset, trains an LSTM model, and predicts future stock prices.

    Args:
        df (pd.DataFrame): The input DataFrame with stock market data.

    Returns:
        str: The predicted future price or a message indicating insufficient information.
    """
    # Initialize label encoder for categorical features
    encoder = LabelEncoder()

    # Handle missing and infinite values
    df = df.fillna(0)
    df = df.replace([np.inf, -np.inf], 0)

    # Drop unnecessary columns
    df = df.drop(columns=['Symbol', 'Date', 'Max', 'Min'])
```

```
    # Encode categorical signals as numerical values
    df['RSI_signal'] = encoder.fit_transform(df['RSI_signal'])
    df['MA_signal'] = encoder.fit_transform(df['MA_signal'])
    df['Overall_signal'] = encoder.fit_transform(df['Overall_signal'])

    # Create sequences and split into training and testing sets
    x, y = create_sequences(df, 'Last_Trade_Price', 10)
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.3, random_state=42, shuffle=False)

    # Build the LSTM model
    model = Sequential([
        LSTM(50, activation='relu', input_shape=(x_train.shape[1], x_train.shape[2])),
        Dense(1)
    ])
    model.compile(optimizer='adam', loss='mse')

    # Train the model
    model.fit(x_train, y_train, epochs=10, batch_size=32, verbose=1)

    # Evaluate model performance on the test set
    loss = model.evaluate(x_test, y_test, verbose=0)

    # Function to predict future prices for a specified number of steps
    def predict_future(steps):
        """
        Predicts future prices based on the trained LSTM model.

        Args:
            steps (int): Number of future time steps to predict.

        Returns:
            list: Predicted future prices.
        """
        future_predictions = []
        last_sequence = x_test[-1]  # Start with the last test sequence

        for _ in range(steps):
            next_prediction = model.predict(last_sequence[np.newaxis, :, :])[0, 0]
            future_predictions.append(next_prediction)

            # Prepare the next input sequence
            next_prediction_array = np.full((1, last_sequence.shape[1]), next_prediction)
            last_sequence = np.append(last_sequence[1:], next_prediction_array, axis=0)

        return future_predictions

    # Predict future prices (e.g., 1 day ahead)
    predictions_1_day = predict_future(1)

    # Get the last known price and calculate the adjusted value
    last_value = df['Last_Trade_Price'].tail(1).values[0]
    adjusted_value = round((float(predictions_1_day[-1]) + last_value) / 2)

    # Validate the predicted value and return the result
    if float(predictions_1_day[-1]) < 0 or adjusted_value > last_value * 2 or adjusted_value <
last_value / 2:
        return "Not enough conclusive information to make good predictions."
    else:
        return "Future Price Prediction: " + str(adjusted_value)
```

Main.py

```
import sqlite3
import numpy as np
import pandas as pd
from flask import Flask, jsonify, request
from flask_cors import CORS
from data_scraper import get_symbols
from data_retriever import get_rows_columns
from database_main_pipeline import main_pipeline
from natural_language_analysis import get_news_sentiment
from stock_price_predictor import predict_prices
from technical_analysis import get_signal_dataframe


# RETRIEVE DATA
main_pipeline()


# FRONTEND
# Define Endpoints
app = Flask(__name__)
CORS(app)

@app.route('/symbols', methods=['GET'])
def fetch_symbols():
    symbols = get_symbols()
    return jsonify(symbols), 200

@app.route('/stocks', methods=['GET'])
def fetch_stock_data():
    symbol = request.args.get('symbol')
    if not symbol:
        return jsonify({"error": "Symbol parameter is required"}), 400

    try:
        rows, columns = get_rows_columns()
        print("Rows and Columns received successfully.")
        data = [dict(zip(columns, row)) for row in rows]
        print("Created Dataframa.")
        df = get_signal_dataframe(data)
        print("Created signal dataframe.")
        sentiment = get_news_sentiment(symbol)
        print("Recieved Sentiment.")
        prediction = predict_prices(df)
        print("Predicted Prices.")
        df = df.fillna(0)
        print("Replaced Nulls.")
        df = df.replace([np.inf, -np.inf], 0)
        print("Replaced Nulls.")
        return jsonify({"dataframe": df.to_dict(orient='records'), "sentiment": sentiment, "prediction":
prediction}), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500


@app.route('/chart', methods=['GET'])
def fetch_chart_data():
    symbol = request.args.get('symbol')
```

```
    interval = request.args.get('interval', '7')

    if not symbol:
        return jsonify({"error": "Symbol parameter is required"}), 400

    try:
        rows, columns = get_rows_columns()
        df = pd.DataFrame(rows, columns=columns)
        df = df.fillna(0)
        df = df.replace([np.inf, -np.inf], 0)
        df['Date'] = pd.to_datetime(df['Date'])
        df.sort_values(by='Date', ascending=False, inplace=True)
        if interval == '7':
            filtered_df = df.head(7)
        elif interval == '30':
            filtered_df = df.head(30)
        elif interval == '60':
            filtered_df = df.head(60)
        else:
            return jsonify({"error": "Invalid interval parameter"}), 400
        filtered_data = filtered_df.to_dict(orient='records')
        return jsonify({"dataframe": filtered_data}), 200

    except Exception as e:
        return jsonify({"error": str(e)}), 500


if __name__ == '__main__':
    app.run(debug=True)
```