```
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'package:math_expressions/math_expressions.dart';

class TradingStrategy {
  final String name;
  final String description;

  TradingStrategy({
    required this.name,
    required this.description,
  });
}

class GenerateTradingStrategy extends StatefulWidget {
  const GenerateTradingStrategy({Key? key}) : super(key: key);

  @override
  _GenerateTradingStrategyState createState() => _GenerateTradingStrategyState();
}

class _GenerateTradingStrategyState extends State<GenerateTradingStrategy> {
  String _tradingStrategyName = '';
  TradingStrategy? generatedStrategy;
  bool _isGenerating = false;
```

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Flutter Trading Strategy Generator'),
    ),
    body: Column(
      children: [
        Expanded(
          child: Padding(
            padding: const EdgeInsets.all(20.0),
            child: TextField(
              decoration: const InputDecoration(
                labelText: 'Write trading strategy',
              ),
              onChanged: (value) {
                setState(() {
                  _tradingStrategyName = value;
                });
              },
            ),
          ),
        ),
        ElevatedButton(
          onPressed: () async {
```

```
    if (_tradingStrategyName.isNotEmpty) {
      setState(() {
        _isGenerating = true;
      });
      final strategy = await generateTradingStrategy(_tradingStrategyName);
      setState(() {
        generatedStrategy = strategy;
        _isGenerating = false;
      });
    } else {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(content: Text('Please write trading strategy')),
      );
    }
  },
  child: const Text('Generate Trading Strategy'),
),
if (_isGenerating)
  const Padding(
    padding: EdgeInsets.all(20.0),
    child: CircularProgressIndicator(),
  ),
if (generatedStrategy != null) ...[
  const SizedBox(height: 20),
  const Text('Generated Strategy:'),
  Expanded(
```

```
          child: Padding(
           padding: const EdgeInsets.all(20.0),
           child: SingleChildScrollView(
            child: ExpansionTile(
             title: Text(generatedStrategy!.name),
             children: [
              ListTile(
               title: const Text('Description:'),
               subtitle: Text(generatedStrategy!.description),
              ),
             ],
            ),
           ),
          ),
         ),
        ],
       ],
      ),
     );
}

Future<TradingStrategy> generateTradingStrategy(String tradingStrategyName) async {
 // Fetch data from APIs
 final alphaVantageTechnicalIndicators = await fetchAlphaVantageTechnicalIndicators();
 final financialMarketData = await fetchFinancialMarketData("AAPL"); // provide a symbol
 final sentimentData = await fetchSentimentData("AAPL"); // provide a symbol
```

```
  final symbolPrediction = await fetchSymbolPrediction("AAPL"); // provide a symbol

  // Filter data based on user input
  final filteredData = await filterData(
    tradingStrategyName,
    sentimentData as Map<String, double>,
    alphaVantageTechnicalIndicators,
    financialMarketData,
    symbolPrediction as List<Map<String, dynamic>>,
  );

  // Generate trading strategy description
  final description = getDescriptionFromInput(tradingStrategyName);

  // Return the generated trading strategy
  return TradingStrategy(
    name: tradingStrategyName,
    description: description,
  );
}

Future<Map<String, dynamic>> fetchAlphaVantageTechnicalIndicators() async {
  // Fetch data from Alpha Vantage API
  final apiKey = '8IS45HX4BAHBN0MN';
  final apiUrl = 'https://www.alphavantage.co/query';
  final url = Uri.parse('$apiUrl?function=SMA&symbol=IBM&apikey=$apiKey');
```

```
  final response = await http.get(url);

  if (response.statusCode == 200) {
    final responseBody = response.body;
    final jsonData = jsonDecode(responseBody);

    // Extract the closing prices from the JSON data
    final List<double> closingPrices = jsonData['Time Series (Daily)'].values.map((e) =>
double.parse(e['4. close'])).toList();

    // Calculate the 20-day simple moving average (SMA)
    final sma = calculateSMA(closingPrices, 20);

    // Calculate the 14-day Relative Strength Index (RSI)
    final rsi = calculateRSI(closingPrices, 14);

    // Calculate the Bollinger Bands
    final bollingerBands = calculateBollingerBands(closingPrices, 20, 2);

    // Calculate the Stochastic Oscillator
    final stochasticOscillator = calculateStochasticOscillator(closingPrices, 14);

    // Print the SMA, RSI, Bollinger Bands, and Stochastic Oscillator for each day
    for (var i = 0; i < sma.length; i++) {
```

```
      print('Day ${i + 1}: SMA = ${sma[i]}, RSI = ${rsi[i]}, Bollinger Bands = ${bollingerBands[i]}, Stochastic Oscillator = ${stochasticOscillator[i]}');
    }

    return {
      'sma': sma,
      'rsi': rsi,
      'bollingerBands': bollingerBands,
      'stochasticOscillator': stochasticOscillator,
    };
  } else {
    throw Exception('Failed to fetch Alpha Vantage technical indicators: ${response.statusCode}');
  }
}

List<double> calculateSMA(List<double> prices, int period) {
  // Calculate the Simple Moving Average (SMA)
  final sma = <double>[];
  for (var i = period - 1; i < prices.length; i++) {
    final sum = prices.sublist(i - period + 1, i + 1).reduce((a, b) => a + b);
    final average = sum / period;
    sma.add(average);
  }
  return sma;
}
```

```
List<double> calculateRSI(List<double> prices, int period) {
  // Calculate the Relative Strength Index (RSI)
  final rsi = <double>[];
  final gain = <double>[];
  final loss = <double>[];
  for (var i = 1; i < prices.length; i++) {
    final change = prices[i] - prices[i - 1];
    if (change > 0) {
      gain.add(change);
      loss.add(0);
    } else {
      gain.add(0);
      loss.add(-change);
    }
  }
  for (var i = period; i < prices.length; i++) {
    final avgGain = gain.sublist(i - period, i).reduce((a, b) => a + b) / period;
    final avgLoss = loss.sublist(i - period, i).reduce((a, b) => a + b) / period;
    final rs = avgGain / avgLoss;
    final rsiValue = 100 - (100 / (1 + rs));
    rsi.add(rsiValue);
  }
  return rsi;
}
```

```
 List<List<double>> calculateBollingerBands(List<double> prices, int period, double multiplier) {
  // Calculate the Bollinger Bands
  final sma = calculateSMA(prices, period);
  final bollingerBands = <List<double>>[];
  for (var i = period - 1; i < prices.length; i++) {
   final stdDev = _calculateStandardDeviation(prices.sublist(i - period + 1, i + 1));
   final upperBand = sma[i] + stdDev * multiplier;
   final lowerBand = sma[i] - stdDev * multiplier;
   bollingerBands.add([lowerBand, sma[i], upperBand]);
  }
  return bollingerBands;
 }

 List<double> calculateStochasticOscillator(List<double> prices, int period) {
  // Calculate the Stochastic Oscillator
  final stochasticOscillator = <double>[];
  for (var i = period - 1; i < prices.length; i++) {
   final low = prices.sublist(i - period + 1, i + 1).reduce((a, b) => a < b ? a : b);
   final high = prices.sublist(i - period + 1, i + 1).reduce((a, b) => a > b ? a : b);
   final currentPrice = prices[i];
   final stochasticOscillatorValue = ((currentPrice - low) / (high - low)) * 100;
   stochasticOscillator.add(stochasticOscillatorValue);
  }
  return stochasticOscillator;
 }
```

```
 double _calculateStandardDeviation(List<double> values) {
  // Calculate the Standard Deviation
  final mean = values.reduce((a, b) => a + b) / values.length;
  final variance = values.map((x) => (x - mean) * (x - mean)).reduce((a, b) => a + b) /
values.length;
  return variance;
 }

 Future<Map<String, dynamic>> fetchFinancialMarketData(String symbol) async {
  final apiKey = '8IS45HX4BAHBN0MN';
  final apiUrl = 'https://www.alphavantage.co/query';

  final requestUrl = Uri.parse('$apiUrl?symbol=$symbol&apikey=$apiKey');

  final response = await http.get(requestUrl);

  if (response.statusCode == 200) {
   return jsonDecode(response.body);
  } else {
   throw Exception('Failed to fetch financial market data: ${response.statusCode}');
  }
 }

 Future<List<String>> fetchSentimentData(String symbol) async {
  final yahooNewsApiKey = 'https://finance.yahoo.com/';
```

```
  final sentimentScores = <String>[];
  var symbols;
  for (final symbol in symbols) {
    var security;
    final url =
'https://api.yahoo.com/news/v1/finance/${security.symbol}?apiKey=$yahooNewsApiKey';
    final response = await http.get(Uri.parse(url), headers: {
      'Authorization': 'Bearer $yahooNewsApiKey',
    });
    if (response.statusCode == 200) {
      final data = jsonDecode(response.body);
      final articles = data['data']
          .map((article) => article['title'] + ' ' + article['summary'])
          .toList();
      final extractedText = articles.join('\n');
      final sentimentScore = await getSentimentScore(extractedText);
      sentimentScores.add(sentimentScore);
    } else {
      throw Exception('Failed to load social media data');
    }
  }
  return sentimentScores;
}

Future<Map<String, int>> fetchSymbolPrediction(String symbol) async {
  final aiApiKeys = [
```

```
  'https://github.com/ra83205/google-bard-api.git',
  'https://github.com/KoushikNavuluri/Claude-API.git',
  'https://github.com/EleutherAI/GPTNeo',
  'https://github.com/microsoft/DialoGPT.git',
];
final futures = aiApiKeys.asMap().entries.map((entry) async {
  final response = await http.post(
    Uri.parse(entry.value),
    headers: {'Authorization': 'Bearer ${aiApiKeys[entry.key]}'},
    body: jsonEncode({
      'symbol': symbol,
      'price': await fetchPrice(Security(symbol: symbol)),
    }),
  );

  if (response.statusCode == 200) {
    final responseJson = jsonDecode(response.body);
    final prediction = extractPredictionFromResponse(responseJson);
    return prediction; // Return the individual prediction
  } else {
    throw Exception('Failed to get prediction from API ${entry.key + 1}');
  }
});

final responses = await Future.wait(futures);
final predictionCounts = <String, int>{};
```

```
  for (var prediction in responses) {
    predictionCounts[prediction] = (predictionCounts[prediction] ?? 0) + 1;
  }

  return predictionCounts; // Return the counts for each prediction
}

Future<Map<String, dynamic>> filterData(
  String expression,
  Map<String, double> sentimentData,
  Map<String, dynamic> alphaVantageTechnicalIndicators,
  Map<String, dynamic> financialMarketData,
  List<Map<String, dynamic>> symbolPrediction,
) async {
  final parsedExpression = parseExpression(
    expression,
    sentimentData,
    alphaVantageTechnicalIndicators,
    financialMarketData,
    symbolPrediction,
  );

  final contextModel = MyContextModel();
  final result = parsedExpression.evaluate(contextModel as EvaluationType,
EvaluationType as ContextModel);
  if (result is bool) {
```

```
    return {'buy': result};
  } else {
    throw Exception('Invalid result type from expression evaluation.');
  }
}

Expression parseExpression(
  String expression,
  Map<String, double> sentimentData,
  Map<String, dynamic> alphaVantageTechnicalIndicators,
  Map<String, dynamic> financialMarketData,
  List<Map<String, dynamic>> symbolPrediction,
) {
  final parser = Parser();

  parser
    ..addFunction('sentiment', (arguments) {
      final symbol = arguments[0] as String;
      return sentimentData[symbol] ?? 0.0;
    })
    ..addFunction('technicalIndicator', (arguments) {
      final symbol = arguments[0] as String;
      final field = arguments[1] as String;
      return alphaVantageTechnicalIndicators[symbol][field] ?? 0.0;
    })
    ..addFunction('financialMarket', (arguments) {
```

```
      final symbol = arguments[0] as String;
      final field = arguments[1] as String;
      return financialMarketData[symbol][field] ?? 0.0;
    })
    ..addFunction('prediction', (arguments) {
      final index = arguments[0] as int;
      return symbolPrediction[index]['prediction'] ?? 0.0;
    });

  final parsedExpression = parser.parse(expression);

  return parsedExpression;
 }

 getDescriptionFromInput(String userInput) {
 // Implement this function to generate a description based on the user input
 return 'This is a placeholder description based on the user input: $userInput';
}

String getCurrentSymbolFromContext() {
 return 'AAPL'; // Replace with actual logic to get the current symbol
}

getSentimentScore(extractedText) {
 // Implement this function to get the sentiment score from the extracted text
 return 'Positive'; // Placeholder sentiment score
```

```
}

extractPredictionFromResponse(responseJson) {
  // Implement this function to extract the prediction from the response JSON
  return 'Up'; // Placeholder prediction
}

fetchPrice(security) {
  // Implement this function to fetch the price for a security
  return '100.00'; // Placeholder price
}
}

Security({required String symbol}) {
}


abstract class ExpressionContext {
  double? lookup(String variable);
}

class MyContextModel implements ExpressionContext {
  final Map<String, double> variables = {};

  @override
  double? lookup(String variable) {
```

```
    return variables[variable];
  }

  void bindVariable(Variable variable, double value) {
    variables[variable.name] = value;
  }
}

void main() {
  runApp(const MaterialApp(
    home: GenerateTradingStrategy(),
  ));
}
```