FXCM Java API

Building a strategy

By Valeri Chakarov

This tutorial is the second part of the FXM Java API tutorials series. It is tightly connected to the piece of software developed in the first part named Pulling Historical Prices. In the following paragraphs I am going to explain how to build a strategy that is based on one of the most popular indicators in technical analysis – the CCI Indicator. The prerequisites for this tutorial include a fundamental to intermediate understanding of the Java language, knowledge of object-oriented concepts and principles, and having a basic idea of how financial markets and trading works.

In this document, I am going to explain thoroughly each part of the strategy in a very understandable way. In addition, at the end of the tutorial I am going to add an Appendix and place all the source code discussed in this tutorial. Both of these will be help you take a solid grasp of how to build a strategy. Have fun coding!

Before we start, please take a moment to read the following ***disclaimer***:

Trading foreign exchange on margin carries a high level of risk, and may not be suitable for all investors. Past performance is not indicative of future results. The high degree of leverage can work against you as well as for you. Before deciding to invest in foreign exchange you should carefully consider your investment objectives, level of experience, and risk appetite. The possibility exists that you could sustain a loss of some or all of your initial investment and therefore you should not invest money that you cannot afford to lose. You should be aware of all the risks associated with foreign exchange trading, and seek advice from an independent financial advisor if you have any doubts.

Building a Strategy

Important note: to fully understand the concepts of building and testing a strategy, be sure to also read the "FXCM Java API - Testing Our Strategy" as they both go hand in hand.

We start by creating an interface called Strategy and have all strategies implement it:

```
public interface Strategy {
StrategyResult runStrategy(List <Candlestick> candleStickList);
}
```

Our interface has only one method that we must implement: `runStrategy`. This method obtains a list of instances from the Candlesticks class, applies the strategy on the Candlestick's value and returns an instance of the StrategyResult class (see "FXCM Java API - Testing Our Strategy" for more details). Using an interface would later enable us to take advantage of the polymorphism mechanism when we'll want to see the results of our strategy's backtesting.

Our strategy is quite straightforward and falls under the label Range Strategy. This type of strategy is commonly used when the market trend is unclear and the Range Strategy is a handy tool to have in this type of situation. This specific type of strategy consists of three stages. The first stage is the process of finding the range which can be done by establishing the support and resistance zones. The second stage is related to managing the risk of the strategy and placing the stop loss and the take profit orders close the support and the resistance levels respectively. The third stage involves using oscillator to time your entry. In our strategy, the Stochastics Oscillator is chosen. This technical indicator is designed to track price by a mathematical calculation which causes the indicator to fluctuate around a centerline. Traders usually will wait for the indicator to reach an extreme as price reaches a zone of support or resistance.

Firstly, we will start by explaining how we have used the stochastics indicator and how we have established the support and resistance level and in the next Chapter("Testing the strategy") we will discuss the risk management decisions taken for the creation of this strategy.

The Stochastic Oscillator is developed by George C.Lane in the 1950s. It is a momentum indicator that shows the close relative to the high-low range over a set number of periods. The reason we are interested in the momentum is because it changes before the price does and helps us in foreshadowing reversals.

Below you can see the formula for calculating the Stochastics Oscillator. For further information, you can have a look here.

```
%K = (Current Close - Lowest Low) / (Highest High - Lowest Low) * 100

%D = 3-day SMA of %K


Lowest Low = lowest low for the look-back period

Highest High = highest high for the look-back period

%K is multiplied by 100 to move the decimal point two places
```

In our strategy we will implement the Range Strategy with Stochastics Oscillator by selling the asset when the Stochastics K coefficient and the Stochastic D coefficient are below the resistance level and buy and the two coefficients are slightly above the support level. In addition, we will be tracking for crossovers between the coefficients and where exactly they happen. If it is the case that the crossover happens bellow the support level, we initiate a buy order and if it occurs above the resistance level, we make a sell order.

For the implementation of our strategy we will create RangeStrategy class:

```
public class RangeStrategy implements Strategy {

     private static final double stopLoss;
     private static final double takeProfit;
     private static final int supportLevel;
     private static final int resistanceLevel;
     static List<Double> listOfStochasticK = new ArrayList<>();;
```

```
     public RangeStrategy(double stopLoss, double takeProfit, int bottomLimit) {
            this.stopLoss = stopLoss;
            this.takeProfit = takeProfit;
            this.supportLevel = supportLevel;
            this.resistanceLevel = 100 - resistanceLevel;
;
}

                  //getters goes here...
```

As you can see we have four fields in this class. The `stopLoss` and `takeProfit` define the stop-loss and take-profit in percentage points. They are marked as final as they will be constant throughout the entire execution of each backtest but will vary from one backtest to another.

As you can see we have four fields in this class. The stopLoss and takeProfit are declared to be used for stop-loss and take-profit in percentage format. The supportLevel and resistanceLevel define the limits that will trigger making an order to buy or sell the currency. Note that the support-resistance level is always between 0 – 100, so if the Strategy argument passed to the constructor is out of this range, we'll notify it by throwing an exception. The mechanism for backtesting a strategy is to first get a hold of historical data of the specific asset we want to test the strategy on (as we did in "FXCM Java API – Pulling Historical Data"), and then running the strategy on that data. In our case, before we can run the strategy we must calculate the CCI for each and every candlestick. To achieve this we will add two static methods in RangeStrategy.class called calculateStochasticsK and calculateStochasticsD.

```
public static void calculateStochasticsK(List<CandleStick> candleSticksList, int duration)
{
		double theCurrentClose = 0;
		double theHighestHigh = 0;
		double theLowestLow = 0;
		double stochasticK = 0;
		for (int i = 0; i < candleSticksList.size(); i++) {
			//theCurrentClose
			theCurrentClose = candleSticksList.get(i).getCloseBid();
			//the Lowest Low
			List<Double> allListedLows = new ArrayList<>();
			allListedLows.add(candleSticksList.get(i).getLow());
			theLowestLow = candleSticksList.get(0).getLow();
			if(theLowestLow>candleSticksList.get(i).getLow()) {
				theLowestLow = candleSticksList.get(i).getLow();
			}
			//the Highest High
			theHighestHigh  = candleSticksList.get(0).getHigh();
			if(theHighestHigh <candleSticksList.get(i).getHigh()) {
				theHighestHigh  = candleSticksList.get(i).getHigh();
			}
			stochasticK = ((theCurrentClose - theLowestLow) / (theHighestHigh -
theLowestLow)) * 100;
			candleSticksList.get(i).setStochasticK(stochasticK);
		}
	}

	public static void calculateStochasticsD(List<CandleStick>candleSticksList, int
duration) {
		int counter = 0;
		List<Double> listOfStochasticD = new ArrayList<>();
		double stochasticD = 0;
		for (int j = 2; j <  listOfStochasticsK.size()+1; j++) {
			if(j % 3 == 0) {
				stochasticD = ( listOfStochasticsK.get(j-1) +
listOfStochasticsK.get(j-2) +  listOfStochasticsK.get(j-3))/3;
				listOfStochasticD.add(stochasticD);
				counter++;
			}
		}
		stochasticD = Utilities.addAllNumbers(listOfStochasticD)/counter;
		for (int i = 0; i < candleSticksList.size(); i++) {
			candleSticksList.get(i).setStochasticD(stochasticD);
		}

	}
```

`calculateStochasticsK` method gets two arguments: a list with all of the historical candlesticks of the asset and a certain period of the trend named as duration. The most typical used duration is 14 candles but for our strategy we will be using a 20 period window. The calculation of Stochastic K coefficient and Stochastics D coefficient are based on the formula displayed above. In the first one you take the highest price and the lowest price for the 20-period window, as well as the current close price. Once we obtain them we perform

the following calculation `stochasticK = ((theCurrentClose - theLowestLow) / (theHighestHigh - theLowestLow)) * 100;` The multiplier in the end, which is 100 is to put the result in percentage form. In addition you create stochasticsK and stochasticsD member fields in the CandleStick.class so that we can track what is their value in relation to each of the candlesticks we have in the 20-period time window. The calculateStochasticsD method takes the average of every three stochasticK values. It is also based on the formula displayed above.

Again, the new piece here is the `setStochasticsK() and the setStochasticsD()`method. We are using them to set the range of each candlestick. You need to add them as member fields in the Candlestick class (see "FXCM Java API – Pulling Historical Data" for more details) to avoid compiler error. The Candlestick object we've used in the prior module describes a generic candlestick but each strategy may require its own unique candlestick with different characteristics. As it is the case here.

Ok so now that we have made the proper adjustments to our relevant classes we can easily use the `stochasticsK and the stochasticsD` method to calculated the CCI for each candle. The next step is to implement the logic of our strategy and we are going to do that by overriding the `runStrategy` method:

```
@Override
      public StrategyResult runRangeStrategyWithStochastics(List<CandleStick>
candleSticksList) {
            double entryBuy;
            double entrySell;
            double stopLossPrice=0;
            double takeProfitPrice=0;
            double strategyProfit=0;
            double maxProfit=0;
            double maxDrawdown=0;
            boolean isOpenPosition=false;
            int winCounter=0;
            int lossCounter=0;


            for (int i = 250; i < candleSticksList.size(); i++) {

                  if(isOpenPosition) {
                        if(stopLossPrice<takeProfitPrice) {
                              if(candleSticksList.get(i).getLow()<stopLossPrice)
{
                                    isOpenPosition=false;
                                    lossCounter++;
                                    strategyProfit-=stopLoss;
                                    maxDrawdown=calculateMaxDrawdown(maxProfit,
strategyProfit, maxDrawdown);
                              }else
if(candleSticksList.get(i).getHigh()>takeProfitPrice) {
                                    isOpenPosition=false;
                                    winCounter++;
                                    strategyProfit+=takeProfit;

      maxProfit=updateMaxProfit(strategyProfit,maxProfit);
```

```
}
                        }else {

      if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                    isOpenPosition=false;
                                    lossCounter++;
                                    strategyProfit-=stopLoss;
                                }else
if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                    isOpenPosition=false;
                                    winCounter++;
                                    strategyProfit+=takeProfit;
                                    maxProfit=updateMaxProfit(strategyProfit,
maxProfit);
                                }
                        }

                    }else if(candleSticksList.get(i).getStochasticK()>supportLevel
&& candleSticksList.get(i).getStochasticK()>supportLevel) {
                        entryBuy=candleSticksList.get(i).getCloseAsk();
                        stopLossPrice= ((1-stopLoss)*entryBuy);
                        takeProfitPrice = ((1+takeProfit)*entryBuy);
                        isOpenPosition=true;
                    }else
if(candleSticksList.get(i).getStochasticK()<resistanceLevel &&
candleSticksList.get(i).getStochasticD()<resistanceLevel ) {
                        entrySell=candleSticksList.get(i).getCloseBid();
                        stopLossPrice=((1+stopLoss)*entrySell);
                        takeProfitPrice=((1-takeProfit)*entrySell);
                        isOpenPosition=true;
                    }else if(candleSticksList.get(i).getStochasticK() ==
candleSticksList.get(i).getStochasticD()){
                        entryBuy=candleSticksList.get(i).getCloseAsk();
                        stopLossPrice= ((1-stopLoss)*entryBuy);
                        takeProfitPrice = ((1+takeProfit)*entryBuy);
                        isOpenPosition=true;
                    }else if(candleSticksList.get(i).getStochasticK() ==
candleSticksList.get(i).getStochasticD() &&
candleSticksList.get(i).getStochasticK()<supportLevel &&
candleSticksList.get(i).getStochasticD()<supportLevel){
                        entryBuy=candleSticksList.get(i).getCloseAsk();
                        stopLossPrice= ((1-stopLoss)*entryBuy);
                        takeProfitPrice = ((1+takeProfit)*entryBuy);
                        isOpenPosition=true;
                    }else if(candleSticksList.get(i).getStochasticK() ==
candleSticksList.get(i).getStochasticD() &&
candleSticksList.get(i).getStochasticK()>supportLevel &&
candleSticksList.get(i).getStochasticD()>supportLevel){
                        entrySell=candleSticksList.get(i).getCloseBid();
                        stopLossPrice=((1+stopLoss)*entrySell);
                        takeProfitPrice=((1-takeProfit)*entrySell);
                        isOpenPosition=true;
```

```
                    }else
if(candleSticksList.get(i).getStochasticK()<supportLevel){
                          entryBuy=candleSticksList.get(i).getCloseAsk();
                          stopLossPrice=((1-stopLoss)*entryBuy);
                          takeProfitPrice=((1+takeProfit)*entryBuy);
                          isOpenPosition=true;
                    }else
if(candleSticksList.get(i).getStochasticK()>resistanceLevel){
                          entrySell=candleSticksList.get(i).getCloseBid ();
                          stopLossPrice=((1+stopLoss)*entrySell);
                          takeProfitPrice=((1-takeProfit)*entrySell);
                          isOpenPosition=true;
                    }
              }
              StrategyResult sr = new StrategyResult(strategyProfit, maxProfit,
maxDrawdown, winCounter, lossCounter, this);
              return sr;
        }
```

Key notes for the logic of the strategy:

There are only 3 possible states at any given moment during the time of the strategy execution: either we have 1 open long position, 1 open short position or no open positions at all. There cannot be a situation where we have more than 1 open position.

The `runStrategy` method goes through the entire list of candlesticks and checks whether there is an open position. If there is, we need to check if we got stopped-out or if our profit target got hit. If there isn't, it means that we can open a new position provided that the asset's stochasticsK and stochasticsD values are lower/higher than the floor/ceiling we defined when we instantiated the CCIIndicatorStrategy class. We set the stop-loss and take-profit immediately after we open a new position.
The price we use for opening a position is the appropriate closing price based on the side of the position: buy at ask and sell at bid. It is consistent with the calculation of the Range Strategy which is based on the *closing* prices of the relevant asset. It also means that our assumption at the core of the strategy is that we can open a trade only at the end of each candle's time interval (1 minute in our case). Obviously this assumption helps simplify the backtesting process as we are limited to the data that each candlestick has, which sometimes doesn't tell the all story. In other words, there are times when the RSI breaches our floor or ceiling during the candle's time interval, but we cannot know the exact time it happened or the exact price of the asset at that time. We are limited to the open, close, low, high of each time interval. The bottom line is this: if we are consistent with the principals used for backtesting the strategy, when running the strategy on real time data, we can expect the results would be somewhat similar. But if we deviate from those principals, the results can vary considerably.
You've probably noticed that the *for* loop at the heart of `runStrategy` starts from 250 which looks a bit odd. The reason for this is to minimize errors in time of high volatility at first calculations as it takes time for the smoothing process to kick in. Therefore, it's considered good practice to use a minimum of 250 data points prior to the starting time of our backtesting.
The `runStrategy` method returns a StrategyResult instance with all of the relevant arguments (profit, maximum drawdown, etc.) plus an instance of the RsiSignals itself. We use it to analyze the results of each backtest, as we will see in the next section: "FXCM Java API – Testing Our Strategy".

Below is a full list of the classes we've used in this tutorial:
Strategy.java:

```
public interface Strategy {
StrategyResult runStrategy(List <Candlestick> candleStickList);
}
```

RangeSrategy.class

```
public class RangeStrategy implements Strategy {

	private double stopLoss;
	private double takeProfit;
	private int supportLevel;
	private int resistanceLevel;
	static List<Double> listOfStochasticsK = new ArrayList<>();

	public RangeStrategy(double stopLoss, double takeProfit, int bottomLimit) {
		this.stopLoss = stopLoss;
		this.takeProfit = takeProfit;
		this.supportLevel = supportLevel;
		this.resistanceLevel = 100 - resistanceLevel;
	}

	public RangeStrategy() {
		// TODO Auto-generated constructor stub
	}


	public double getStopLoss() {
		return stopLoss;
	}

	public void setStopLoss(double stopLoss) {
		this.stopLoss = stopLoss;
	}

	public double getTakeProfit() {
		return takeProfit;
	}

	public void setTakeProfit(double takeProfit) {
		this.takeProfit = takeProfit;
	}

	public int getBottomLimit() {
		return supportLevel;
	}

	public void setBottomLimit(int bottomLimit) {
		this.supportLevel = bottomLimit;
	}

	public int getUpperLimit() {
```

```
            //the implementation of the class continues here…..


return resistanceLevel;
      }

      public void setUpperLimit(int upperLimit) {
            this.resistanceLevel = upperLimit;
      }

      public static void calculateStochasticsK(List<CandleStick> candleSticksList, int
duration) {
            double theCurrentClose = 0;
            double theHighestHigh = 0;
            double theLowestLow = 0;
            double stochasticK = 0;
            for (int i = 0; i < candleSticksList.size(); i++) {
                  //theCurrentClose
                  theCurrentClose = candleSticksList.get(i).getCloseBid();
                  //the Lowest Low
                  List<Double> allListedLows = new ArrayList<>();
                  allListedLows.add(candleSticksList.get(i).getLow());
                  theLowestLow = candleSticksList.get(0).getLow();
                  if(theLowestLow>candleSticksList.get(i).getLow()) {
                        theLowestLow = candleSticksList.get(i).getLow();
                  }
                  //the Highest High
                  theHighestHigh  = candleSticksList.get(0).getHigh();
                  if(theHighestHigh <candleSticksList.get(i).getHigh()) {
                        theHighestHigh  = candleSticksList.get(i).getHigh();
                  }
                  stochasticK = ((theCurrentClose - theLowestLow) / (theHighestHigh -
theLowestLow)) * 100;
                  candleSticksList.get(i).setStochasticK(stochasticK);
            }
      }

      public static void calculateStochasticsD(List<CandleStick>candleSticksList, int
duration) {
            int counter = 0;
            List<Double> listOfStochasticD = new ArrayList<>();
            double stochasticD = 0;
            for (int j = 2; j <  listOfStochasticsK.size()+1; j++) {
                  if(j % 3 == 0) {
                        stochasticD = ( listOfStochasticsK.get(j-1) +
listOfStochasticsK.get(j-2) +  listOfStochasticsK.get(j-3))/3;
                        listOfStochasticD.add(stochasticD);
                        counter++;
                  }
            }
            stochasticD = Utilities.addAllNumbers(listOfStochasticD)/counter;
            for (int i = 0; i < candleSticksList.size(); i++) {
                  candleSticksList.get(i).setStochasticD(stochasticD);
            }
                        }
```

```
                        //the implementation of the class continues here…..
      @Override
      public StrategyResult runRangeStrategyWithStochastics(List<CandleStick>
candleSticksList) {
            double entryBuy;
            double entrySell;
            double stopLossPrice=0;
            double takeProfitPrice=0;
            double strategyProfit=0;
            double maxProfit=0;
            double maxDrawdown=0;
            boolean isOpenPosition=false;
            int winCounter=0;
            int lossCounter=0;

            for (int i = 250; i < candleSticksList.size(); i++) {

                  if(isOpenPosition) {
                        if(stopLossPrice<takeProfitPrice) {
                              if(candleSticksList.get(i).getLow()<stopLossPrice) {
                                    isOpenPosition=false;
                                    lossCounter++;
                                    strategyProfit-=stopLoss;
                                    maxDrawdown=calculateMaxDrawdown(maxProfit,
strategyProfit, maxDrawdown);
                              }else if(candleSticksList.get(i).getHigh()>takeProfitPrice) {
                                    isOpenPosition=false;
                                    winCounter++;
                                    strategyProfit+=takeProfit;
                                    maxProfit=updateMaxProfit(strategyProfit,maxProfit);
                              }
                        }else {
                              if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                    isOpenPosition=false;
                                    lossCounter++;
                                    strategyProfit-=stopLoss;
                              }else if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                    isOpenPosition=false;
                                    winCounter++;
                                    strategyProfit+=takeProfit;
                                    maxProfit=updateMaxProfit(strategyProfit, maxProfit);
                              }
                        }
                  }else if(candleSticksList.get(i).getStochasticK()>supportLevel &&
candleSticksList.get(i).getStochasticK()>supportLevel) {
                        entryBuy=candleSticksList.get(i).getCloseAsk();
                        stopLossPrice= ((1-stopLoss)*entryBuy);
                        takeProfitPrice = ((1+takeProfit)*entryBuy);
                        isOpenPosition=true;
                  }else if(candleSticksList.get(i).getStochasticK()<resistanceLevel &&
candleSticksList.get(i).getStochasticD()<resistanceLevel ) {
                        entrySell=candleSticksList.get(i).getCloseBid();
                        stopLossPrice=((1+stopLoss)*entrySell);
                        //the implementation of the class continues here…..
                        takeProfitPrice=((1-takeProfit)*entrySell);
```

```
                        isOpenPosition=true;
                  }else if(candleSticksList.get(i).getStochasticK() ==
candleSticksList.get(i).getStochasticD()){
                        entryBuy=candleSticksList.get(i).getCloseAsk();
                        stopLossPrice= ((1-stopLoss)*entryBuy);
                        takeProfitPrice = ((1+takeProfit)*entryBuy);
                        isOpenPosition=true;
                  }else if(candleSticksList.get(i).getStochasticK() ==
candleSticksList.get(i).getStochasticD() &&
candleSticksList.get(i).getStochasticK()<supportLevel &&
candleSticksList.get(i).getStochasticD()<supportLevel){
                        entryBuy=candleSticksList.get(i).getCloseAsk();
                        stopLossPrice= ((1-stopLoss)*entryBuy);
                        takeProfitPrice = ((1+takeProfit)*entryBuy);
                        isOpenPosition=true;
                  }else if(candleSticksList.get(i).getStochasticK() ==
candleSticksList.get(i).getStochasticD() &&
candleSticksList.get(i).getStochasticK()>supportLevel &&
candleSticksList.get(i).getStochasticD()>supportLevel){
                        entrySell=candleSticksList.get(i).getCloseBid();
                        stopLossPrice=((1+stopLoss)*entrySell);
                        takeProfitPrice=((1-takeProfit)*entrySell);
                        isOpenPosition=true;
                  }else if(candleSticksList.get(i).getStochasticK()<supportLevel){
                        entryBuy=candleSticksList.get(i).getCloseAsk();
                        stopLossPrice=((1-stopLoss)*entryBuy);
                        takeProfitPrice=((1+takeProfit)*entryBuy);
                        isOpenPosition=true;
                  }else if(candleSticksList.get(i).getStochasticK()>resistanceLevel){
                        entrySell=candleSticksList.get(i).getCloseBid ();
                        stopLossPrice=((1+stopLoss)*entrySell);
                        takeProfitPrice=((1-takeProfit)*entrySell);
                        isOpenPosition=true;
                  }
            }
            StrategyResult sr = new StrategyResult(strategyProfit, maxProfit, maxDrawdown,
winCounter, lossCounter, this);
            return sr;
      }
      private static double updateMaxProfit(double strategyProfit, double maxProfit) {
            if(strategyProfit>maxProfit) {
                  maxProfit=strategyProfit;
            }
            return maxProfit;
      }
      private static double calculateMaxDrawdown(double maxProfit, double strategyProfit,
double maxDrawdown) {
            double currentDrawdown = ((1+strategyProfit)-(1+maxProfit))/(1+maxProfit);
            if(currentDrawdown<maxDrawdown) {
                  maxDrawdown=currentDrawdown;
            }
            return maxDrawdown;
      }
   }
```