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 back testing.

Our strategy is quite straightforward and falls under the label Breakout Strategy. A breakout strategy is a stock price that moves outside a defined support or resistance level with increased volume. A breakout trader enters a long position after the stock price breaks above resistance or enters a short position after the stock breaks below support. The importance of the breakouts is that they are indicative of future volatility increase and large price swings in the future. In this tutorial we are going to present an example of a Breakout Strategy by using Donchian Channel.

The Donchian Channel was developed in 1930s and is used in technical analysis to measure a market's volatility. It is a banded indicator which measures market volatility. In addition, it is used to identify potential breakouts or overbought/oversold conditions when price reaches either the Upper or Lower Band. These instances would indicate possible trading signals.

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

```
For this example, a 20 day period is used which is a very commonly used
timeframe.

Upper Channel = 20 Day High;
Lower Channel = 20 Day Low;
Middle Channel = (20 Day High + 20 Day Low)/2;
```

In our Breakout Strategy we will implement the Donchian channel by taking a defined number of periods (20 days for example and calculates the Upper and Lower Bands.) The Upper band is the high price for the period. The Lower Band is the low price for the period. The Middle Line is simply the average of the two. The main function of Donchian Channels is to measure volatility. When volatility is high, the bands will widen wand when volatility is low, the bands become narrower. When price reaches or breaks through one of the bands, this indicate an overbought or oversold condition. This can essentially result in one of two things. First, the current trend has been confirmed and the breakthrough will cause a significant move in price in the same direction. The second results is that the trend has already been confirmed and the

breakthrough indicates a possible small reversal before continuing in the same direction. More information could be found here and here.

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

```
public class BreakoutStrategy implements Strategy {

    private static final double STOP_LOSS;
    private static final TAKE_PROFIT;
    public double FLOOR_DONCHIAN;
    public static double CEILING_DONCHIAN;
    double upperDonchianChannel = 0;
    double lowerDonchianChannel = 0;
    List<Double> candleStickHighPrices = new ArrayList<>();
    List<Double> candleStickLowPrices = new ArrayList<>();
    List<CandleStick> candleSticksList;
```

```
    public BreakoutStrategy(double stopLoss, double takeProfit, double floorDonchian) {
        super();
        STOP_LOSS = stopLoss;
        TAKE_PROFIT = takeProfit;
        FLOOR_DONCHIAN = floorDonchian;
        CEILING_DONCHIAN = 1-floorDonchian;
    }
            //getters goes here...
```

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 `FLOOR_DONCHIAN` and *`CEILING_DONCHIAN`* 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 – 1, 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 Donchian Channel Limits for each and every candlestick. To achieve this we will add two static methods in BreakoutStrategy.class called calculateUpperDonchianCHannel and calculateLowerDonchianCHannel.

The method for implementing the formula above is called `calculateDonchianIndicatorForDataSet.` In this method, the two methods (calculateUpperDOnchianChannel and calculateLowerDonchianChannel) are called and the method calculateDonchianIndicator is called from the class DonchianIndicator class where the two averages for the Low and the High prices are added and divided by two.

```
	public static void calculateDonchianIndicatorForDataSet(List<CandleStick>
candleSticksList, int duration){
		double donchianChannel = 0;
		double upperDonchianChannel = 0;
		double lowerDonchianChannel = 0;
		List<Double> candleStickLowPrices = new ArrayList<>();
		List<Double> candleStickHighPrices = new ArrayList<>();
		DonchianIndicator donchianInd = new DonchianIndicator();

		for (int i = 0; i < candleSticksList.size(); i++) {
			candleStickLowPrices.add(candleSticksList.get(i).getLow());
			candleStickHighPrices.add(candleSticksList.get(i).getHigh());

		}
		upperDonchianChannel =
calculateUpperDonchianChannel(List<CandleStick>candleSticksList)/20;
		lowerDonchianChannel =
calculateUpperDonchianChannel(List<CandleStick>candleSticksList)/20;
		donchianChannel =
donchianInd.calculateDonchianIndicator(lowerDonchianChannel, upperDonchianChannel);
		for (CandleStick price : candleSticksList) {
			price.setDonchianChannel(donchianChannel);
		}
	}
```

```
	public double calculateUpperDonchianChannel(List<CandleStick>candleSticksList) {
		double donchianChannel = 0;
		double upperDonchianChannel = 0;
		double lowerDonchianChannel = 0;
		List<Double> candleStickLowPrices = new ArrayList<>();
		List<Double> candleStickHighPrices = new ArrayList<>();
		DonchianIndicator donchianInd = new DonchianIndicator();

		for (int i = 0; i < candleSticksList.size(); i++) {
			candleStickLowPrices.add(candleSticksList.get(i).getLow());
		}
		upperDonchianChannel = Utilities.addAllNumbers(candleStickLowPrices)/20;
		return upperDonchianChannel;
	}
	public double calculateLowerDonchianChannel(List<CandleStick>candleSticksList) {
		double donchianChannel = 0;
		double upperDonchianChannel = 0;
		double lowerDonchianChannel = 0;
		List<Double> candleStickLowPrices = new ArrayList<>();
		List<Double> candleStickHighPrices = new ArrayList<>();
		DonchianIndicator donchianInd = new DonchianIndicator();

		for (int i = 0; i < candleSticksList.size(); i++) {
			candleStickHighPrices.add(candleSticksList.get(i).getHigh());

		}
```

```
                    //the code continues here….
              lowerDonchianChannel = Utilities.addAllNumbers(candleStickHighPrices)/20;
            return lowerDonchianChannel;
      }
```

`calculateUpperDonchian` 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. You go through all the candlestick values and add the low in the cnadleStickLowPrices list which is then divided by the defined period to find the upper donchian level.

Analogically, you implement the same logic for calculating the lower bound of the donchian channel by going through all the candlestick values, extracting the High value of the candlesticks and putting it in a list which is then divided by the defined time period.

The next step is to implement the method `runStrategyWithDonchian.`

```
@Override
public StrategyResult runStrategy(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-=STOP_LOSS;
                                          maxDrawdown=calculateMaxDrawdown(maxProfit,
strategyProfit, maxDrawdown);
                                    }else
if(candleSticksList.get(i).getHigh()>takeProfitPrice) {
                                          isOpenPosition=false;
                                          winCounter++;
                                          strategyProfit+=TAKE_PROFIT;

      maxProfit=updateMaxProfit(strategyProfit,maxProfit);
                                    }
                              }else {

      if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                          isOpenPosition=false;
                                          lossCounter++;
                                          strategyProfit-=STOP_LOSS;
                                    }else
if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                          isOpenPosition=false;
                                          winCounter++;
                                          strategyProfit+=TAKE_PROFIT;
                                          maxProfit=updateMaxProfit(strategyProfit,
maxProfit);
                                    }
                              }
                              //System.out.println(candleSticksList.get(i).getCCI() +
"THis is the CCI for the FLOOR");
                        }else
if(candleSticksList.get(i).getDonchianChannel()<FLOOR_DONCHIAN) {
                              entryBuy=candleSticksList.get(i).getCloseAsk();
                              stopLossPrice= ((1-STOP_LOSS)*entryBuy);
                              takeProfitPrice = ((1+TAKE_PROFIT)*entryBuy);
                              isOpenPosition=true;
                        }else
if(candleSticksList.get(i).getDonchianChannel()>CEILING_DONCHIAN) {
                              entrySell=candleSticksList.get(i).getCloseBid();
                              stopLossPrice=((1+STOP_LOSS)*entrySell);
                              takeProfitPrice=((1-TAKE_PROFIT)*entrySell);
                              isOpenPosition=true;
                        }
                  }
                  StrategyResult sr = new StrategyResult(strategyProfit, maxProfit,
maxDrawdown, winCounter, lossCounter, this);

                  return sr;
            }
```

The logic of the strategy is implemented by the `runStrategy` method shown above, with the help of the 2 static methods `calculateMaxDrawdown` and `updateMaxProfit`:

```
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;
}

private static double updateMaxProfit(double strategyProfit, double maxProfit) {
         if(strategyProfit>maxProfit) {
                maxProfit=strategyProfit;
         }
         return maxProfit;
}
```

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 RSI is lower/higher than the floor/ceiling we defined when we instantiated the BreakoutStrategy 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 RSI 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 that RSI is very volatile at first calculations as it takes time for the smoothing process to kick in (see the formula for more details). 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 need this instance as it holds the values of the parameters used for that particular backtest i.e. stop-loss, take-profit, RSI values. 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 BreakoutStrategy implements Strategy {

	private double STOP_LOSS;
	private double TAKE_PROFIT;
	public double FLOOR_DONCHIAN;
	public static double CEILING_DONCHIAN;
	double upperDonchianChannel = 0;
	double lowerDonchianChannel = 0;
	List<Double> candleStickHighPrices = new ArrayList<>();
	List<Double> candleStickLowPrices = new ArrayList<>();
	List<CandleStick> candleSticksList;

	public BreakoutStrategy() {
	}

	public BreakoutStrategy(double stopLoss, double takeProfit, double floorDonchian) {
		super();
		STOP_LOSS = stopLoss;
		TAKE_PROFIT = takeProfit;
		FLOOR_DONCHIAN = floorDonchian;
		CEILING_DONCHIAN = 1-floorDonchian;
	}
	public static void calculateDonchianIndicatorForDataSet(List<CandleStick>
candleSticksList, int duration){
		double donchianChannel = 0;
		double upperDonchianChannel = 0;
		double lowerDonchianChannel = 0;
		List<Double> candleStickLowPrices = new ArrayList<>();
		List<Double> candleStickHighPrices = new ArrayList<>();
		DonchianIndicator donchianInd = new DonchianIndicator();

		for (int i = 0; i < candleSticksList.size(); i++) {
			candleStickLowPrices.add(candleSticksList.get(i).getLow());
			candleStickHighPrices.add(candleSticksList.get(i).getHigh());

		}
		upperDonchianChannel = Utilities.addAllNumbers(candleStickLowPrices)/20;
		lowerDonchianChannel = Utilities.addAllNumbers(candleStickHighPrices)/20;
		donchianChannel = donchianInd.calculateDonchianIndicator(lowerDonchianChannel,
upperDonchianChannel);
		for (CandleStick price : candleSticksList) {
			price.setDonchianChannel(donchianChannel);
		}
	}
```

```
	public double calculateUpperDonchianChannel(List<CandleStick>candleSticksList) {
		double donchianChannel = 0;
		double upperDonchianChannel = 0;
		double lowerDonchianChannel = 0;
		List<Double> candleStickLowPrices = new ArrayList<>();
		List<Double> candleStickHighPrices = new ArrayList<>();
		DonchianIndicator donchianInd = new DonchianIndicator();

		for (int i = 0; i < candleSticksList.size(); i++) {
			candleStickLowPrices.add(candleSticksList.get(i).getLow());
			candleStickHighPrices.add(candleSticksList.get(i).getHigh());

		}
		upperDonchianChannel = Utilities.addAllNumbers(candleStickLowPrices)/20;
		return upperDonchianChannel;
	}

	public double calculateLowerDonchianChannel(List<CandleStick>candleSticksList) {
		double donchianChannel = 0;
		double upperDonchianChannel = 0;
		double lowerDonchianChannel = 0;
		List<Double> candleStickLowPrices = new ArrayList<>();
		List<Double> candleStickHighPrices = new ArrayList<>();
		DonchianIndicator donchianInd = new DonchianIndicator();

		for (int i = 0; i < candleSticksList.size(); i++) {
			candleStickLowPrices.add(candleSticksList.get(i).getLow());
			candleStickHighPrices.add(candleSticksList.get(i).getHigh());

		}
		lowerDonchianChannel = Utilities.addAllNumbers(candleStickHighPrices)/20;
		return lowerDonchianChannel;
	}

	@Override
	public StrategyResult runStrategy(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-=STOP_LOSS;
                                        maxDrawdown=calculateMaxDrawdown(maxProfit,
strategyProfit, maxDrawdown);
                                  }else if(candleSticksList.get(i).getHigh()>takeProfitPrice) {
                                        isOpenPosition=false;
                                        winCounter++;
                                        strategyProfit+=TAKE_PROFIT;
                                        maxProfit=updateMaxProfit(strategyProfit,maxProfit);
                                  }
                            }else {
                                  if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                        isOpenPosition=false;
                                        lossCounter++;
                                        strategyProfit-=STOP_LOSS;
                                  }else if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                        isOpenPosition=false;
                                        winCounter++;
                                        strategyProfit+=TAKE_PROFIT;
                                        maxProfit=updateMaxProfit(strategyProfit, maxProfit);
                                  }
                            }
                      }else if(candleSticksList.get(i).getDonchianChannel()<FLOOR_DONCHIAN) {
                            entryBuy=candleSticksList.get(i).getCloseAsk();
                            stopLossPrice= ((1-STOP_LOSS)*entryBuy);
                            takeProfitPrice = ((1+TAKE_PROFIT)*entryBuy);
                            isOpenPosition=true;
                      }else if(candleSticksList.get(i).getDonchianChannel()>CEILING_DONCHIAN) {
                            entrySell=candleSticksList.get(i).getCloseBid();
                            stopLossPrice=((1+STOP_LOSS)*entrySell);
                            takeProfitPrice=((1-TAKE_PROFIT)*entrySell);
                            isOpenPosition=true;
                      }
                }
                StrategyResult sr = new StrategyResult(strategyProfit, maxProfit, maxDrawdown,
winCounter, lossCounter, this);

                return sr;
          }

          private double updateMaxProfit(double strategyProfit, double maxProfit) {

                if(strategyProfit>maxProfit) {
                      maxProfit=strategyProfit;
                }

                return maxProfit;
          }

          private double calculateMaxDrawdown(double maxProfit, double strategyProfit, double
maxDrawdown) {

                double currentDrawdown = ((1+strategyProfit) - (1+maxProfit))/(1+maxProfit);

                if (currentDrawdown<maxDrawdown) {
```

```
                    maxDrawdown=currentDrawdown;
            }

            return maxDrawdown;
      }

      public double getSTOP_LOSS() {
            return STOP_LOSS;
      }
      public void setSTOP_LOSS(double sTOP_LOSS) {
            STOP_LOSS = sTOP_LOSS;
      }
      public double getTAKE_PROFIT() {
            return TAKE_PROFIT;
      }
      public void setTAKE_PROFIT(double tAKE_PROFIT) {
            TAKE_PROFIT = tAKE_PROFIT;
      }
      public double getFLOOR_DONCHIAN() {
            return FLOOR_DONCHIAN;
      }
      public void setFLOOR_DONCHIAN(int fLOOR_DONCHIAN) {
            FLOOR_DONCHIAN = fLOOR_DONCHIAN;
      }
      public double getCEILING_DONCHIAN() {
            return  CEILING_DONCHIAN;
      }
      public void setCEILING_DONCHIAN(int cEILING_DONCHIAN) {
            CEILING_DONCHIAN = cEILING_DONCHIAN;
      }
}
```

DonchianIndicator.class

```
public class DonchianIndicator {

	public Double calculateDonchianIndicator(Double sumOfHighs, Double
sumOfLows) {
		double donchianChannel = (sumOfHighs + sumOfLows)/2;
		return donchianChannel;
	}
  }
```