FXCM Java API

Building a strategy

By Itamar Eden

This tutorial is the second part of the FXCM Java API tutorials series. In the first part we saw how to effectively pull historical data from the FXCM API. In this part I'm going to demonstrate and explain how to build a strategy that is based on one of the most popular indicators in technical analysis – the RSI.

The prerequisites for this tutorial include a basic to moderate understanding of the Java language, including object oriented approach for developing applications, and also a basic understanding of the financial markets and trading concepts.

In this tutorial I'm going to go over each part of the strategy in details, step by step, in a straightforward way. At the end of this tutorial you can find the complete listing of all source code files. So stay tuned and see how you can take advantage of the FXCM Java API to build and test your own strategies!

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

In this tutorial we are going to build a framework for generating strategies. our approach is to make it as generic as possible so whenever a good idea for a strategy comes to our mind, we would be able to implement it easily by just changing the logic at the core of the strategy's class.

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 gets a list of candlesticks, runs the strategy on them 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, but more on that later.

The strategy we are going to build is quite simple and includes the use of a very popular oscillator called Relative Strength Index, or in short – RSI. The RSI has been around for decades and is definitely one of the most used indicators in technical analysis. It was developed by J. Welles Wilder in 1978. In short, the RSI is a momentum indicator that compares the magnitude of recent gains and losses over a specified time period to measure speed and change of price movements of an asset. It is primarily used as a contrarian indicator to attempt to identify if the asset is in overbought or oversold territory. The RSI oscillates between 0 and 100. Traditionally, an asset is considered overbought when the RSI is above 70 and oversold when the RSI is below 30.

In our strategy we will implement the RSI principals by selling the asset when its RSI is above a predefined ceiling threshold and buy the asset when its RSI is below a predefined floor threshold.

For the implementation of our strategy we will create `RsiSignals` class:

```
public class RsiSignals implements Strategy {

	private final double STOP_LOSS;
	private final double TAKE_PROFIT;
	private final int FLOOR_RSI;
	private final int CEILING_RSI;

	public RsiSignals(double stopLoss, double takeProfit, int floorRSI) throws Exception {

		if(floorRSI<=0 || floorRSI>=100) {
			throw new Exception("Invalid RSI: "+ floorRSI +". Should be between 0-100");
		}
		this.STOP_LOSS = stopLoss;
		this.TAKE_PROFIT = takeProfit;
		this.FLOOR_RSI = floorRSI;
		this.CEILING_RSI = 100-floorRSI;
	}

			// getters goes here...
}
```

As you can see we have four fields in this class. The `STOP_LOSS` and `TAKE_PROFIT` 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.

The `FLOOR_RSI` and `CEILING_RSI` define the RSI rates that will trigger opening a new trade to buy/sell the asset respectively. They are also marked as final. Note that the RSI is always between 0 – 100, so if the RSI 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 RSI for each and every candlestick. To achieve that we will add a static method in `RsiSignals`:

```
public static void calculateRsiForDataSet(List<Candlestick> candleSticksList, int duration) {

        // duration must be smaller than the data set size
        if(duration>=candleSticksList.size()) return ;

        double sessionGainSum = 0;
        double sessionLossSum = 0;
        double sessionChange;

        for (int i = 1; i <= duration; i++) {    // calculate first RSI.

           sessionChange = candleSticksList.get(i).getClose() - candleSticksList.get(i - 1).getClose();
           if (sessionChange > 0) {
                sessionGainSum += sessionChange;
           }else{
                sessionLossSum += sessionChange;
           }
        }

        double averageGain = sessionGainSum / duration;
        double averageLoss = sessionLossSum / duration;
        double rs =  (averageGain / -averageLoss);
        double rsi = 100 - (100 / (1 + rs));             // first RSI.
        candleSticksList.get(duration).setRsi(rsi);

        for (int i = duration + 1; i < candleSticksList.size(); i++) {    // for smoothing

           sessionChange = candleSticksList.get(i).getClose() - candleSticksList.get(i - 1).getClose();

           if (sessionChange > 0) {
                averageGain = (averageGain*(duration-1) + sessionChange)/duration;
                averageLoss = (averageLoss*(duration-1) + 0)/duration;
           }else{
                averageGain = (averageGain*(duration-1) + 0)/duration;
                averageLoss = (averageLoss*(duration-1) + sessionChange)/duration;
           }

           rs = (averageGain / -averageLoss);
           rsi = 100 - (100 / (1 + rs));
           candleSticksList.get(i).setRsi(rsi);
        }
}
```

`calculateRsiForDataSet` method gets two arguments: a list with all of the historical candlesticks of the asset and the duration of the RSI. The most typical used duration is 14 candles and that's what we are going to use for our strategy. We'll not go through the calculation of RSI in details as it's not the purpose

of this tutorial, but for those who want to explore it more thoroughly, I recommend this site for more information.

One thing looks odd though, and that's the `setRsi(rsi)` method we are using to set the RSI for each candlestick. It's odd since it does not exist in the Candlestick object (see "FXCM Java API – Pulling Historical Data" for more details) and so it results in a 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. To tackle this issue, let's change the Candlestick class to fit to our current needs, and add the extra fields/methods we will need to use for the execution of our strategy.

The Candlestick class should have a field called rsi with its getter and setter. In addition, as we will see in a bit, for the proper execution of our strategy we should have 2 different closing prices: bid and ask. Since the generic Candlestick class had only one close field which refers to the bid price, we should add a closeAsk field to store the closing ask price of the asset. After updating it, the Candlestick class looks like this:

```
public class Candlestick{

	private String date;
	private double open;
	private double low;
	private double high;
	private double closeBid;
	private double closeAsk;
	private double rsi;

	public Candlestick(String date, double open, double low, double high, double closeBid,
			double closeAsk) {

		this.date = date;
		this.open = open;
		this.low = low;
		this.high = high;
		this.closeBid = closeBid;
		this.closeAsk = closeAsk;
	}

	public Candlestick(String date, double open, double low, double high, double closeBid) {

		this.date = date;
		this.open = open;
		this.low = low;
		this.high = high;
		this.closeBid = closeBid;
	}

			// getters and setters...


}
```

As you can see, now we have 2 closing prices-bid and ask. Also we have a new field called `rsi`, which we will use to store the rsi we calculate for each candle. In addition to the original constructor, we add a new one, which accepts the 2 closing prices – bid and ask.

There is still one problem though: in the method we use to convert historical data into candlesticks in the `HistoryMiner` class (see "FXCM Java API – Pulling Historical Data" for more details), we have only 1 closing price:

```
public void convertHistoricalRatesToCandleSticks(){

    // get the keys of the historicalRates map into a sorted list
    SortedSet<UTCDate> dateList = new TreeSet<>(historicalRates.keySet());
    // define a format for the dates
    SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy HH:mm");
    // make the date formatter above convert from GMT to BST
    sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));

    for(UTCDate date : dateList){

      MarketDataSnapshot candleData;

      candleData = historicalRates.get(date);
      // convert the key to a Date
      Date candleDate = date.toDate();
      String sdfDate = sdf.format(candleDate);
      double open = candleData.getBidOpen();
      double low = candleData.getBidLow();
      double high = candleData.getBidHigh();
      double close = candleData.getBidClose();

      Candlestick candlestick = new Candlestick(sdfDate, open, low, high, close);

      candlesticksList.add(candlestick);
    }
}
```

Let's fix it by adding a `closeAsk` variable and changing the close to `closeBid`. We should also change the constructor we use to initialize each candlestick instance to the new one, which accepts both closing prices:

```
public void convertHistoricalRatesToCandleSticks(){

    // get the keys of the historicalRates map into a sorted list
    SortedSet<UTCDate> dateList = new TreeSet<>(historicalRates.keySet());
    // define a format for the dates
    SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy HH:mm");
    // make the date formatter above convert from GMT to BST
    sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));

    for(UTCDate date : dateList){

      MarketDataSnapshot candleData;

      candleData = historicalRates.get(date);
      // convert the key to a Date
      Date candleDate = date.toDate();
      String sdfDate = sdf.format(candleDate);
      double open = candleData.getBidOpen();
      double low = candleData.getBidLow();
      double high = candleData.getAskHigh();     // ask is the relevant price for short positions
      double closeBid = candleData.getBidClose();// close refers to bid
      double closeAsk = candleData.getAskClose();

      Candlestick candlestick = new Candlestick(sdfDate, open, low, high, closeBid, closeAsk);

      candlesticksList.add(candlestick);
    }
}
```

In addition to adding a `closeAsk` field, we also change the `high` field price to ask instead of the bid, which makes more sense in the context of our strategy, as we will see in a bit.

Ok so now that we have made the proper adjustments to our relevant classes we can easily use the `calculateRsiForDataSet` method to calculated the RSI 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 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) {            // long position
                        if(candleSticksList.get(i).getLow()<stopLossPrice) {
                                isOpenPosition=false;            // position closed at a loss
                                lossCounter++;
                                strategyProfit-=STOP_LOSS;
                                maxDrawdown=calculateMaxDrawdown(maxProfit, strategyProfit, maxDrawdown);
                         }else if(candleSticksList.get(i).getHigh()>takeProfitPrice) {
                                isOpenPosition=false;            // position closed at a profit
                                winCounter++;
                                strategyProfit+=TAKE_PROFIT;
                                maxProfit=updateMaxProfit(strategyProfit,maxProfit);
                         }
                   }else{                                                 // short position
                        if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                isOpenPosition=false;            // position closed at a loss
                                lossCounter++;
                                strategyProfit-=STOP_LOSS;
                                maxDrawdown=calculateMaxDrawdown(maxProfit, strategyProfit, maxDrawdown);
                        }else if(candleSticksList.get(i).getLow()<takeProfitPrice) {
                                isOpenPosition=false;            // position closed at a profit
                                winCounter++;
                                strategyProfit+=TAKE_PROFIT;
                                maxProfit=updateMaxProfit(strategyProfit,maxProfit);
                        }
                    }
              }else if(candleSticksList.get(i).getRsi()<FLOOR_RSI){         // no open positions. check RSI
                        entryBuy=candleSticksList.get(i).getCloseAsk();  // use closeAsk for long position
                        stopLossPrice=((1-STOP_LOSS)*entryBuy);
                        takeProfitPrice=((1+TAKE_PROFIT)*entryBuy);
                        isOpenPosition=true;
             }else if(candleSticksList.get(i).getRsi()>CEILING_RSI){       // no open positions. check RSI
                        entrySell=candleSticksList.get(i).getCloseBid (); // use close (bid) for short position
                        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 RsiSignals 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);

}
```

Candle.java (Updated):

```
public class Candlestick{

	private String date;
	private double open;
	private double low;
	private double high;
	private double closeBid;
	private double closeAsk;
	private double rsi;

	public Candlestick(String date, double open, double low, double high, double closeBid,
			double closeAsk) {

		this.date = date;
		this.open = open;
		this.low = low;
		this.high = high;
		this.closeBid = closeBid;
		this.closeAsk = closeAsk;
	}

	public Candlestick(String date, double open, double low, double high, double closeBid) {

		this.date = date;
		this.open = open;
		this.low = low;
		this.high = high;
		this.closeBid = closeBid;
	}



				// getters and setters...



	@Override
	public String toString() {
		return "Candlestick [date=" + date + ", open=" + open + ", low=" + low + ", high=" +
			high + ", closeBid=" + closeBid + ", closeAsk=" + closeAsk + "]";
	}


}
```

RsiSignals.java:

```
import java.util.List;

public class RsiSignals implements Strategy {

  private final double STOP_LOSS;
  private final double TAKE_PROFIT;
  private final int FLOOR_RSI;
  private final int CEILING_RSI;

  public RsiSignals(double stopLoss, double takeProfit, int floorRSI) throws Exception {

        if(floorRSI<=0 || floorRSI>=100) {
                throw new Exception("Invalid RSI: "+ floorRSI +". Should be between 0-100");
        }
        this.STOP_LOSS = stopLoss;
        this.TAKE_PROFIT = takeProfit;
        this.FLOOR_RSI = floorRSI;
        this.CEILING_RSI = 100-floorRSI;
  }

                        // getters and setters goes here...

  public static void calculateRsiForDataSet(List<Candlestick> candleSticksList, int duration) {

        // duration must be smaller than the data set size
        if(duration>=candleSticksList.size()) return ;

        double sessionGainSum = 0;
        double sessionLossSum = 0;
        double sessionChange;

        for (int i = 1; i <= duration; i++) {    // calculate first RSI.

           sessionChange = candleSticksList.get(i).getCloseBid() - candleSticksList.get(i - 1).getCloseBid();

           if (sessionChange > 0) {
                sessionGainSum += sessionChange;
           }else{
                sessionLossSum += sessionChange;
           }
        }

        double averageGain = sessionGainSum / duration;
        double averageLoss = sessionLossSum / duration;
        double rs =  (averageGain / -averageLoss);
        double rsi = 100 - (100 / (1 + rs));             // first RSI.
        candleSticksList.get(duration).setRsi(rsi);

        for (int i = duration + 1; i < candleSticksList.size(); i++) {    // for smoothing

           sessionChange = candleSticksList.get(i).getCloseBid() - candleSticksList.get(i - 1).getCloseBid();

           if (sessionChange > 0) {
                averageGain = (averageGain*(duration-1) + sessionChange)/duration;
                averageLoss = (averageLoss*(duration-1) + 0)/duration;
           }else{
                averageGain = (averageGain*(duration-1) + 0)/duration;
                averageLoss = (averageLoss*(duration-1) + sessionChange)/duration;
           }

           rs = (averageGain / -averageLoss);
           rsi = 100 - (100 / (1 + rs));
           candleSticksList.get(i).setRsi(rsi);

        }
  }
```

```java
                                // RsiSignals.java continued...

@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) {              // long position
                        if(candleSticksList.get(i).getLow()<stopLossPrice) {
                                isOpenPosition=false;            // position closed at a loss
                                lossCounter++;
                                strategyProfit-=STOP_LOSS;
                                maxDrawdown=calculateMaxDrawdown(maxProfit, strategyProfit, maxDrawdown);
                         }else if(candleSticksList.get(i).getHigh()>takeProfitPrice) {
                                isOpenPosition=false;            // position closed at a profit
                                winCounter++;
                                strategyProfit+=TAKE_PROFIT;
                                maxProfit=updateMaxProfit(strategyProfit,maxProfit);
                         }
                }else{                                           // short position
                        if(candleSticksList.get(i).getHigh()>stopLossPrice) {
                                isOpenPosition=false;            // position closed at a loss
                                lossCounter++;
                                strategyProfit-=STOP_LOSS;
                                maxDrawdown=calculateMaxDrawdown(maxProfit, strategyProfit, maxDrawdown);
                        }else if(candleSticksList.get(i).getLow()<takeProfitPrice) {
                                isOpenPosition=false;            // position closed at a profit
                                winCounter++;
                                strategyProfit+=TAKE_PROFIT;
                                maxProfit=updateMaxProfit(strategyProfit,maxProfit);
                        }
                }
            }else if(candleSticksList.get(i).getRsi()<FLOOR_RSI){        // no open positions. check RSI
                        entryBuy=candleSticksList.get(i).getCloseAsk();  // use closeAsk for long position
                        stopLossPrice=((1-STOP_LOSS)*entryBuy);
                        takeProfitPrice=((1+TAKE_PROFIT)*entryBuy);
                        isOpenPosition=true;
               }else if(candleSticksList.get(i).getRsi()>CEILING_RSI){       // no open positions. check RSI
                        entrySell=candleSticksList.get(i).getCloseBid(); // use close (bid) for short position
                        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 static double updateMaxProfit(double strategyProfit, double maxProfit) {

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

                return maxProfit;
}
```

```
                                // RsiSignals.java continued...

  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;
  }
}
```

HistoryMiner.java (Updated. See “FXCM Java API – Pulling Historical Data” for more details):

```
                                        //  rest of HistoryMiner class...

                                        //  updated part:

public void convertHistoricalRatesToCandleSticks(){

    SortedSet<UTCDate> dateList = new TreeSet<>(historicalRates.keySet());
    SimpleDateFormat sdf = new SimpleDateFormat("dd.MM.yyyy HH:mm");
    sdf.setTimeZone(TimeZone.getTimeZone("Europe/London"));

    for(UTCDate date : dateList){

      MarketDataSnapshot candleData;

      candleData = historicalRates.get(date);
      // convert the key to a Date
      Date candleDate = date.toDate();
      String sdfDate = sdf.format(candleDate);
      double open = candleData.getBidOpen();
      double low = candleData.getBidLow();
      double high = candleData.getAskHigh();     // ask is the relevant price for short positions
      double closeBid = candleData.getBidClose();// close refers to bid
      double closeAsk = candleData.getAskClose();

      Candlestick candlestick = new Candlestick(sdfDate, open, low, high, closeBid, closeAsk);

      candlesticksList.add(candlestick);
    }
}

public void displayHistory() {

          if(candlesticksList.size()<1) {
                  System.out.println("No data do display");
                  return;
          }

          System.out.println("Date\t   Time\t\tOpen\tHigh\tLow\tClose");

          for (Candlestick candlestick : candlesticksList) {

                System.out.println(
                candlestick.getDate() + "\t" +
                candlestick.getOpen() + "\t" +
                candlestick.getHigh() + "\t" +
                candlestick.getLow() + "\t" +
                candlestick.getCloseBid());  // the close bid for the candle
        }
}
                                        //  end of updated part.

                                        //  rest of HistoryMiner class...
```