FXCM Java API

Pulling historical prices

By Itamar Eden

In the following tutorial I'm going to demonstrate and explain how to build an application that logs in to the FXCM Java API and pulls historical data for the purpose of backtesting a strategy.

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 market.

In this tutorial I'm going to go over each part of the application in details, step by step, in a straightforward way. This application was built to be reusable so you could easily integrate it in your own algo trading application. After finishing this tutorial you will know how to pull historical data for any asset and time period you desire, out of the ones supported by FXCM.

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.

API Review

Before we get started with building the application, the first thing we need to do is download the JAVA trading SDK from the FXCM github page and follow the instructions of how to extract the files. The relevant files for us are the fxcm-api and fxmsg jars. We need to add them to our project so we would be able to communicate with the FXCM API.

The FXCM JAVA API model is based on the FIX specification for FX. It implements the Listener design pattern. What it means is that essentially we register as listeners to receive messages that are "answers" to the requests that we send to the API. These messages can be the status of our connection to the API, the amount of collateral we have in our account, real-time streaming quotes, historical data and so on. The FXCM JAVA API contains many classes that serves different functionalities, so let's go over the important ones that are relevant for the purpose of this section - pulling historical data:

IGateway: This is the primary interface into the FXCM trading platform. It contains all the entry points into application usability.

FXCMLoginProperties: This class is used in the login method of IGateway and contains the properties necessary to log in.

IGenericMessageListener: Implementations of this interface are registered with IGateway to receive application messages.

IStatusMessageListener: Implementations of this interface are registered with IGateway to receive application status messages.

TradingSessionStatus: This class deals with the first message received after the login process is complete. Contains information regarding the trading session like available securities, status and more.

MarketDataRequest: This class is used to request market data, either real-time streaming or historical.

MarketDataSnapshot: This class is used to receive a snapshot of market data (bid, ask, expiration, etc.) for a specific asset.

We will be using each of these classes to pull historical data from the API.

Pulling Historical Data from the FXCM Java API

In the following section we are going to see how to properly log in to the FXCM API and pull historical prices. For this mission we're going to create the HistoryMiner class. This class will be responsible for pulling raw historical data from the API and organizing this data in a list of candlesticks, which you can later use for backtesting any strategy that comes into your mind.

Let's start with the basics:

```
import com.fxcm.external.api.transport.FXCMLoginProperties;
import com.fxcm.external.api.transport.GatewayFactory;
import com.fxcm.external.api.transport.IGateway;
import com.fxcm.external.api.transport.listeners.IGenericMessageListener;
import com.fxcm.external.api.transport.listeners.IStatusMessageListener;
import com.fxcm.fix.FXCMTimingIntervalFactory;
import com.fxcm.fix.IFixDefs;
import com.fxcm.fix.Instrument;
import com.fxcm.fix.SubscriptionRequestTypeFactory;
import com.fxcm.fix.UTCDate;
import com.fxcm.fix.UTCTimeOnly;
import com.fxcm.fix.UTCTimestamp;
import com.fxcm.fix.pretrade.MarketDataRequest;
import com.fxcm.fix.pretrade.MarketDataRequestReject;
import com.fxcm.fix.pretrade.MarketDataSnapshot;
import com.fxcm.fix.pretrade.TradingSessionStatus;
import com.fxcm.messaging.ISessionStatus;
import com.fxcm.messaging.ITransportable;

public class HistoryMiner implements IGenericMessageListener, IStatusMessageListener {

}
```

We start by importing multiple classes that we'll need to use later on. In addition, we must implement the `IGenericMessageListener` and `IStatusMessageListener` interfaces for login in to the API. As you can see the compiler notifies us that we must override the `messageArrived(ISessionStatus)` and `messageArrived(ITransportable)` methods. Ignore it for now as we will get to that later. The next step is to create a constructor for our class:

```
public class HistoryMiner implements IGenericMessageListener, IStatusMessageListener{

  private static final String server = "http://www.fxcorporate.com/Hosts.jsp";
  private Instrument asset;
  private UTCDate startDate;
  private UTCTimeOnly startTime;
  private FXCMLoginProperties login;

  public HistoryMiner(String username, String password, String terminal, UTCDate startDate,
                      UTCTimeOnly startTime, Instrument asset){

          this.asset=asset;
          this.startDate = startDate;
          this.startTime = startTime;

          // create a local LoginProperty
          this.login = new FXCMLoginProperties(username, password, terminal, server);

  }
```

Our constructor gets 6 arguments. The first 3 are used for the login process while the last 3 are used for defining the asset that we want the data for, as well as the start date and time of the data. The `username`, `password` and `terminal` are the credentials that you use for login in to your account. The terminal is either “Demo” or “Real”, depending on the type of your account. The API serves both types.

The `startDate` and `startTime`, when combined, represents the starting point of the historical prices. UTC stands for Universal Time Coordinated, also known as "GMT". The `UTCDate` and `UTCTimeOnly` constructors takes Java’s Date object as an argument.

The `asset` field represents the asset we pull the data for and it is of type `Instrument`. The Instrument constructor takes a String literal as an argument, for example: “EUR/USD”.

Now that we’ve set a constructor, let’s continue with the first method we are going to call for pulling historical data - the login method. As its name suggests, we will use this method to login to the FXCM API.

First, we need to add the following 2 fields:

```
private IGateway gateway;
private String currentRequest;
```

As noted above, IGateway is the primary interface into the FXCM trading platform. We’ll use the `gateway` field to register as listeners, login, logout and to send requests to the API.

The `currentRequest` field is used for differentiating between requests. Each request we send has an id which we store in `currentRequest`. Each message we receive back as a response to a request, has that request’s id. So we’ll use `currentRequest` to identify if a certain message is a response to the current request.

The login method should look like this:

```
public boolean login(IGenericMessageListener genericMessageListener, IStatusMessageListener
                     statusMessageListener){

          try{
              if(gateway == null)
                gateway = GatewayFactory.createGateway();

              gateway.registerGenericMessageListener(genericMessageListener);
              gateway.registerStatusMessageListener(statusMessageListener);

              if(!gateway.isConnected())
                gateway.login(this.login);

                currentRequest = gateway.requestTradingSessionStatus();

              // return that this process was successful
              return true;

            }catch(Exception e) {
              e.printStackTrace();
            }
            // if any error occurred, return that this process failed
            return false;
  }
```

First we check if the `gateway` field has not been defined yet. If it hasn't, we assign it a new gateway created by the factory. Then we register the `genericMessageListener` and the `statusMessageListener` with the gateway in order to receive application and status messages. Next, we check whether we are already connected to the API and if not, we attempt to login with the `login` field we initialized in the constructor. Finally, we call the `requestTradingSessionStatus` method. We must call this method after login as part of a handshake process otherwise we won't receive any messages.

After we are logged in to the FXCM API, the FXCM server start sending us messages. In order for us to receive those messages and act upon them, we must implement the `messageArrived(ISessionStatus)` and `messageArrived(ITransportable)` methods of the and `IStatusMessageListener` and `IGenericMessageListener` interfaces, respectively. These methods are the "main windows" to the FXCM API and any messages regarding status or market data would come through them.

Let's start with the implementation of the `messageArrived(ISessionStatus)` method:

```
@Override
  public void messageArrived(ISessionStatus status){

              // check the status code
            if(status.getStatusCode() == ISessionStatus.STATUSCODE_ERROR ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_DISCONNECTING ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_CONNECTING ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_CONNECTED ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_CRITICAL_ERROR ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_EXPIRED ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_LOGGINGIN ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_LOGGEDIN ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_PROCESSING ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_DISCONNECTED)
            {
              // display status message
                System.out.println("\t\t" + status.getStatusMessage());
            }
  }
```

Any message received via this method contains information regarding the status of our session (i.e. connection) with the FXCM server. In the method above we check if the status code equals any of the ones written in the condition and if it is, we print it. We use it to keep track on the session status during the login/logout process or at any time during the data mining process. In the list above we have the important status codes for connecting to and disconnecting from the API. Of course we can add or remove status codes to this list as needed.

The `messageArrived(ITransportable)` method is our main window to the FXCM server in terms of receiving market data. Each message that contains market data of financial instruments will come through this method. This method is also responsible for receiving messages which contain data on the status of our trading session, the status of our data requests and data on our account (collateral, balance, etc.).

For the sake of simplicity and readability of our code, we will separate the different functionalities of this method into different methods, each one handles a different task, depending on the type of the message received.

```
@Override
  public void messageArrived(ITransportable message){

    // decide which child function to send a cast instance of the message
    try {

      if(message instanceof MarketDataSnapshot) messageArrived((MarketDataSnapshot)message);

      if(message instanceof MarketDataRequestReject) messageArrived((MarketDataRequestReject)message);

      else if(message instanceof TradingSessionStatus) messageArrived((TradingSessionStatus)message);

    }catch(Exception e){
        e.printStackTrace();
    }

  }
```

Since the purpose of this section is historical data mining, there's no need to request data on our account (balance, collateral, open positions, etc.) and therefore we don't need to handle said data. The `messageArrived(ITransportable)` method would therefore check the type of the message and do the following:

If the `message` argument is of type `MarketDataSnapshot`, it is a response to a request we've sent for historical data and we will handle it with `messageArrived(MarketDataSnapshot message)` .

If the `message` argument is of type `MarketDataRequestReject`, it means a request we've sent for historical data got rejected and we will handle it with `messageArrived(MarketDataRequestReject message)` .

If the `message` argument is of type `TradingSessionStatus`, it is a response to a request for trading session status (usually after login) and we will handle it with `messageArrived(TradingSessionStatus message)` .

When declaring methods with the same name but different signatures we must use casting to let the compiler know which method to call. Alternatively, we can use different names for each method.

The next step is to create each of this methods to handle the messages appropriately. we will start with the `messageArrived(TradingSessionStatus tss)`:

As part of a handshake process, we must call the `IGateWay`'s `requestTradingSessionStatus` method after we login, otherwise we will not receive trading session messages. After calling this method, the first message we'll receive back from the API will be of type `TradingSessionStatus`.

```
public void messageArrived(TradingSessionStatus tss){

      if(currentRequest.equals(tss.getRequestID())){

              MarketDataRequest mdr = new MarketDataRequest();
              // set the subscription type to ask for only a snapshot of the history
              mdr.setSubscriptionRequestType(SubscriptionRequestTypeFactory.SNAPSHOT);
              // request the response to be formated FXCM style
              mdr.setResponseFormat(IFixDefs.MSGTYPE_FXCMRESPONSE);
              // set the interval of the data candles
              mdr.setFXCMTimingInterval(FXCMTimingIntervalFactory.MIN1);
              mdr.setFXCMStartDate(startDate);
              mdr.setFXCMStartTime(startTime);
              mdr.addRelatedSymbol(asset);

              currentRequest=sendRequest(mdr);
      }

  }
```

As this is the first method that's get called after we are logged in, we will use it to fire the first request for historical data. The FXCM API sends back a batch of up to 300 candles of historical data for each request. If we want more than 300 (and in most cases we will), we need to send multiple requests. We will send the subsequent requests in a different method we will see in a bit.

To send a request for historical data we must create an instance of `MarketDataRequest` and set the characteristics of the request appropriately. We set the `SubscriptionRequestType` to ***`SNAPSHOT`*** as we are looking for historical data. Each snapshot is in fact a candlestick. The alternative is ***`SUBSCRIBE`*** which is streaming current prices. The response format should always be set to ***`MSGTYPE_FXCMRESPONSE`***. The `FXCMTimingIntervalFactory` indicates the duration of each candle. The duration can be as short as a tick or 10 seconds and as long as a week or a month. In this tutorial we'll set the time interval to ***`MIN1,`*** which means each candle (snapshot) has a duration of 1 minute.

We also need to set the starting date and time, and the instrument we want the data for. remember that we are passing this data to the constructor when we instantiate the HistoryMiner class, so it's already stored in the appropriate fields.

After we've set all the relevant characteristics of the request we can send it to the FXCM API with the `sendRequest` method:

```
public String sendRequest(MarketDataRequest request){

                try{

                // send the request message to the API.
                currentRequest = gateway.sendMessage(request);

                // return the request id for authentication when messages would arrive from the API.
                return currentRequest;

                }catch(Exception e) {
                        e.printStackTrace();
                }
                // if an error occurred, return no result
                return null;
  }
```

After we send a request for a market data snapshot, the FXCM server generates a response in the form of a `MarketDataSnapshot` class. This response contains a candlestick data (open, low, high, close etc.) for our designated asset. Since we are requesting historical data, we are in essence asking for multiple candlesticks where each candlestick is a market data snapshot. As mentioned above, the FXCM API limits each request to 300 responses (i.e. each response is a candlestick), which means that if we want our data to consist of more than 300 candlesticks, we need to send multiple requests where each request will generate 300 responses of `MarketDataSnapshot` class.

Testing a strategy with short candlestick duration (for example: 1-minute candlesticks) requires much more than 300 candlesticks, as many trading patterns tend to last for days, weeks or even months. It is considered good practice to use at least 1 year of data for backtesting a strategy in a credible way. To achieve this, we are going to send multiple requests in the following way: for each batch of 300 responses that contain 300 candlesticks, we will save each candlestick in a list. After the batch is over, we will generate a new request for another batch of 300 candlesticks, and we will continue to do so until we will have the full amount of candlesticks we need (1 year of 1-minute data).

Let's start by adding the following field:

```
private final HashMap<UTCDate, MarketDataSnapshot> historicalRates = new HashMap<>();
private int dataCounter=0;
private UTCTimestamp openTimestamp;
boolean stillMining=true;
```

We are going to store each candlestick in a hashmap called `historicalRates`, where the key is a date and the value is an instance of `MarketDataSnapshot`. Using a hashmap with dates as keys would later enable us to sort the data based on the date and make sure our candlesticks are ordered in a chronological order.

The next step is to create the method to handle messages that contain the market data candlesticks - `messageArrived(MarketDataSnapshot mds)`.

```
public void messageArrived(MarketDataSnapshot mds){

        if(mds.getRequestID() != null && mds.getRequestID().equals(currentRequest))
           historicalRates.put(mds.getDate(), mds);


        if (mds.getRequestID() != null){

            dataCounter++;

            if (openTimestamp == null){
                //get the time stamp of first candle of this batch. use it to set the end time of next batch.
                openTimestamp = mds.getOpenTimestamp();
                System.out.println("first\t= " + mds.getOpenTimestamp() + " = " + mds.getRequestID());
            }

            if (mds.getFXCMContinuousFlag() == IFixDefs.FXCMCONTINUOUS_END){      // end of packet.
                System.out.println("last\t= " + mds.getOpenTimestamp() + " = " + mds.getRequestID()+"\n");

                MarketDataRequest mdr = new MarketDataRequest();
                    mdr.setSubscriptionRequestType(SubscriptionRequestTypeFactory.SNAPSHOT);
                    mdr.setResponseFormat(IFixDefs.MSGTYPE_FXCMRESPONSE);
                    mdr.setFXCMTimingInterval(FXCMTimingIntervalFactory.MIN1);
                    mdr.setFXCMStartDate(startDate);
                    mdr.setFXCMStartTime(startTime);
                    mdr.setFXCMEndDate(new UTCDate(openTimestamp));
                    mdr.setFXCMEndTime(new UTCTimeOnly(openTimestamp));
                    mdr.addRelatedSymbol(asset);

                    System.out.println("FXCMStartDate\t= " + mdr.getFXCMStartDate());
                    System.out.println("FXCMStartTime\t= " + mdr.getFXCMStartTime());
                    System.out.println("FXCMEndDate\t\t= " + mdr.getFXCMEndDate());
                    System.out.println("FXCMEndTime\t\t= " + mdr.getFXCMEndTime());
                    System.out.println("----------------------\ntotal mds received = " + dataCounter+"\n");

                    if(!(mds.getOpenTimestamp().equals(openTimestamp))) {
                        // send another request for historical data
                        currentRequest=sendRequest(mdr);
                        openTimestamp = null;

                    }else {
                        stillMining=false;
                        System.out.println("mining over...");
                    }

            }
        }

    }
```

The first part of the method checks whether the market data snapshot is part of the response to the last request we've sent. if it is, we add it to the `historicalRates` hashmap.

The second part of the method is responsible for generating another market data request for another batch of candles, if needed. We do that by capturing the opening timestamp of the current batch in the `openTimestamp` field, and using it to set the ending time of the next batch. That way the next batch's first candle will be 300 candles prior to the first candle of the current batch. For setting the ending date and time we use `setFXCMEndDate` and `setFXCMEndTime` methods respectively. If we don't set an ending time in the market data request (like in the `messageArrived(TradingSessionStatus tss)` method), the FXCM Java API automatically sets the ending time to the moment we generated the request.

Please note that we need to generate the request for the next batch of candlesticks only *after* the current batch is over. We can use the `getFXCMContinuousFlag` method of the `MarketDataSnapshot` class for exactly that purpose. This method can tell us whether the current message is the last one in this batch of messages and then we're free to send the next request.

Ok let's do a short recap of the mechanism we've set so far for pulling historical data: after logging in to the Java FXCM API we must call `IGateWay`'s `requestTradingSessionStatus` method to receive messages. The first message we'll receive thereafter would be of type `TradingSessionStatus` and so we'll handle it via the `messageArrived(TradingSessionStatus tss)` method. In this method we generate the first market data request. The response to this request comes in the form of a `MarketDataSnapshot` class via the `messageArrived(MarketDataSnapshot mds)` method. We use this method to process the response and send a subsequent request for another batch of historical data. The next batch of data would also come through this method, we will process it and send another request. This loop should continue until we have all the historical data we need. Once we have all the data we stop this loop. The last part of the `messageArrived(MarketDataSnapshot mds)` method is responsible for exactly that.

In the last part we check whether the timestamp of the last candlestick of the current batch is equal to the timestamp of the first candlestick of the current batch. If it is, it means that the last batch consist of only one candlestick. It also means that now we have all the historical data that we need. The logic behind this condition is a bit hard to explain but in short, it deals with the fact that the end time of each request we send is rounded to the nearest time interval we've set (minute, hour etc.) but the `startTime` field isn't. this small difference can result in an endless loop where we keep receiving the last candlestick of the entire data set over and over again. By using the condition above we eliminate this problem.

As you can see we use a boolean `stillMining` field as a flag that indicates whether we should keep requesting historical data from the API. We will see its use when we will build the main method that runs the data mining process.

When sending a request for market data, some errors might occur. For example: sending a request in which the ending date is prior to the starting date, typo in the asset's name and so on. When a request does in fact contains an error, the FXCM server sends back a response of type `MarketDataRequestReject`. To handle this response, we'll create the following method:

```
public void messageArrived(MarketDataRequestReject mdrr){

        System.out.println("Historical data rejected; " + mdrr.getMDReqRejReason());

        stillMining=false;

}
```

The method displays a note consisting of the reason the request was rejected and stops the mining process by assigning a false value to the `stillMining` flag.

Once we're done with data mining and have all the historical data stored in a list, there is no need to stay connected to the Java API. We should logout so we would stop receiving any messages. Let's create a logout method to do exactly that:

```
public void logout(IGenericMessageListener genericMessageListener, IStatusMessageListener
                   statusMessageListener){

           // attempt to logout of the api
           gateway.logout();
           // remove the generic message listener, stop listening to updates
           gateway.removeGenericMessageListener(genericMessageListener);
           // remove the status message listener, stop listening to status changes
           gateway.removeStatusMessageListener(statusMessageListener);
  }
```

Ok so after the process of receiving historical rates from the FXCM API ends, we have the data stored in the `historicalRates` hashmap with dates as keys and `MarketDataSnapshot` instances as values. That is far from ideal as we can't really use it in a flexible and straightforward way. What we should have is a list of instances of a Candlestick object that stores each data type of the `MarketDataSnapshot` class in a relevant field. To achieve this, we should first create a Candlestick object and then create a method that converts the data stored in the `historicalRates` hashmap into a list of Candlestick instances.

We'll start with the Candlestick class (see below). This class is rather basic and contains all the characteristics of a candlestick. Please note that each candle's descriptor (i.e. open, low, high, close) can in fact have two prices – bid and ask. In this example we only use one field for each descriptor for the sake of simplicity, but if for example we need both the bid and the ask for the candle's closing price for a certain strategy, we can easily change the `close` field to `closeBid` and add a `closeAsk` field to capture both the bid and the ask.

```
public class CandleStick {

	private String date;
	private double open;
	private double low;
	private double high;
	private double close;

	public CandleStick(String date, double open, double low, double high, double close) {

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


	// getters and setters goes here...


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

}
```

Let's move on with the method to convert the data into Candlesticks. First we add a `candlesticksList` field to the HistoryMiner class. this list will hold the final data after converted to candlesticks:

```
   List <Candlestick> candlesticksList = new ArrayList<Candlestick>();
```

In this method (see below) we first sort all the keys in the `historicalRates` hashmap and then use a *for* loop to go through each date and assign the data stored in a `MarketDataSnapshot` class into the relevant Candlestick field. We use Java's TreeSet class for the implementation of the SortedSet interface. We need to use a set to make sure that each date is unique and we don't have any duplicates. We need to have it sorted in a chronological order for the backtesting process to make sense.

We use Java's SimpleDateFormat class to set the date format and the time zone we want for our data set. For a list of available time zone ids in Java's TimeZone class use the `TimeZone.getAvailableIDs()` method.

```
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, closeBid, closeAsk);

      candlesticksList.add(candlestick);
    }

  }
```

The last method in the HistoryMiner class will be responsible for displaying the historical data:

```
public void displayHistory() {

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

          // give the table column headings
          System.out.println("Date\t   Time\t\tOpen\tHigh\tLow\tClose");

          for (Candlestick candlestick : candlesticksArr) {

                System.out.println(
                candlestick.getDate() + "\t" +
                candlestick.getOpen() + "\t" +   // the open bid for the candle
                candlestick.getHigh() + "\t" +   // the high bid for the candle
                candlestick.getLow() + "\t" +    // the low bid for the candle
                candlestick.getClose ());        // the close bid for the candle
        }

  }
```

Ok so after we've seen how to connect to the FXCM JAVA API and pull historical prices via the HistoryMiner class, the only thing left to do is to test it. For this purpose, let's create the following Tester class that will pull historical data for the EUR/USD pair for the past year and print it to the console:

```
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Collections;
import java.util.List;

import com.fxcm.fix.Instrument;
import com.fxcm.fix.UTCDate;
import com.fxcm.fix.UTCTimeOnly;

public class Tester{

   public static void main(String[] args){

          List <Candlestick> candleSticksList;
          UTCDate startDate;
          UTCTimeOnly startTime;
          Instrument asset = new Instrument("EUR/USD");

          // get the current time and roll back 1 year
          Calendar instance = Calendar.getInstance();
          instance.roll(Calendar.YEAR, -1);

          // set the starting date and time of the historical data
          startDate = new UTCDate(instance.getTime());
          startTime = new UTCTimeOnly(instance.getTime());


          try{

            // create an instance of the JavaFixHistoryMiner
            HistoryMiner miner = new HistoryMiner("your user name", "your password", "Demo", startDate,
                                                  startTime, asset);

            // login to the api
            miner.login(miner,miner);

            // keep mining for historical data before logging out
            while(miner.stillMining) {
                Thread.sleep(1000);
            }

            Thread.sleep(1000);

            // log out of the api
            miner.logout(miner,miner);

            // convert rates to candlesticks
            miner.convertHistoricalRatesToCandleSticks();

            // display the collected rates
            miner.displayHistory();

          }catch (Exception e) {
            e.printStackTrace();
          }

   }

}
```

Key notes for the Tester class

In the main method we start with creating an instance of the Calendar class to take advantage of the powerful roll method. We then use this instance to set the `startDate` and `startTime` variables before passing them to the HistoryMiner constructor along with our account credentials and the instrument we want the data for. after constructing a HistoryMiner instance, we call the login method and assign the HistoryMiner instance itself as listeners.

For the strategy testing process to be successful, we need it to be synchronous. We first need to get all the historical data from the FXCM server and *only* then print it to the console. We can achieve this by using a *while* loop with the `stillMining` flag as the exit condition. As long as the program is still mining for historical data, it will stay in the loop and only after that process is complete, it would move forward to display the data.

After we obtain all the data that we need, we logout, convert our raw data into candlesticks, and print those candlesticks into the console.

The following results were derived from testing the HistoryMiner class with 1-minute historical data for the EUD/USD pair, ranging from October 11, 2016 until October 11, 2017:

```
Date      Time      Open         High         Low          Close
11.10.2016 10:28    1.11097      1.11125      1.11093      1.11093
11.10.2016 10:29    1.11093      1.11131      1.11093      1.11106
11.10.2016 10:30    1.11106      1.1113       1.11088      1.11098
11.10.2016 10:31    1.11098      1.1113       1.11096      1.11107
11.10.2016 10:32    1.11107      1.11133      1.11092      1.11092
11.10.2016 10:33    1.11092      1.11121      1.11088      1.11097
11.10.2016 10:34    1.11097      1.11132      1.11092      1.11097
11.10.2016 10:35    1.11097      1.11125      1.11084      1.1109
11.10.2016 10:36    1.1109       1.11114      1.11079      1.11079
11.10.2016 10:37    1.11079      1.11111      1.11076      1.11082
11.10.2016 10:38    1.11082      1.11107      1.11079      1.11081
11.10.2016 10:39    1.11081      1.11116      1.11076      1.11076
11.10.2016 10:40    1.11076      1.11111      1.11072      1.11087
11.10.2016 10:41    1.11087      1.11115      1.11084      1.11087
11.10.2016 10:42    1.11087      1.11125      1.11079      1.11101
11.10.2016 10:43    1.11101      1.11127      1.11088      1.11088
11.10.2016 10:44    1.11088      1.11115      1.11081      1.11088
11.10.2016 10:45    1.11088      1.11111      1.11078      1.11083
11.10.2016 10:54    1.11073      1.11107      1.11065      1.11081
11.10.2016 11:05    1.11079      1.11103      1.11056      1.11059
.
.
.
.
11.10.2017 10:25    1.18197      1.18224      1.18191      1.18194
11.10.2017 10:26    1.18194      1.18217      1.18188      1.1819
11.10.2017 10:27    1.1819       1.18214      1.18186      1.18188
```

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

HistoryMiner.java:

```
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.SortedSet;
import java.util.TimeZone;
import java.util.TreeSet;

import com.fxcm.external.api.transport.FXCMLoginProperties;
import com.fxcm.external.api.transport.GatewayFactory;
import com.fxcm.external.api.transport.IGateway;
import com.fxcm.external.api.transport.listeners.IGenericMessageListener;
import com.fxcm.external.api.transport.listeners.IStatusMessageListener;
import com.fxcm.fix.FXCMTimingIntervalFactory;
import com.fxcm.fix.IFixDefs;
import com.fxcm.fix.Instrument;
import com.fxcm.fix.NotDefinedException;
import com.fxcm.fix.SubscriptionRequestTypeFactory;
import com.fxcm.fix.UTCDate;
import com.fxcm.fix.UTCTimeOnly;
import com.fxcm.fix.UTCTimestamp;
import com.fxcm.fix.pretrade.MarketDataRequest;
import com.fxcm.fix.pretrade.MarketDataRequestReject;
import com.fxcm.fix.pretrade.MarketDataSnapshot;
import com.fxcm.fix.pretrade.TradingSessionStatus;
import com.fxcm.messaging.ISessionStatus;
import com.fxcm.messaging.ITransportable;


private static final String server = "http://www.fxcorporate.com/Hosts.jsp";
private Instrument asset;
private FXCMLoginProperties login;
private IGateway gateway;
private String currentRequest;
private final HashMap<UTCDate, MarketDataSnapshot> historicalRates = new HashMap<>();
private int dataCounter=0;
List <Candlestick> candlesticksList = new ArrayList<Candlestick>();
private UTCDate startDate;
private UTCTimeOnly startTime;
private UTCTimestamp openTimestamp;
boolean stillMining=true;


public class HistoryMiner implements IGenericMessageListener, IStatusMessageListener{

  private static final String server = "http://www.fxcorporate.com/Hosts.jsp";
  private Instrument asset;
  private UTCDate startDate;
  private UTCTimeOnly startTime;
  private FXCMLoginProperties login;

  public HistoryMiner(String username, String password, String terminal, UTCDate startDate,
              UTCTimeOnly startTime, Instrument asset){

          this.asset=asset;
          this.startDate = startDate;
          this.startTime = startTime;

          // create a local LoginProperty
          this.login = new FXCMLoginProperties(username, password, terminal, server);

  }
```

```
                        // HistoryMIner.java continued...

public boolean login(IGenericMessageListener genericMessageListener, IStatusMessageListener
                     statusMessageListener){

          try{
              if(gateway == null)
                gateway = GatewayFactory.createGateway();

              gateway.registerGenericMessageListener(genericMessageListener);
              gateway.registerStatusMessageListener(statusMessageListener);

              if(!gateway.isConnected())
                gateway.login(this.login);

                currentRequest = gateway.requestTradingSessionStatus();

              // return that this process was successful
              return true;

            }catch(Exception e) {
              e.printStackTrace();
            }
            // if any error occurred, then return that this process failed
            return false;
  }


@Override
public void messageArrived(ISessionStatus status){

                // check the status code
            if(status.getStatusCode() == ISessionStatus.STATUSCODE_ERROR ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_DISCONNECTING ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_CONNECTING ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_CONNECTED ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_CRITICAL_ERROR ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_EXPIRED ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_LOGGINGIN ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_LOGGEDIN ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_PROCESSING ||
               status.getStatusCode() == ISessionStatus.STATUSCODE_DISCONNECTED)
            {
              // display status message
                System.out.println("\t\t" + status.getStatusMessage());
            }
 }


@Override
public void messageArrived(ITransportable message){

    // decide which child function to send a cast instance of the message
    try {

      if(message instanceof MarketDataSnapshot) messageArrived((MarketDataSnapshot)message);

      if(message instanceof MarketDataRequestReject) messageArrived((MarketDataRequestReject)message);

      else if(message instanceof TradingSessionStatus) messageArrived((TradingSessionStatus)message);

    }catch(Exception e){
        e.printStackTrace();
    }

 }
```

```
                        // HistoryMIner.java continued...


public void messageArrived(TradingSessionStatus tss){

      if(currentRequest.equals(tss.getRequestID())){

              MarketDataRequest mdr = new MarketDataRequest();
              // set the subscription type to ask for only a snapshot of the history
              mdr.setSubscriptionRequestType(SubscriptionRequestTypeFactory.SNAPSHOT);
              // request the response to be formated FXCM style
              mdr.setResponseFormat(IFixDefs.MSGTYPE_FXCMRESPONSE);
              // set the interval of the data candles
              mdr.setFXCMTimingInterval(FXCMTimingIntervalFactory.MIN1);
              mdr.setFXCMStartDate(startDate);
              mdr.setFXCMStartTime(startTime);
              mdr.addRelatedSymbol(asset);

              // send request for historical data
              currentRequest=sendRequest(mdr);
      }

  }


public String sendRequest(MarketDataRequest request){

                try{

                // send the request message to the API.
                currentRequest = gateway.sendMessage(request);

                // return the request id for authentication when messages would arrive from the API.
                return currentRequest;

                }catch(Exception e) {
                        e.printStackTrace();
                }
                // if an error occurred, return no result
                return null;
  }



public void logout(IGenericMessageListener genericMessageListener, IStatusMessageListener
                    statusMessageListener){

           // attempt to logout of the api
           gateway.logout();
           // remove the generic message listener, stop listening to updates
           gateway.removeGenericMessageListener(genericMessageListener);
           // remove the status message listener, stop listening to status changes
           gateway.removeStatusMessageListener(statusMessageListener);
  }
```

```
                        // HistoryMIner.java continued...


public void messageArrived(MarketDataSnapshot mds){

        if(mds.getRequestID() != null && mds.getRequestID().equals(currentRequest))
           historicalRates.put(mds.getDate(), mds);


        if (mds.getRequestID() != null){

            dataCounter++;

            if (openTimestamp == null){
                //get the time stamp of first candle of this batch. use it to set the end time of next
batch.
                openTimestamp = mds.getOpenTimestamp();
                System.out.println("first\t= " + mds.getOpenTimestamp() + " = " + mds.getRequestID());
            }

            if (mds.getFXCMContinuousFlag() == IFixDefs.FXCMCONTINUOUS_END){      // end of packet.
                System.out.println("last\t= " + mds.getOpenTimestamp() + " = " +
mds.getRequestID()+"\n");

                MarketDataRequest mdr = new MarketDataRequest();
                    mdr.setSubscriptionRequestType(SubscriptionRequestTypeFactory.SNAPSHOT);
                    mdr.setResponseFormat(IFixDefs.MSGTYPE_FXCMRESPONSE);
                    mdr.setFXCMTimingInterval(FXCMTimingIntervalFactory.MIN1);
                    mdr.setFXCMStartDate(startDate);
                    mdr.setFXCMStartTime(startTime);
                    mdr.setFXCMEndDate(new UTCDate(openTimestamp));
                    mdr.setFXCMEndTime(new UTCTimeOnly(openTimestamp));
                    mdr.addRelatedSymbol(asset);

                    System.out.println("FXCMStartDate\t= " + mdr.getFXCMStartDate());
                    System.out.println("FXCMStartTime\t= " + mdr.getFXCMStartTime());
                    System.out.println("FXCMEndDate\t\t= " + mdr.getFXCMEndDate());
                    System.out.println("FXCMEndTime\t\t= " + mdr.getFXCMEndTime());
                    System.out.println("----------------------\ntotal mds received = " +
dataCounter+"\n");

                    if(!(mds.getOpenTimestamp().equals(openTimestamp))) {
                        // send another request for historical data
                        currentRequest=sendRequest(mdr);
                        openTimestamp = null;

                    }else {
                        stillMining=false;
                        System.out.println("mining over...");
                    }

            }
        }

    }

public void messageArrived(MarketDataRequestReject mdrr){

        System.out.println("Historical data rejected; " + mdrr.getMDReqRejReason());

        stillMining=false;

}
```

```
                        // HistoryMIner.java continued...


public void messageArrived(MarketDataSnapshot mds){

        if(mds.getRequestID() != null && mds.getRequestID().equals(currentRequest))
           historicalRates.put(mds.getDate(), mds);


        if (mds.getRequestID() != null){

            dataCounter++;

            if (openTimestamp == null){
                //get the time stamp of first candle of this batch. use it to set the end time of next
                // batch.
                openTimestamp = mds.getOpenTimestamp();
                System.out.println("first\t= " + mds.getOpenTimestamp() + " = " + mds.getRequestID());
            }

            if (mds.getFXCMContinuousFlag() == IFixDefs.FXCMCONTINUOUS_END){      // end of packet.
                System.out.println("last\t= " + mds.getOpenTimestamp() + " = " +
mds.getRequestID()+"\n");

                MarketDataRequest mdr = new MarketDataRequest();
                    mdr.setSubscriptionRequestType(SubscriptionRequestTypeFactory.SNAPSHOT);
                    mdr.setResponseFormat(IFixDefs.MSGTYPE_FXCMRESPONSE);
                    mdr.setFXCMTimingInterval(FXCMTimingIntervalFactory.MIN1);
                    mdr.setFXCMStartDate(startDate);
                    mdr.setFXCMStartTime(startTime);
                    mdr.setFXCMEndDate(new UTCDate(openTimestamp));
                    mdr.setFXCMEndTime(new UTCTimeOnly(openTimestamp));
                    mdr.addRelatedSymbol(asset);

                    System.out.println("FXCMStartDate\t= " + mdr.getFXCMStartDate());
                    System.out.println("FXCMStartTime\t= " + mdr.getFXCMStartTime());
                    System.out.println("FXCMEndDate\t\t= " + mdr.getFXCMEndDate());
                    System.out.println("FXCMEndTime\t\t= " + mdr.getFXCMEndTime());
                    System.out.println("----------------------\ntotal mds received = " +
dataCounter+"\n");

                    if(!(mds.getOpenTimestamp().equals(openTimestamp))) {
                        // send another request for historical data
                        currentRequest=sendRequest(mdr);
                        openTimestamp = null;

                    }else {
                        stillMining=false;
                        System.out.println("mining over...");
                    }

            }
        }

    }

public void messageArrived(MarketDataRequestReject mdrr){

        System.out.println("Historical data rejected; " + mdrr.getMDReqRejReason());

        stillMining=false;

}
```

```
                        // HistoryMIner.java continued...


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);
    }

  }


public void displayHistory() {

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

          // give the table column headings
          System.out.println("Date\t   Time\t\tOpen\tHigh\tLow\tClose");

          for (Candlestick candlestick : candlesticksArr) {

                System.out.println(
                candlestick.getDate() + "\t" +
                candlestick.getOpen() + "\t" +   // the open bid for the candle
                candlestick.getHigh() + "\t" +   // the high bid for the candle
                candlestick.getLow() + "\t" +    // the low bid for the candle
                candlestick.getClose ());        // the close bid for the candle
        }

  }
```

CandleStick.java:

```
public class CandleStick {

	private String date;
	private double open;
	private double low;
	private double high;
	private double close;

	public CandleStick(String date, double open, double low, double high, double close) {

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


	// getters and setters goes here...


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

}
```

Tester.java:

```
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Collections;
import java.util.List;
import com.fxcm.fix.Instrument;
import com.fxcm.fix.UTCDate;
import com.fxcm.fix.UTCTimeOnly;

public class Tester{

   public static void main(String[] args){

          List <Candlestick> candleSticksList;
          UTCDate startDate;
          UTCTimeOnly startTime;
          Instrument asset = new Instrument("EUR/USD");

          // get the current time and roll back 1 year
          Calendar instance = Calendar.getInstance();
          instance.roll(Calendar.YEAR, -1);

          // set the starting date and time of the historical data
          startDate = new UTCDate(instance.getTime());
          startTime = new UTCTimeOnly(instance.getTime());

          try{

            // create an instance of the JavaFixHistoryMiner
            HistoryMiner miner = new HistoryMiner("your user name", "your password", "Demo", startDate,
                                                  startTime, asset);

            // login to the api
            miner.login(miner,miner);

            // keep mining for historical data before logging out
            while(miner.stillMining) {
                Thread.sleep(1000);
            }

            Thread.sleep(1000);

            // log out of the api
            miner.logout(miner,miner);

            // convert rates to candlesticks
            miner.convertHistoricalRatesToCandleSticks();

            // display the collected rates
            miner.displayHistory();

          }catch (Exception e) {
            e.printStackTrace();
          }

   }

}
```