Criterion C: Development

My first step was to start testing the two API wrappers I would be using to get data from the two exchanges. Later on in the project, for the sake of simplicity, I moved from the two APIs to one unified wrapper called CCXT, but for now I will discuss the two API wrappers I initially used.

In an effort to avoid writing my own wrappers which would have wasted my limited time, I found two existing wrappers on GitHub for my two exchanges.

```
const Bithumb = require("bithumb.js")
const bithumb = new Bithumb("REDACTED", "REDACTED");
const binance = require("node-binance-api");
const request = require("request");
binance.options({
  "APIKEY":"REDACTED",
  "APISECRET":"REDACTED"
});
```

bithumb.js is the wrapper for the Korean side, node-binance-api is the wrapper for the Hong Kong side

The one for the exchange on the Hong Kong side was extremely well documented and was written by one of the main maintainers of Node.js. On the other hand, the wrapper for the Korean exchange was not well documented, but I successfully got it working for the initial steps.

My first step was simply getting my data collection function for the two exchanges working. This was my first introduction to something I had never seen before, the async/await structure from the new ECMA standards. Initially I did not understand it, but I eventually grew to love it since it would allow me to wait for a trade to be fully executed before the next function in the loop ran. The entire data gathering function was an async with the promises being the data from the Korean exchange.

```
let koreanXRPBBuy, koreanXRPSell, usXRP, koreanETHBuy, koreanETHSell, usETH, USDtoBTC, usETHBalance, usXRPBalance, koreanETHBalance,
async function koreanPrices() {
  binance.prices((error, ticker) => {
    USDtoBTC = parseInt(ticker.BTCUSDT);
    usB = parseFloat(ticker.XRPBTC) * USDtoBTC;
    usETH = parseFloat(ticker.ETHBTC) * USDtoBTC;
  });
  binance.useServerTime(function() {
    binance.balance((error, balances) => {
      usETHBalance = balances.ETH.available;``
      usXRPBalance = balances.XRP.available;
    });
  })
  try {
    let promises = [bithumb.getTicker("XRP"), bithumb.getTicker("ETH"), bithumb.getBalance("XRP"), bithumb.getBalance("ETH")]
    let prices = await Promise.all(promises);
    koreanXRPBuyRaw = parseInt((prices[0]).data.buy_price);
    koreanETHBuyRaw = parseInt((prices[1]).data.buy_price);
    koreanETHSellRaw = parseInt((prices[1]).data.sell_price);
    koreanXRPSellRaw = parseInt((prices[0]).data.sell_price);
    koreanXRPBalance = parseFloat(prices[2].data.available_xrp);
    koreanETHBalance = parseFloat(prices[3].data.available_eth);
  } catch(err) {
    console.log("Korean Data not working")
  }
}
```

This screenshot shows the structure of my data collection function, along with the majority of my variables declared above it. When the async ran, it would wait for the promises to be fulfilled by the Korean data before running the whole function.

After finalizing my data collection, I move on to making a ready function that would convert the Korean prices to USD before supplying them to the spread checker as inputs.

```
let koreanXRPETH, koreanETHXRP
async function ready() {
  try {
    let promises = [koreanPrices(), getConversion()];
    await Promise.all(promises);
    koreanXRPBuy = koreanXRPBuyRaw * krwTousd;
    koreanETHBuy = koreanETHBuyRaw * krwTousd;
    koreanXRPSell = koreanXRPSellRaw * krwTousd;
    koreanETHSell = koreanETHSellRaw * krwTousd;
    koreanXRPETH = koreanETHBuyRaw/koreanXRPSellRaw;
    koreanETHXRP = koreanXRPBuyRaw/koreanETHSellRaw;
    checkSpread(koreanXRPBuy, koreanETHBuy, koreanETHSell, koreanXRPSell);
    setTimeout(ETHtoXRPKorea, 5000);
  } catch(err) {
    console.log("Korean prices or conversion rate not applying correctly ", err)
  }
}
```

The above screenshot shows the structure of my prices function which finalizes the prices from the initial data gathering function.

Once I had established the prices it was time to use them to check the spreads. In order to do this I wrote a function that divides the price of the initial Korean currency by the price for the same currency on the Hong Kong side. This was then divided by the price of the final Korean currency by the price of the same currency on the Hong Kong side. After doing this I wrote two methods in the same function to check whether either of the spreads was greater than 0.7%. If it was, these methods would add the spread to the array of good spreads and initialize the trade loop.

After initially building the trading functions around the two APIs I had settled on using at first turned out to be a huge pain. After doing some more research, I decided to change from using two separate API's to the CCXT unified API. Doing this would allow for me to simplify my code, and it helps to consolidate my data as well as offering buy and sell prices for both exchanges. With more accurate data I would be able to much more accurate calculate spreads in order to maximise the potential returns.

The hardest part of writing my final version was finding a way to keep each function from running until the balance was confirmed to have been updated. To do this I used async/await structures, with one function to check each balance, with a boolean as input to see which direction the code is trading.

```
function koreanXRPCheck(which) {
  console.log(arguments.callee.name + ' called')
  if(koreanXRPBalance*usXRPB > 10) {
    if(which) {
      setTimeout(XrptoEthKorea, 1000);
    } else {
      setTimeout(XrpKoreatoUS, 1000);
    }
  } else {
    setTimeout(function() {
      koreanXRPCheck(which);
    }, 1000);
  }
}
```

Above is an example of one of the functions. As you can see, it checks if the balance is worth more than $10. If it is above $10, it checks the value of which to and runs the next function depending on which direction the trading is going. If it is less than $10, it waits 1 second before trying again.

Finally after building out the balance checking functions, I finish up the trading functions. Two of the functions were written with initial checks for balance to make sure that I am never trading more than $5000 at a time. I did this because my initial math indicated that market depth could not tolerate more. This was the final step and at this point I felt that I was finished with my code.