# Sonar Trading Programming Challenge

This is intended to be a full-stack representation of the types of programming challenges all of our team must be able to solve. We hope that this challenge 1) gives you a good glimpse into the type of engineering we're doing at Sonar Trading, and 2) helps us understand your software engineering abilities. Following the instructions below, you will have to write a Java 8 application that connects to the Bitso cryptocurrency exchange, maintains state between the application and Bitso, and simulates trade execution following a simple trading strategy.

In order to simplify the problem, only public APIs need to be used. The application will also be completely passive, collecting real information from the exchange but simulating orders locally only.

**There is no time limit to complete this challenge.** You should submit your work once you have completed all the steps and it has reached a point where you believe the code is high quality and representative of your ability. We will consider quality and timeliness of the submission in our evaluation.

**You are prohibited from using any third-party code made specifically for Bitso integration (for example github.com/bitsoex/bitso-java).** You may utilize any third-party libraries you'd like to manage connection level details, including WebSocket, REST requests, JSON parsing, UI themes, etc. Only direct wrappers of the Bitso API are prohibited.

This exercise is intended to test your Java skills specifically. All code for this submission should be in Java 8, with the exception of Gradle, Maven, and other build tools.

We utilize JavaFX for our frontend but **you may use anything you like to build the UI** (HTML, Javascript, etc). Remember that the goal of this exercise is to evaluate your Java skills. **Although front-end logic in other languages is allowed, all core implementation should be done in Java and outside the UI.**

Complete the steps below in any order that feels most comfortable for you:

1. Create a private Git repository (and share it with us) on the platform of your choice (Github, Bitbucket, etc).
2. Read about Bitso's public REST API and WebSocket API. We will use the BTCMXN order book for all parts of this exercise.
3. Follow the instructions detailed on the WebSocket API's General page to maintain real-time orderbook state through coordination between REST and websocket. **The only channel you should connect to is the 'diff-orders' channel. Do not connect to any**

**others for any other reason.** Pay special attention to this excerpt of the relevant instructions:

> In theory, you can get a copy of the full order book via REST once, and keep it up to date by using the diff-orders channel with the following algorithm:
>
> - Subscribe to the diff-orders channel.
> - Queue any message that come in to this channel.
> - Get the full orderbook from the REST orderbook endpoint.
> - Playback the queued message, discarding the ones with sequence number below or equal to the one from the REST orderbook.
> - Apply the next queued messages to your local order book data structure.
> - Apply real-time messages to your local orderbook as they come in through the stream.

4. Display the **X** best bids and **X** best asks in real-time on a graphical UI. The number **X** is an integer and should be a runtime configuration.
5. Use the REST API (not the websocket) to poll for recent trades at some regular interval.
6. Display the **X** most recent trades on the same UI.
7. Write a contrarian trading strategy that does not actually execute trades, but will update the UI when it would have traded if this were in a live, production trading strategy. This strategy will work by counting the **M** consecutive upticks and **N** consecutive downticks. A trade that executes at a price that is the same as the price of the trade that executed immediately preceding it is known as a "zero tick". An uptick is when a trade executes at a higher price than the most recent non-zero-tick trade before it. A downtick is when a trade executes at a lower price than the most recent non-zero-tick trade before it. After **M** consecutive upticks, the algorithm should sell 1 BTC at the price of the most recent uptick. After **N** consecutive downticks, it should buy 1 BTC at the price of the most recent downtick. **M** and **N** should also be runtime configurations. For example:
    a. UP -> UP -> UP = 3 upticks
    b. UP -> DOWN -> UP = 1 uptick
    c. DOWN -> ZERO -> DOWN -> DOWN = 3 downticks
    d. DOWN -> UP -> DOWN -> DOWN = 2 downticks
8. Instead of actually trading, the algorithm only needs to add the trade it would have wanted to execute to the list of trades that is displayed on the UI. There should be something on the UI that helps differentiate real trades from imaginary trades executed by the trading algorithm.
9. Add a README that explains how we can run your application and any other relevant info.
10. At the bottom of your README, copy the checklist included below and fill in the blank spaces.
11. Email a link to your repository with CV attached to marvin@sonartrade.com when you have completed this assignment. Feel free to email a question as well if something isn't clear in this problem statement.

# Appendix A: Checklist

Copy this checklist into your README file and fill in the blanks with the filename and method name where we can find implementation of specific features.

| **Feature** | **File name** | **Method name** |
|---|---|---|
| Schedule the polling of trades over REST. | | |
| Request a book snapshot over REST. | | |
| Listen for diff-orders over websocket. | | |
| Replay diff-orders. | | |
| Use config option X to request recent trades. | | |
| Use config option X to limit number of ASKs displayed in UI. | | |
| The loop that causes the trading algorithm to reevaluate. | | |