# Pragma Web UI Developer Coding Challenge – 2020

## Overview

We want you to implement a React Web SPA that displays in real-time the first 2 minutes of trading activity on a specific day for IBM stock across all US stock exchanges. Your Web app will receive the trading activity as a stream of Server Sent Events (SSE) from the provided web server. Your app should display this activity on a couple of components described later in this document.

## Supporting material

Supporting material for this challenge is available on GitHub. You'll receive an invitation to collaborate on this repo: https://github.com/sr33pragma/WebCodingChallenge.git
Here's a summary of the supporting material:

### 1. Pragma Web Developer Coding Challenge.xls -

An XLS with two pages:

- The *Events* page contains the data that our webserver streams to your app.
- The *Mock UX* page has UX for the components you will implement. The components are described later in this document. The data displayed on this UX is a snapshot of what the UI should display when event# 1380 is the current event.

### 2. A provided Web Server that streams trading activity

This webserver that streams the trading activity as SSE events. You'll find it in the **server/ folder** of the repo. Note that the fully functional webserver is **provided to you and you do NOT need to implement it**, just use it. It is a Node.js server. Instructions on using this server:

- To start the server:
  - node sse.js

- To test the server from your browser, browse to localhost:3000 and try it out. You should see the events display on your browser as json, with the most recent event at the top. See the file **server/sse.html** for calls made to the server. Buttons:
  - Start/Stop - start or stop streaming trade events to your browser.
  - Reset - stop & reset to seq#1, 1x replay speed
  - 0.1x … 10X - stop & set replay speed to 0.2x, takes effect on the next start.

## The Coding challenge

You will implement a React Web SPA with two components described further down. Your web app will make http requests to the provided server, receive a stream of SSE trading events, and display them on your components. The server will stream approximately 3000 trading events over 2 minutes. Your app should also be able to gracefully handle and display events at 5x this rate.

## What you'll implement

Implement the following in the order specified. Refer to the Mock UX page on the xls for how things should look. Feel free to improve on the UX.

## 1. Clock, Stats and Replay speed controls

- Implement the ***Events*** panel from the Mock UX page
- Process all events from the event stream, including Bid, Ask, NBBO and Trade events
- Display the time corresponding to the current event as hh:mm:ss.nnn. Time should display to millisecond precision, and derived from the Unix Epoch Timestamp column of the event
- Display the Event details of the current event – Event, Exch1, Price1, Shares1, Exch2, Price2, Shares2 map to the correspondingly named columns on the Events page of the xls
- Implement Replay controls that can be used to (re)start, stop/pause, and reset events from the Server, and change the replay speed. Changes to the replay speed should take effect on the subsequent start.
- Add some color/style to make it look cool

## 2. Order Book panel

The order book panel shows the current best Bid and Ask at each exchange. Implement the ***Order Book*** panel from the Mock UX page on the xls:

- updated by Bid and Ask events only.
- Bid events update the *Bids* grid, Ask events update the *Asks* grid
- Values for cells – both Bid and Ask:
    - Exch: xchg1 of the event.
    - Shares: shares1 of the event
    - Price: price1 of the event
    - Age: calculated by you, seconds (not msec) elapsed between this event and the current event (1. above)
- Sorting
    - The Bids grid is sorted in **descending price** order. If two exchanges have the same price, the price with the earlier timestamp sorts ahead of the later price
    - The Asks grid is sorted in **ascending price** order. If two exchanges have the same price, the price with the earlier timestamp sorts ahead of the later price
    - Note that with the sorting, Bids and Asks for a given exchange may not line up side-by-side
- UX considerations:
    - indicate the most recently updated row on each grid with a visual cue – e.g. bold.
    - Price color bands – visually group rows with identical prices on the Bids grid using a visual cue – e.g. same background or text color. Do the same for the Asks grid.
    - Add some color to make it less sterile
    - Nice to have - Age bands : provide a visual cue (background color ???) to bucket the age of a Bid or Ask in one of 3 categories - less than 1 sec; 1 – 5 secs; greater than 5 secs

## The API

Here's the api and data formats to use when communicating with the provided server.

- Server API http GET endpoints for you to use:
    - /start - start streaming, or resume streaming from where you stopped. This is the only SSE call and of type text/event-stream.

    - /stop - stop streaming. More like 'pause' than 'stop'
    - /reset - stop and reset to start of stream, also set replaySpeed to 1x.
    - /set?speed=0.2 - stop, then set replay speed to 0.2x. Calling /start after this will replay at a speed of 0.2. Mae sure you can handle a speed of 5x

- Data format – each SSE message is a JSON with 2 elements:
    - **event**: corresponds to a row in the Events page of the XLS. Presented as an array of values corresponding to [Event#, Unix Epoch Timestamp, Event, Price1, Shares1, Xchg1, Price2, Shares2, Xchg2].
    - **time** : only used for troubleshooting. Two attributes:
        - wallClock – the actual elapsed time in secs since the most recent call to /start. Independent of the replaySpeed
        - eventTime – simulated seconds elapsed since the most recent /start. Depends on the replaySpeed. For replaySpeed = 1 this should match wallClock. At 10x speed, this will be 10x the wall clock. For 0.1x, will be 0.1x the wall clock.
- Sample data:
    - Bid - {"event":[1025,1563197467602,"Bid",142.5,800,"X"], "time":{"wallClock":2.648, "eventTime":52.887}}
    - Ask - {"event":[1028,1563197467604,"Ask",142.9,200,"N"], "time":{"wallClock":2.648,"eventTime":52.889}}
    - Trade - {"event":[1027,1563197467604,"Trade",142.89,200,"N"], "time":{"wallClock":2.648, "eventTime":52.889}}
    - NBBO - {"event":[1026,1563197467602,"NBBO",142.84,100,"Q",142.89,200,"N"], "time":{"wallClock":2.648,"eventTime":52.887}}

## Points to consider:

- It may not make sense to update the screen for every event - both from a performance perspective - will the UI freeze with 100s of updates/second), as well as utility - is there any value in updating the screen 20+ times in one second.
- However, you may NOT discard events because every event needs to be processed in order to keep the model / state accurate.
- Feel free to use a 3rd party grid, but make sure there's enough of 'your code' in there.

## What we're looking for in your implementation

- We want you to use Typescript if possible (or else JavaScript), React, and implement this as a SPA.
- First and foremost, we are looking for a functionally accurate and complete solution, one that demonstrates you've thought through the problem including corner cases.
- Include instructions on how to build and test your work.
- We're happy to consider opinionated UI/UX choices and deviations from the xls mock UX.
- We want to see your ability to write production quality code – maintainable, performant and good design choices.
- Performant – try to make your implementation performs adequately at 5x speed without hogging resources.
- Given the time constraints, you'll need to make intelligent tradeoffs in terms of how much time you spend on different aspects of the implementation.

- Finally, we're looking to gain insight into your thought process, engineering skills and craftsmanship from your solution – so make sure 'your code represents you'

Feel free to ping me with questions, weekends and evenings to 10pm are fine - sramanathan@pragmatrading.com or 609.310.1662. If you don't hear back within an hour, call me.