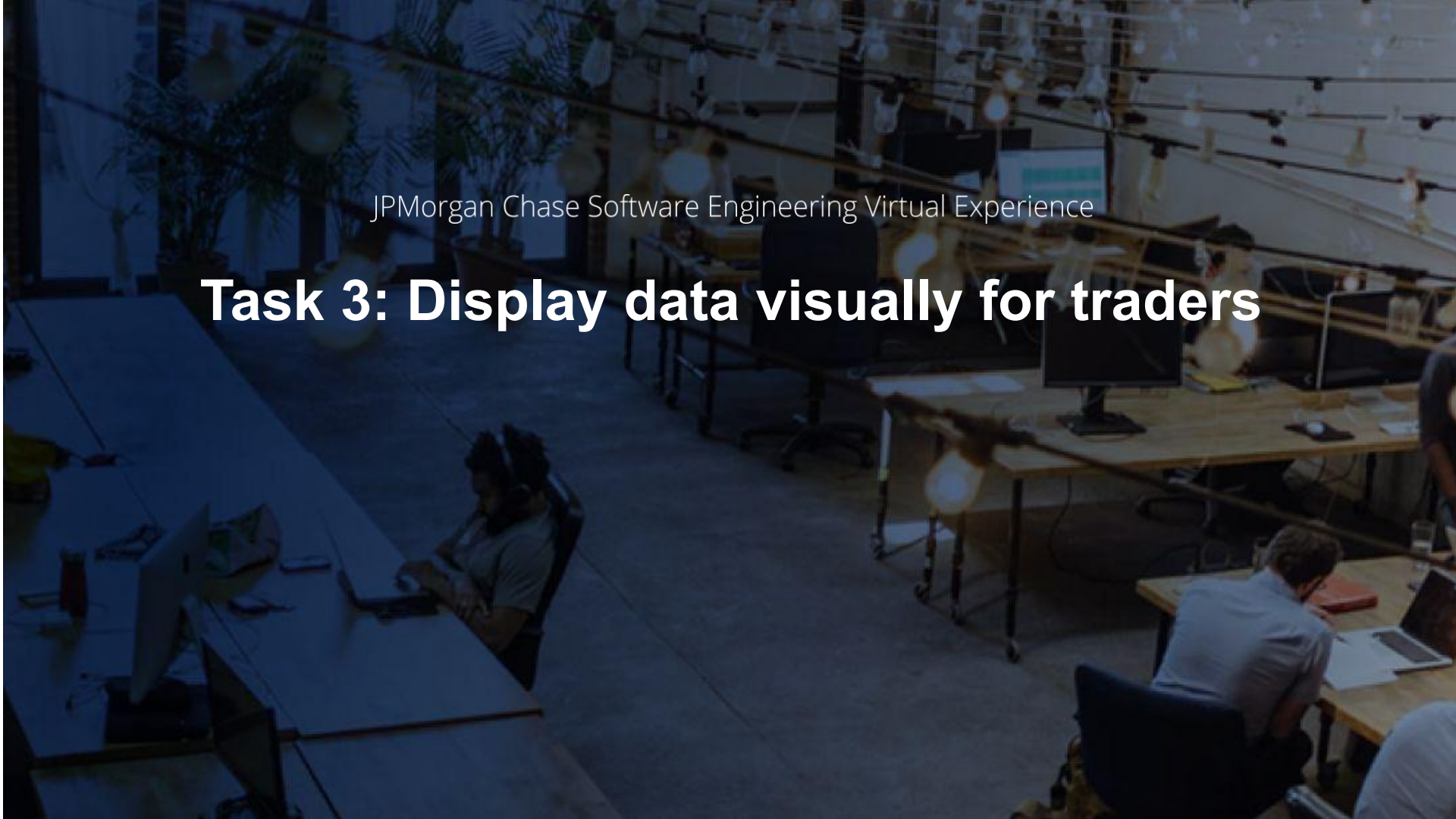

JPMorgan Chase Software Engineering Virtual Experience
Task 3: Display data visually for traders

Initial Client State

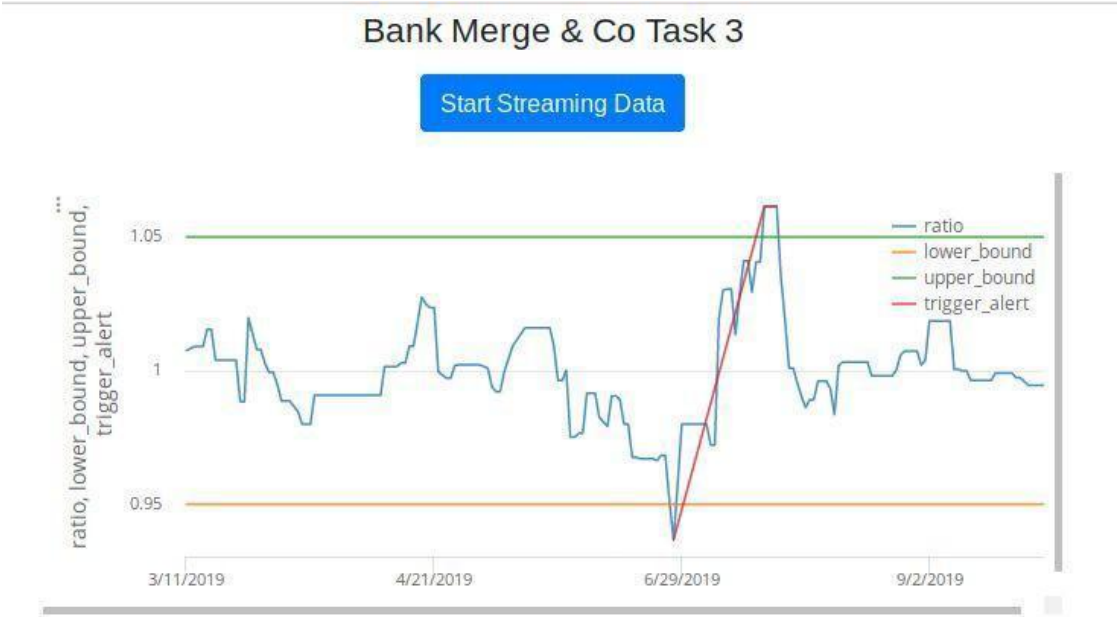
This is what the initial state of the client looks like when you click the blue “**Start Streaming Data”** button

It’s nearly identical to what you ended up with in task 2. Two stocks are displayed and their top_ask_price changes over time

Objectives

- There are two things we have to achieve:
    - (1) We want to make this graph more useful to traders by making it track the **ratio** between two stocks over time and NOT the two stocks' top_ask_price.
    - (2) As mentioned before, traders want to monitor the ratio of two stocks against a historical correlation with **upper and lower bounds**. This can help them determine a trading opportunity. Therefore we also want to make this graph plot the upper and lower bounds and show when they get crossed

- In the end we want to achieve a graph that looks something like this



- To accomplish our first objective, we'll make changes to the `**Graph.tsx**` file. Recall that this is the file which determines how the Graph component of our App will be rendered and reactive to state changes.
- We're not starting from a static graph anymore, beginning instead with what we had at the end of task 2. We want to implement one main line tracking the ratio of two stocks, as well as upper and lower bounds

- To accomplish this, we need to modify the `**componentDidMount**` method. Recall that the **componentDidMount()** method runs after the component output has been rendered to the DOM. If you want to learn more about it and other lifecycle methods/parts of react components, read more here.

- In this method, we'll modify the **schema** object, which configures the Perspective table view of our graph. In the next slide we'll show you what to change.

```
const schema = {
  stock: 'string',
  top_ask_price: 'float',
  top_bid_price: 'float',
  timestamp: 'date',
};
```

```
const schema = {
  price_abc: 'float',
  price_def: 'float',
  ratio: 'float',
  timestamp: 'date',
  upper_bound: 'float',
  lower_bound: 'float',
  trigger_alert: 'float',
};
```

Before

After

- Notice the changes we've made to schema:
  - Since we don't want to distinguish between two stocks now, preferring to track their ratios, we added the **ratio** field. Since we also wanted to track **upper_bound**, **lower_bound,** and the moment when these bounds are crossed i.e. **trigger_alert**, we've added corresponding fields.
  - Finally, we added **price_abc** and **price_def** because these are necessary to calculate the **ratio.** We won't be configuring the graph to show them.
  - Since we're tracking all of this with respect to time, we need a **timestamp** field

- Next, to configure the graph, you will need to modify/add more attributes to the element. The change should look something like:

```
  elem.load(this.table);
  elem.setAttribute('view', 'y_line');
  elem.setAttribute('column-pivots', '["stock"]');
  elem.setAttribute('row-pivots', '["timestamp"]');
  elem.setAttribute('columns', '["top_ask_price"]');
  elem.setAttribute('aggregates', JSON.stringify({
    stock: 'distinctcount',
    top_ask_price: 'avg',
    top_bid_price: 'avg',
    timestamp: 'distinct count',
  }));
}
```

Before

```
  elem.load(this.table);
  elem.setAttribute('view', 'y_line');
  elem.setAttribute('row-pivots', '["timestamp"]');
  elem.setAttribute('columns', '["ratio", "lower_bound", "upper_bound", "trigger_alert"]');
  elem.setAttribute('aggregates', JSON.stringify({
    price_abc: 'avg',
    price_def: 'avg',
    ratio: 'avg',
    timestamp: 'distinct count',
    upper_bound: 'avg',
    lower_bound: 'avg',
    trigger_alert: 'avg',
  }));
}
```

After

- ‘**view**’ is the kind of graph we want to visualize the data with. Initially, this is already set to **y_line**.

- **‘column-pivots’** used to exist and was what allowed us to distinguish / split stock ABC with DEF back in task 2. We removed this because we’re concerned about the ratios between the two stocks and not their separate prices

- ‘**row-pivots’** takes care of our x-axis. This allows us to map each datapoint based on its timestamp. Without this, the x-axis would be blank.

- ‘**columns’** is what allows us to focus on a particular part of a datapoint’s data along the y-axis. Without this, the graph would plot all the fields and values of each datapoint, yielding significant noise. We want to track **ratio, lower_bound, upper_bound** and **trigger_alert**.

- **‘aggregates’** allows us to handle the duplicate data we observed in task 2 and consolidate them into one data point. In our case we only want to consider a data point unique if it has a timestamp. Otherwise, we’ll average the values of the other non-unique fields these ‘similar’ data points share before treating them as one (e.g. ratio, price_abc, …)

- Finally, we have to make a slight update to the **componentDidUpdate** method. This is another component lifecycle method that gets executed whenever the component updates, i.e. when the graph gets new data. The change we want to make is on the argument we put in **this.table.update**.

```
    componentDidUpdate() {
        if (this.table) {
            this.table.update([
                DataManipulator.generateRow(this.props.data),
            ] as unknown as TableData);
        }
    }
}
```

- Now we have to make some modifications in the **DataManipulator.ts** file. This file is responsible for processing the raw stock data we receive from the server before the Graph component renders it.
- The first thing we have to modify in this file is the **Row** interface. The **Row** interface is initially almost identical to the old **schema** in **Graph.tsx** before we updated it, so we have to update it to match the new **schema**.

```
3   export interface Row {
4     stock: string,
5     top_ask_price: number,
6     timestamp: Date,
7   }
```

```
4   export interface Row {
5     price_abc: number,
6     price_def: number,
7     ratio: number,
8     timestamp: Date,
9     upper_bound: number,
10    lower_bound: number,
11    trigger_alert: number | undefined,
12  }
```

- This change is necessary because it determines the structure of the object returned by the **generateRow** function
- This return object must correspond to the **schema** of the table in the Graph component

note: Interfaces help define the values a certain entity must have. If you want to learn more about interfaces in Typescript you can read this material

- Finally, we have to update the **generateRow** function of the DataManipulator class to properly process raw server data.

- First, we'll compute **price_abc** and **price_def** properly (recall what you did back in task 1). Afterwards we'll compute the ratio using both prices, set **lower** and **upper** bounds, and determine **trigger_alert**.

```
15  export class DataManipulator {
16    static generateRow(serverRespond: ServerRespond[]): Row {
17      const priceABC = (serverRespond[0].top_ask.price + serverRespond[0].top_bid.price) / 2;
18      const priceDEF = (serverRespond[1].top_ask.price + serverRespond[1].top_bid.price) / 2;
19      const ratio = priceABC / priceDEF;
20      const upperBound = 1 + 0.05;
21      const lowerBound = 1 - 0.05;
22      return {
23        price_abc: priceABC,
24        price_def: priceDEF,
25        ratio,
26        timestamp: serverRespond[0].timestamp > serverRespond[1].timestamp ?
27          serverRespond[0].timestamp : serverRespond[1].timestamp,
28        upper_bound: upperBound,
29        lower_bound: lowerBound,
30        trigger_alert: (ratio > upperBound || ratio < lowerBound) ? ratio : undefined,
31      };
32    }
```

Feel free to change this to +/-10% of the 12 month historical average ratio

This was just for a test value

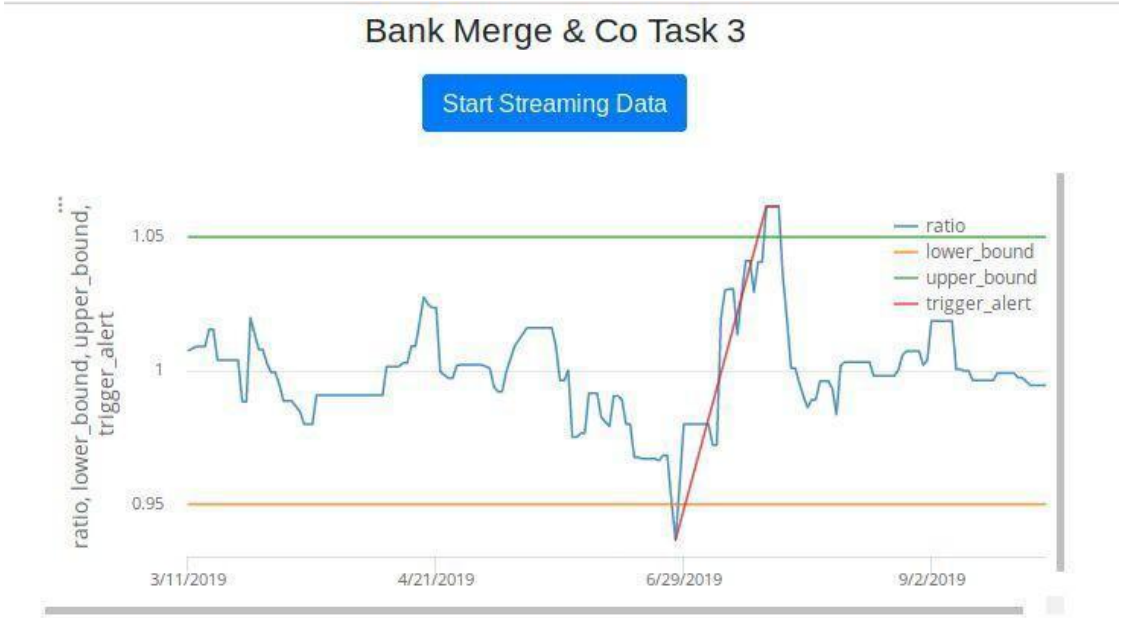
- Observe how we're able to access **serverRespond** as an array wherein the first element is stock ABC and the second element is stock DEF

- Also note how the return value is changed from an array of **Row objects** to just a single **Row object** This change explains why we also adjusted the argument we passed to **table.update** in Graph.tsx earlier so that consistency is preserved.

- The **upper_bound** and **lower_bound** are pretty much constant for any data point. This is how we will be able to maintain them as steady upper and lower lines on the graph. While 1.05 and 0.95 isn't really +/-10% of the 12 month historical average ratio (i.e. 1.1 and 0.99) you're free to play around with the values and see which has a more conservative alerting behavior.

- The **trigger_alert** field is pretty much just a field that has a value (e.g. the ratio) if the threshold is passed by the ratio. Otherwise if the ratio remains within the threshold, no value/undefined will suffice.

End Result


Bank Merge & Co Task 3
Start Streaming Data
ratio, lower_bound, upper_bound, trigger_alert
1.05
1
0.95
3/11/2019
4/21/2019
6/29/2019
9/2/2019
ratio
lower_bound
upper_bound
trigger_alert