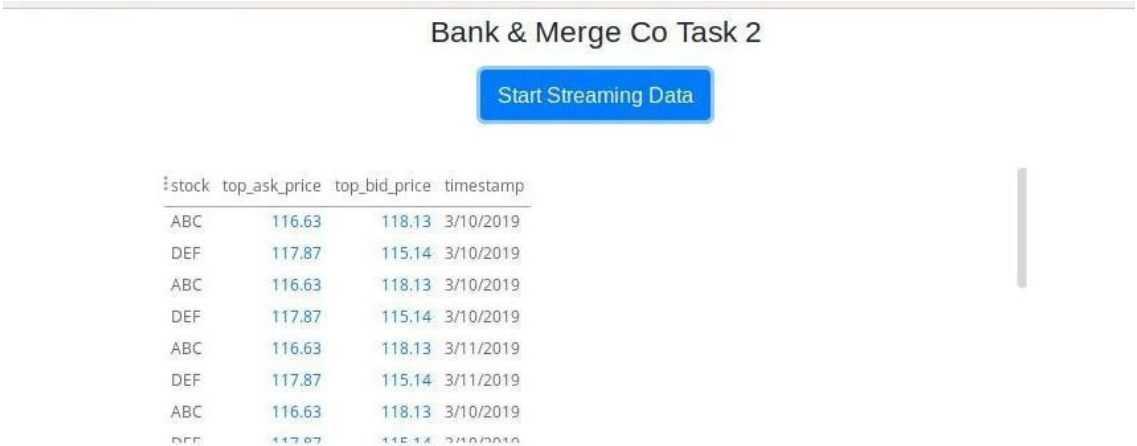

JPMorgan Chase Software Engineering Virtual Experience
Task 2: Use JPMC frameworks and tools

Observe Initial State Of The Client



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

Observe Initial State of The Client

If you look back at the data again, you'll observe it's just a bunch of duplicates being printed for stocks ABC and DEF until newer data becomes available

But printing duplicate data doesn't seem useful at all...

Objectives

- There are two things we need to do to complete this task
    - (1) Make the graph update continuously instead of having to click it a bunch of times. We want to serve a line graph whose y axis is the stock's **top_ask_price** and x-axis is the stock's timestamp
    - (2) Remove / disregard the duplicate data we saw earlier…

Objectives

- The kind of graph we want is something like this:

Objectives

- To achieve this we have to change (2) files: **src/App.tsx** and **src/Graph.tsx**

- You can use any text editor your machine has and just open the files in the repository that must be changed (the guide will instruct you in the following slides which files these will be)

- Our recommendation of editors you can use would be VSCode or SublimeText as these are the most commonly used code editors out there.

Making changes in `**App.tsx**`

- App.tsx is the main app (*typescript*) file of our client side react application.

- Don’t be intimidated by words like React, which is just a JavaScript library to help us build interfaces and ui components, or Typescript which is just a superset of JavaScript... We’ll walk you through the changes needed.

Making changes in `**App.tsx**`

- App.tsx or the App component, is basically the first component our browser will render as it's the parent component of the other parts of the simple page that shows up when you first start the application in the set up phase.

- Components are the building blocks of our web application. A component has a common set of properties / functions and as such, each unique component just inherits from the base React component

- App.tsx is the first file we will have to change to achieve our objective

Making changes in `**App.tsx**`

- Red box: The App component (App.tsx)
- Yellow box: The Graph component (Graph.tsx)

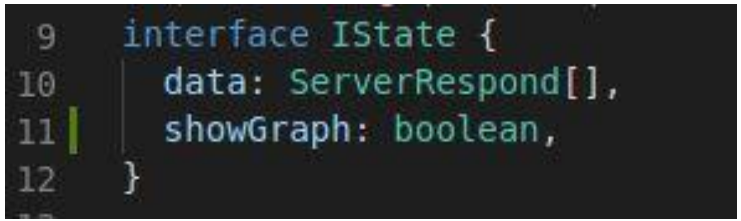
```
49    render() {
50      return (
51        <div className="App">
52          <header className="App-header">
53            Bank & Merge Co Task 2
54          </header>
55          <div className="App-content">
56            <button className="btn btn-primary Stream-button" …
65            <div className="Graph">
66              {this.renderGraph()}
67            </div>
68          </div>
69        </div>
70      )
71    }
```

Making changes in `**App.tsx**`

- To achieve the objectives listed earlier, we'll first need to modify App.tsx to change the static table into a live graph. Follow instructions in the next few slides

Making changes in `**App.tsx**`

- First you'll need to add the `showGraph` property in the **IState** interface defined in App.tsx. It should be of the type`boolean`



*note: Interfaces help define the values a certain entity must have. In this case, whenever a type of **IState** is used, our application knows it should always have **data** and **showGraph** as properties in order to be valid. If you want to learn more about interfaces in Typescript you can read this material in your spare time*

Making changes in `**App.tsx**`

- Next, in the constructor for the App component, you should define the initial state of the graph as hidden. This is because we want the graph to show when the user clicks ‘Start Streaming Data’. That means you should set the `showGraph` property of the App’s state to `false` in the constructor

```
18  class App extends Component<{}, IState> {
19    constructor(props: {}) {
20      super(props);
21
22      this.state = {
23        // data saves the server responds.
24        // We use this state to parse data down to t
25        data: [],
26        showGraph: false,
27      };
28    }
```

Making changes in `**App.tsx**`

- To ensure that the graph doesn't render until a user clicks the 'Start Streaming' button, you should also edit the `**renderGraph**` method of the App, adding a condition to render the graph when the application state's `**showGraph**` property is `**true**`.

```
33    renderGraph() {
34      if (this.state.showGraph) {
35        return (<Graph data={this.state.data}/>)
36      }
37    }
38
```

*note: we had to do this because **renderGraph** gets called in the **render** method of the App component. To learn more about how react renders components you can read more here*

Making changes in `**App.tsx**`

- Finally, you must also modify the `**getDataFromServer**`method to contact the server and get data from it continuously instead of just getting data from it once every time you click the button.

- Javascript has a way to do things in intervals via the setInterval function. We can make it continuous with a guard value to stop the interval process we started.

- You should arrive at result similar to the next slide

Making changes in `**App.tsx**`

note: the only place you should assign the local state directly is in the constructor - use setState() elsewhere.

```
getDataFromServer() {
  let x = 0;
  const interval = setInterval(() => {
    DataStreamer.getData((serverResponds: ServerRespond[]) => {
      this.setState({
        data: serverResponds,
        showGraph: true,
      });
    });
    x++;
    if (x > 1000) {
      clearInterval(interval);
    }
  }, 100);
}
```

Making changes in `**Graph.tsx**`

- To completely achieve the desired output, we must also make changes to the `**Graph.tsx**` file. This is the file that takes care of how the Graph component of our App is rendered and reacts to any state changes.

- First, you must enable the `**PerspectiveViewerElement**` to behave like an **HTMLElement**. To do this, you can extend the `HTMLElement` class from the `**PerspectiveViewerElement**` interface.

```
17  interface PerspectiveViewerElement extends HTMLElement {
18      load: (table: Table) => void,
19  }
```

Making changes in `**Graph.tsx**`

- Now we need to modify the `**componentDidMount**` method. Just as a note, 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.

- Since you changed the `**PerspectiveViewerElement**` to extend the `**HTMLElement**` earlier, you can simplify the `**const elem**` definition by assigning it directly to the result of `**document.getElementsByTagName**`.

```
33  componentDidMount() {
34    // Get element to attach the table from the DOM.
35    const elem = document.getElementsByTagName('perspective-viewer')[0] as unknown as PerspectiveViewerElement;
36
```

Making changes in `**Graph.tsx**`

- Finally, you need to add more attributes to the element. You'll need to add the following attributes: `**view**`, `**column-pivots**`, `**row-pivots**`, `**columns**` and `**aggregates**`. The end result should look something like:

```
52        elem.setAttribute('view', 'y_line');
53        elem.setAttribute('column-pivots', '["stock"]');
54        elem.setAttribute('row-pivots', '["timestamp"]');
55        elem.setAttribute('columns', '["top_ask_price"]');
56        elem.setAttribute('aggregates', `
57          {"stock":"distinct count",
58          "top_ask_price":"avg",
59          "top_bid_price":"avg",
60          "timestamp":"distinct count"}`);
```

Making changes in `**Graph.tsx**`

- ‘**view**’ is the kind of graph we want to visualize the data with. Initially, we were using a **grid** type. However, since we want a continuous line graph we’re using a **y_line.**

- **‘column-pivots’** is what will allow us to distinguish stock ABC from DEF. We use ‘[**“stock”**]’ as its value here.

- ‘**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.

Making changes in `**Graph.tsx**`

- ‘**columns’** allows us to focus on a particular part of a stock’s data along the y-axis. Without this, the graph would plot different data points of a stock, i.e.: top_ask_price, top_bid_price, stock, timestamp. For this instance we only care about **top_ask_price**

- **‘aggregates’** allows us to handle the duplicated data we observed earlier and consolidate it into a single data point. In our case we only want to consider a data point unique if it has a unique stock name and timestamp. If there are duplicates like what we had before, we want to average out the top_bid_prices and the top_ask_prices of these ‘similar’ data points before treating them as one.

Wrapping up

- By now you've accomplished all the objectives of the task (as specified in the earlier Objectives slide) and ended up with a graph like the one in the next slide
- Well done! Time to commit and push your changes, then submit your work!

End Result


Bank & Merge Co Task 2
Start Streaming Data
top_ask_price
150
140
130
120
110
3/11/2019
6/15/2019
10/11/2019
2/8/2020
timestamp: 9/30/2019
stock: DEF
top_ask_price: 149.43
ABC
DEF