



InsideSherpa

JPMorgan Chase Software Engineering Virtual Experience

# **Task 2 - software engineering task : code changes**

Module 2 - Use JPMorgan Chase frameworks and tools

# Disclaimer

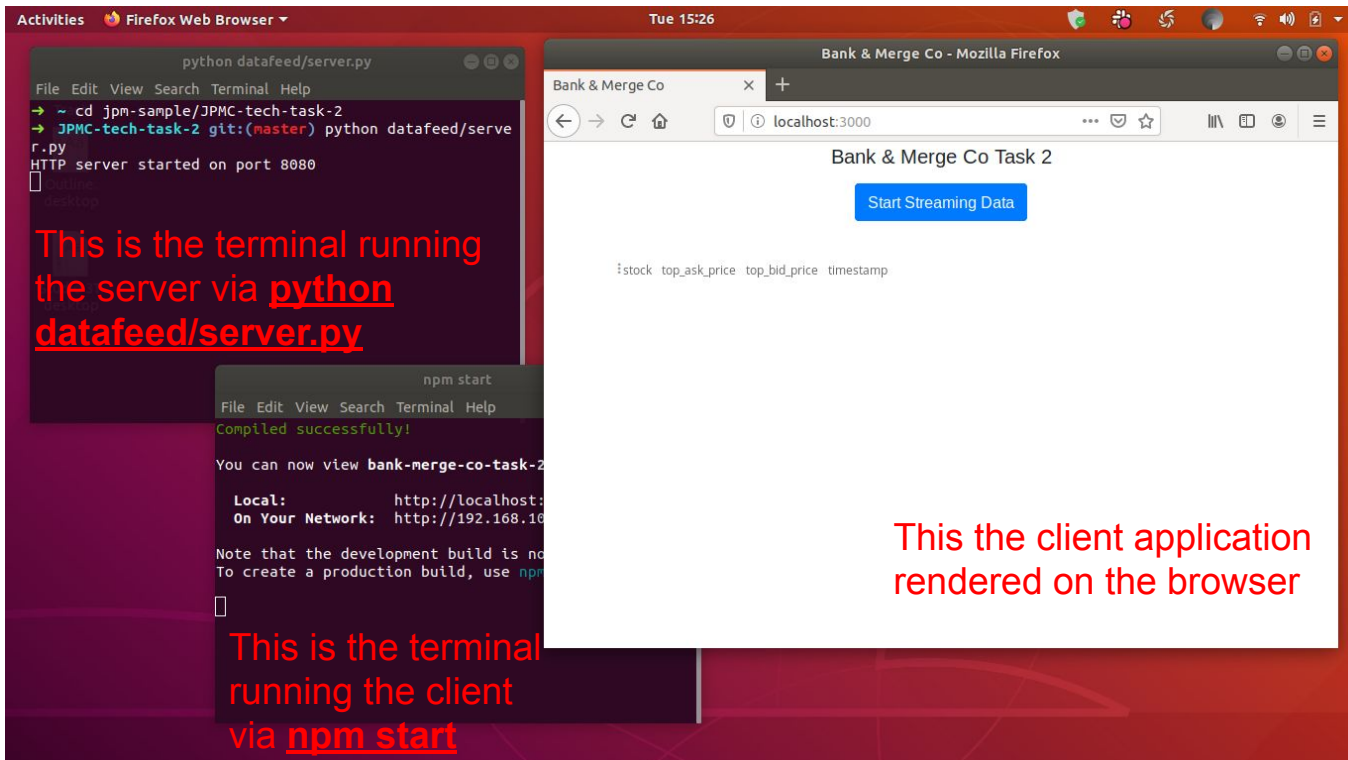
- This guide is only for those who did the setup locally on their machines.

# Prerequisite

- Set up should have been done. This means, your server and client applications should have been running with no problems without introducing any changes to the code yet. You can verify this if you get a similar result to any of the following slides that include a picture of the server and client app running together

# Prerequisite

## Mac OS / Linux OS



The screenshot shows a Linux desktop with a terminal window and a web browser window. The terminal window is titled 'python datafeed/server.py' and shows the following commands and output:

```
File Edit View Search Terminal Help
→ ~ cd jpm-sample/JPMC-tech-task-2
→ JPMC-tech-task-2 git:(master) python datafeed/server.py
HTTP server started on port 8080
```

Below the terminal window, there is a smaller terminal window titled 'npm start' showing the following output:

```
File Edit View Search Terminal Help
Compiled successfully!
You can now view bank-merge-co-task-2
Local: http://localhost:3000
On Your Network: http://192.168.1.10:3000
Note that the development build is not optimized.
To create a production build, use npm run build
```

The web browser window is titled 'Bank & Merge Co - Mozilla Firefox' and shows a web application titled 'Bank & Merge Co Task 2'. The application has a blue button labeled 'Start Streaming Data' and a table with the following columns: 'stock', 'top\_ask\_price', 'top\_bid\_price', and 'timestamp'.

This is the terminal running the server via python datafeed/server.py

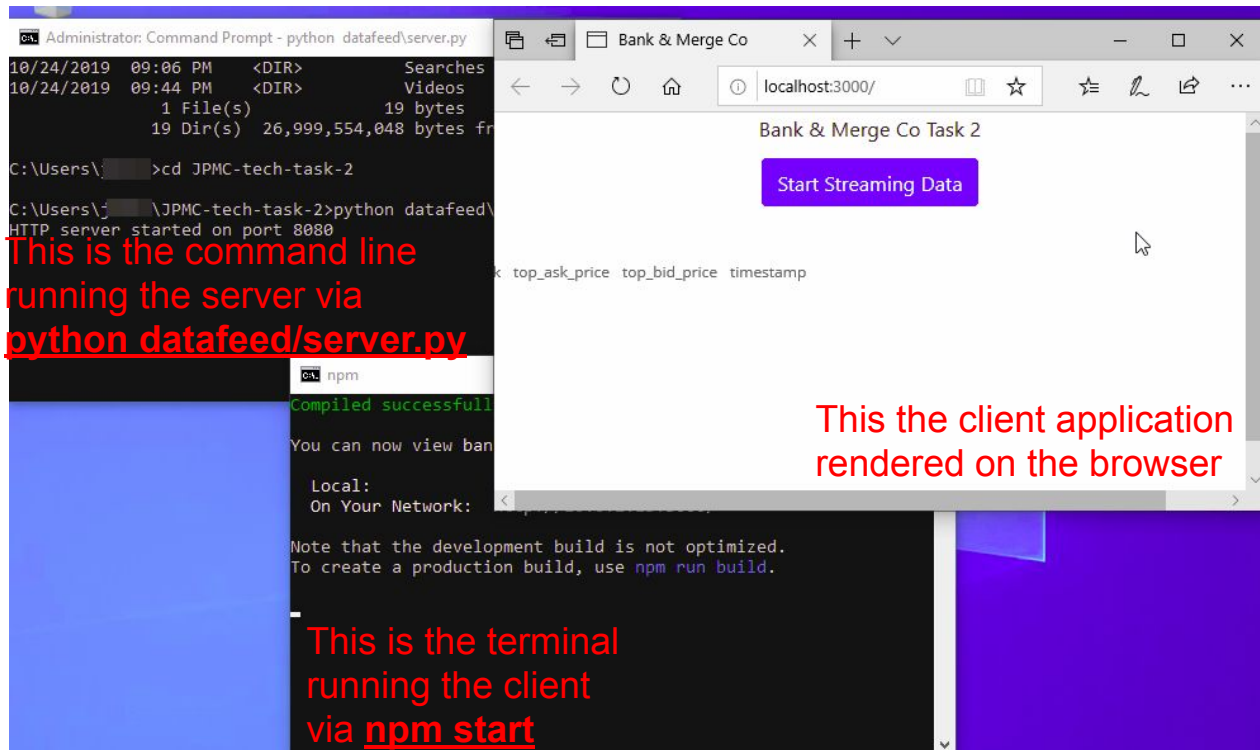
This is the terminal running the client via npm start

This the client application rendered on the browser

Note: The state is similar for those who used the python3 version of the repository

# Prerequisite

## Windows OS



Note: The state is similar for those who used the python3 version of the repository

This the client application rendered on the browser

This is the terminal running the client via **npm start**

This is the command line running the server via **python datafeed/server.py**

# Observe Initial State Of Client App in Browser

Bank & Merge Co Task 2

Start Streaming Data

stock	top_ask_price	top_bid_price	timestamp
ABC	116.63	118.13	3/10/2019
DEF	117.87	115.14	3/10/2019
ABC	116.63	118.13	3/10/2019
DEF	117.87	115.14	3/10/2019
ABC	116.63	118.13	3/11/2019
DEF	117.87	115.14	3/11/2019
ABC	116.63	118.13	3/10/2019
DEF	117.87	115.14	3/10/2019

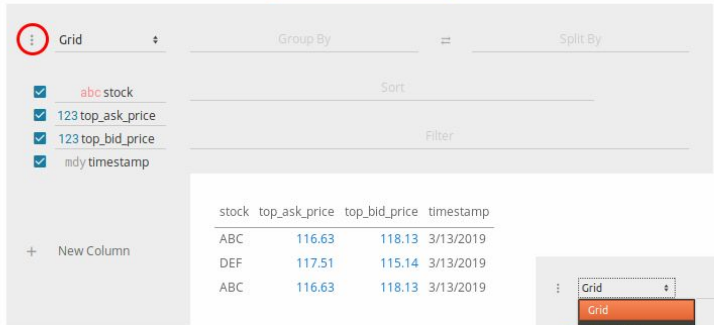
This is how the initial state of the client app looks like when you click the blue “**Start Streaming Data**” button a number of times

# Observe Initial State Of Client App in Browser

Bank & Merge Co Task 2

Start Streaming Data

image(a)



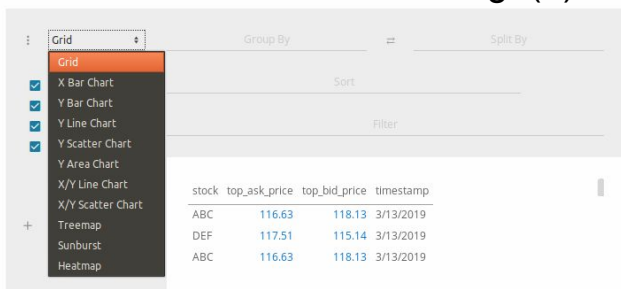
The screenshot shows the 'Grid' view selected in the upper left corner. The data table is as follows:

stock	top_ask_price	top_bid_price	timestamp
ABC	116.63	118.13	3/13/2019
DEF	117.51	115.14	3/13/2019
ABC	116.63	118.13	3/13/2019

Bank & Merge Co Task 2

Start Streaming Data

image(b)



The screenshot shows the 'Grid' view selected in the upper left corner. The data table is as follows:

stock	top_ask_price	top_bid_price	timestamp
ABC	116.63	118.13	3/13/2019
DEF	117.51	115.14	3/13/2019
ABC	116.63	118.13	3/13/2019

If you clicked on the 3-dotted button on the upper left corner of the graph you'll see something like image(a) on this slide.

This tells you that the graph is configurable

Image(b) further shows you the different types of views you can use to visualize the data you have so far

# Observe Initial State Of Client App in Browser

## Bank & Merge Co Task 2

Start Streaming Data

stock	top_ask_price	top_bid_price	timestamp
ABC	116.63	118.13	3/10/2019
DEF	117.87	115.14	3/10/2019
ABC	116.63	118.13	3/10/2019
DEF	117.87	115.14	3/10/2019
ABC	116.63	118.13	3/11/2019
DEF	117.87	115.14	3/11/2019
ABC	116.63	118.13	3/10/2019
DEF	117.87	115.14	3/10/2019

If looked back at the data again, you'll also observe it's just a bunch of duplicate data being printed for stocks ABC and DEF until such point that there becomes newer data i.e. different timestamp,

**ask\_price** and **bid\_price** for ABC and DEF stocks

But the printing of duplicated data doesn't seem useful at all...



# Objectives

- There are two things we have to achieve here to complete this task
  - (1) Make the graph continuously update instead of having to click it a bunch of times. Also the kind of graph we want to serve as visual here is kind of a continuously updating line graph whose **y axis is the stock's top\_ask\_price** and the **x-axis is the timestamp of the stock**
  - (2) Remove / disregard the duplicated data we saw earlier...

# Objectives

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

Bank & Merge Co Task 2

Start Streaming Data



# Objectives

- To achieve this we have to change (2) files: **src/App.tsx** and **src/Graph.tsx**
- Don't worry we'll walk you through how to get these things done
- 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 but is still alike with javascript but with stronger type checking... 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 started the application in the set up phase. 提交
- Components are basically the building blocks / parts 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 objectives

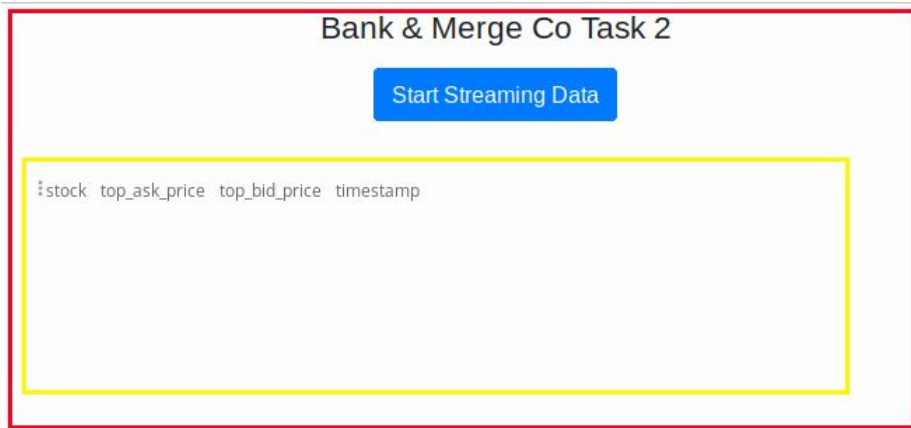
# Making changes in `App.tsx`

- Red box: The App component (App.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  }

```



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

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

- To achieve the objectives listed earlier, we'll first need to make changes in App.tsx file to help us change the static table into a live / updating 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`

```
9  interface IState {  
10     data: ServerRespond[],  
11     showGraph: boolean,  
12 }
```

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 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 of the App component, you should define that the initial state of the App not to show the graph yet. This is because we want the graph to show when the user clicks 'Start Streaming Data'. That means you should set `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   }
```

*note: It's okay if you don't fully grasp what a constructor is yet. You can read up more on it [here](#). For now, just understand it's a method / function that's automatically called when you create an instance of a class.*

*Also, the keyword **extends** comes from [Object Oriented Programming Paradigm's inheritance](#) that basically allows classes to inherit properties of another class (in this case the [base React Component](#)).*

# 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. In there, you must add a condition to only render the graph when the state's **showGraph** property of the App's state 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 and that is via the [setInterval function](#). What we can do to make it continuous (at least up to an extended period of time) is to have a guard value that we can check against when to stop / clear the interval process we started.
- You should arrive with a similar result as the next slide when you apply the modifications properly...

# Making changes in `App.tsx`

```

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

```

*note: the only place that you should assign the local state directly is in the constructor. Any place else in our component, you should rely on `setState()`. This hinges on the concept of immutability*

# Making changes in `App.tsx`

- If you noticed in the image in previous slide, it's in the same method, **getDataFromServer**, that we set **showGraph** to **true** as soon as the data from the server comes back to the requestor.
- The line **DataStream.getData(... => ...)** is an asynchronous process that gets the data from the server and when that process is complete, it then performs what comes after the **=>** as a [callback function](#).

# Making changes in `App.tsx`

- Changes in **App.tsx** end here.
- By now you should've accomplished modifying the client to request data from server continuously
- By now you should've also accomplished setting the initial state of the graph not to show until the user clicks the “**Start Streaming Data**” button
- Proceed to the next set of slides to finish the exercise with changes in **Graph.tsx**

# 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 will be rendered and react to the state changes that occur within the App.
- 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`

- After doing this, we now need to modify `**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've changed the `**PerspectiveViewerElement**` to extend the `**HTMLElement**` earlier, you can now make the definition of the `**const elem**` simpler, i.e. you can just assign it straight to the result of the `**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. For this you have to have read thru the [Perspective configurations particularly on the table.view configurations](#). You'll need to add the following attributes: ``view``, ``column-pivots``, ``row-pivots``, ``columns`` and ``aggregates``. If you remember the [earlier observations we did in the earlier slides](#), this is the configurable part of the table/graph. 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 the kind of graph we wanted to visualize the data as. Initially, if you remember this was the **grid** type. However, since we wanted a continuous line graph to be the final outcome, the closest one would be **y\_line**
- **'column-pivots'** is what will allow us to distinguish stock ABC with DEF. Hence we use `["stock"]` as its corresponding value here. By the way, we can use stock here because it's also defined in the **schema** object. This accessibility goes for the rest of the other attributes we'll discuss.
- **'row-pivots'** takes care of our x-axis. This allows us to map each datapoint based on the timestamp it has. Without this, the x-axis is blank.

# Making changes in `Graph.tsx`

- **'columns'** is what will allow us to only focus on a particular part of a stock's data along the y-axis. Without this, the graph will plot different datapoints of a stock ie: `top_ask_price`, `top_bid_price`, `stock`, `timestamp`. For this instance we only care about **`top_ask_price`**
- **'aggregates'** is what will allow us to handle the duplicated data we observed earlier and consolidate them as just one data point. In our case we only want to consider a data point unique if it has a unique stock name and timestamp. Otherwise, if there are duplicates like what we had before, we will average out the `top_bid_prices` and the `top_ask_prices` of these 'similar' datapoints before treating them as one.

# Wrapping up

- Changes in **Graph.tsx** are done too.
- By now you should'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
- Feel free to poke around before completely saving everything and creating your patch file, e.g. see the different effects of changing the configurations would do to your table/graph.
- Please don't forget to leave comments in your code especially at the places where the fixes for bugs and where you piped in the data feed - this will help with other team member's understanding of your work.

# End Result

## Bank & Merge Co Task 2

Start Streaming Data

