React Tutorial

Setting Up The Environment | How React Works | Creating Our Web App | Calling The API | Year Chart | Month Chart | Final Small Changes | Conclusion

This tutorial will walk you through how to build a React.js website calls a web API and displays the resulting data in three charts.

We will be recreating this site: https://philly-311-react.glitch.me/

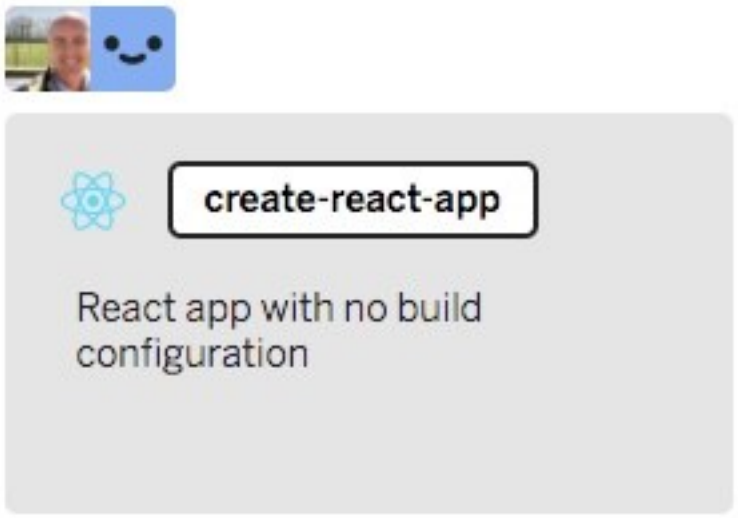
The dataset we will use is "311 Service and Information Requests" from Open Data Philly.

This tutorial assumes you are familiar with HTML, CSS, and JavaScript.

Setting Up The Environment

We are going to build our react site on a free web design playground called glitch. Go to glitch.com and click sign in to make an account. Follow the prompts to make a free account.

Next, in the search bar, type in "create-react-app." Click on the "create-react-app" box that comes up that looks like this:



Next, click on “remix your own” as seen in the photo below:

At this point, glitch will make you a default file from that create-react-app template. You will have something that looks like this:

The name in the top left (mine seems to be “incandescent-air” …how poetic) is randomly created by glitch. If you want to change the name this link shows you how.
If you click the “show live” button at the top, the browser will open a new tab and you should see your web app, which at this point your site should look like this:

And we’re off! You now have a react app running that we can edit.

A Brief Overview Of How React Works

React works by inserting HTML elements into the DOM’s root. In practice, you create javascript class files which each return an HTML element. You also create a main javascript class file, often called ‘app’, which calls each of your child JS classes. These elements are added to a “virtual DOM.” Then, react uses something called the “diffing algorithm” to compare what is on the virtual DOM with what is currently on the actual DOM. If there are differences, react inserts those different elements from the virtual DOM into the real DOM.

Elements

The React documentation calls elements the "…smallest building blocks of React…they describe what you want to see on the screen." (reference)

As I learned React, I was also learning, JSX, ES6 and some ES7. They will be a bit interwoven here. React's documentation introduces the JSX syntax, and we will use it in this tutorial.

Because React encourages small class files which each return one HTML element, this JSX syntax becomes useful. We can write code that looks like this:

```
var myTitle = <h2>Sean Jawn</h2>;
```

Components

A component in React is composed of elements. You can think of a component like a div that contains other divs. In our case, we will do things like instantiate a component, loop over a database, create an HTML element for each returned row, then return that composite item as one component.

Render

Class file in React must have a "render" method. What this means is that when this class file is called, it creates an HTML element and returns it. We will use this to create each HTML element in our application. The advice is to keep classes as small as possible and to, in general, have each class return just one element or component to be inserted into the DOM.

State & Props

State refers to the data that a component has. I think of it as a little data store (but it's not a database).

tip... I often forgot when learning react that not every react class *needs* to have state.

Props refers to data that is passed into a component. If you create a child class that renders each item in an array as a list item, that class does not need to store the array in state. The parent class can pass in the array as a prop during instantiation like this myArray={this.state.myArray} and the child class can access that with this.props.myArray.

I (and many others I found online) ran into trouble related to trying to take advantage of state to show data on the screen. I assumed if I changed the state of a child class at some point, that child class would instantly re-render itself and show my new data on the screen. But... that's not the case.

Understanding how, and when, a component's state is updated is important and tricky to understand. State does not update in the manner you would expect, and so it starts to seem untrustworthy. If a child component has a local state, that state is set when the component is instantiated. Any calls to setState with a new value after that will add the requested change to a queue. The setState call *will* happen, but you are not guaranteed when, or in what order, the updates will happen.

The real world problem this caused me was that I would pass in new data to a child component, and set the child's state with the new data. I would then expect the child to rerender with the new data. It would only sometimes happen that way...other times the child would rerender with the "old" data before the new state information was set.

We will cover how to work with this and get the component to render as you expect.

Creating Our Web App

Open App.js and let's get coding.

App.js is going to be our parent class. App.js will have state information, and we will pass that information as props to child components. Those child components will take those props and do things like make API calls, render elements to the DOM, and return data that App.js will use to update its state information.

In order to have state, we need to make a constructor in App.js.

After the class declaration "class App extends Component {", enter this:

```
constructor(){
super();
this.state= {
};
}
```

We will, throughout this tutorial, continue to add data to the state we just created.

For our first element, we will call the Philly 311 API and find out the total number of requests to the agency since it began. Let's add a "totalRequests" parameter to state and initialize it to zero.

```
constructor(){
 super();
 this.state= {
  totalRequests: 0
 };
 }
```

Next, we will add a div in our return function that will use this state information. Around line 20 in app.js should be a closing from className="AppHeader."

After that closing div, paste in this code:

```
<div className="agencyselect">
 <h1>Philly 311 has handled {this.state.totalRequests} requests. </h1>
</div>
```

At this point, my app looks like this:

All right! We've created state and added a div that shows data from that state. Now, I happen to know that our claim that Philly 311 has handled 0 requests is false. They have handled thousands of requests. So, how do we get that data and show the correct number? We are going to use the Fetch API to make our request to the Philly 311 open data philly dataset.

An archive 1 of the project at this point: https://drive.google.com/open?id=1D1OwmSuQMK-Eic7gbfVnUUg-e5Zo00iJ

Calling The API

Next, we will connect to the Open Data Philly API to request the 311 data. We will start with getting the total number of rows in the 311 database, which we can interpret as the total requests Philly 311 has responded to since it began tracking requests.

As you see in app.js around line 23, we are trying to render {this.state.totalRequests} to the screen. We are storing a variable called "totalRequests" in the state of app.js. We will call the API, get back the total number, and store that data in this.state.totalRequests. That will trigger React to rerender the virtual DOM.

Let's use the factory method "componentDidMount" and put our API call in there. Put the following somewhere above the render() method in app.js.

```
componentDidMount(){
```

```
 var tempURL = this.state.baseURL + " count(*) as NUM FROM public_cases_fc";
 (async() => {
 try {
 var response = await fetch(tempURL);
 var data = await response.json();

this.setState({totalRequests: data.rows[0].num});
 } catch (e) {
 console.log("error")
 }
 })();
 }
```

There's one extra bit of work we need to do in glitch to get this to work: we need to install one dependency called fetch. In Glitch…

1. click the name of your project to open a drop down menu
2. click advanced options on the bottom of the drop down menu
3. click open console
4. in the new browser tab that opens, type the following then hit enter: npm install fetch

The photos below will show this process.

Next, we need to add some data to the state that the rest of our react app can use. If we look at the Open Data Philly API for the 311 data, we can see our queries will always have a common string that starts the url: "https://phl.carto.com/api/v2/sql?q=SELECT"

Let's put that into state so the state looks like this:

```
constructor(){
super();
this.state= {
baseURL: "https://data.phila.gov/carto/api/v2/sql?q=SELECT",
totalRequests: 0
};
}
```

Then, when we make a request, we will append the rest of whatever our query is to that baseURL. Since our first query is to get the total rows, we'll be querying COUNT(*) from a particular table. As you see under componentDidMount(), the concatenated URL will be: this.state.baseURL + " count(*) as NUM FROM public_cases_fc"

If you look further into the code under componentDidMount(), you'll see the words async, await, fetch, and setState. Async declares to be an asynchronous function. Await acts like "then, " which waits for a complete response before proceeding. Fetch is the dependency we are using to make actual requests over the network to a URL. Here is a fantastic post I read repeatedly about fetch, await, and async: https://jakearchibald.com/2015/thats-so-fetch/

Lastly, setState is part of react and is the proper way to change the value of a state variable. React warns strongly not to try setting state in way like this: this.state.myWhatever = 6. *tip* don't do that.
React suggests you think of state variables as immutable and that a difference is that setState acts as a request to change a state variable's value.

So, if all went well, your app now has two variables within state, code under componentDidMount() that makes and API call, and your site loads text similar to: "Philly 311 has handled 1778027 requests."

Here is an archive 2 of the project at this point: https://drive.google.com/open?id=1hRUFnPLUwfNlI10VkUNRc609M00ckdFa

Next, let's add another state variable called "yearToDisplay" that we will use throughout the app. We will give it an initial value of 2014, which is the first year there is data available in the Philly 311 database. So, our state should look like this:

```
constructor(){
super();
this.state= {
yearToDisplay: 2014,
baseURL: "https://data.phila.gov/carto/api/v2/sql?q=SELECT",
totalRequests: 0
};
}
```

Now, in app.js's render function, we can add in this.state.yearToDisplay to make the site a bit more informative. Change "<h1>Philly 311 has handled {this.state.totalRequests} requests. </h1>" to "<h1>Philly 311 has handled {this.state.totalRequests} requests since {this.state.yearToDisplay}. </h1>"

The site should now display something similar to: "Philly 311 has handled 1778027 requests since 2014." At this point, let's delete the next boilerplate HTML tag: "
To get started, edit <code>src/App.js</code> and save to reload.
"

We are now going to make a major new class file. On the left hand side of your browser in Glitch, click "new file." Type the following then click add file: src/service_name_data.js

At this point we need to install another dependency. Open the console again (project title -> advanced -> open console) then type: npm install react-chartjs-2 chart.js –save

This will install the chart.js library we will use to make graphic charts.

Paste the following import statements into the top of service_name_data.js:

```
import React from 'react';
import { Chart } from 'react-chartjs-2';
import './App.css';
import {Doughnut, Pie} from 'react-chartjs-2';
```

Next, create the class and call it ServiceNameChart.

```
class ServiceNameChart extends React.Component {


} //end class
export default ServiceNameChart;
```

What we are going to create in this class is code that calls the Philly 311 API and returns a doughnut chart with the types of services people call Philly 311 about.

This class will make use of its own state, as well as properties passed into it from app.js. As the user interacts with the website, different requests, such as mouse clicks, will be sent to and from this class so it can alter what data is shown.

Just below the class declaration, paste the following:

```
constructor(props){
 super(props);
 this.state = {
 localJson:{},
 serviceNameCountNumbers:[],
 serviceNameCountNames:[],
 backgroundColors:[],
 URLextension : " service_name, " +
 "COUNT (service_name) " +
```

```
 "FROM " +
 "public_cases_fc " +
 "GROUP BY service_name " +
 "ORDER BY service_name " +
 "ASC;"
 };


}
```

Let's go through these one by one. The constructor and super calls are required. As it says in the react documentation: "The constructor for a React component is called before it is mounted. When implementing the constructor for a `React.Component` subclass, you should call `super(props)` before any other statement. Otherwise, `this.props` will be undefined in the constructor, which can lead to bugs."

Next, we have state:

- localJson: this is an object into which we will pass JSON we receive back from an API call
- serviceNameCountNumbers: this is an array into which we will put the count associated with each service type. For example: the number of calls about Abandoned Bikes, Building Construction…
- serviceNameCountNames: this is an array into which we will put the name associated with each service type. For example: Abandoned Bikes, Building Construction…
- backgroundColors: this is an array into which we'll do a clever trick. We will create a random color for each service type and save that data here. This will add some color to the graph.
- URLextension: this is the the string extension we will combine with app.js's this.state.baseURL in order to query the API.

Next, we will make a function called getData that is quite similar to the API call we made in App.js. We haven't yet passed in the baseURL, but we will further down in the process. As you see below, we are correctly calling setState to pass the received data into

this.state.localJson

```
getData(){
 var tempURL = this.props.baseURL + this.state.URLextension;
 ///beeeeeeeeautiful compact async call
 (async() => {
 try {
 var response = await fetch(tempURL);
 var data = await response.json();

this.setState({localJson: data});
 this.formatDataForChart();

} catch (e) {
 console.log("error")
 }
 })();
 }
```

You'll see above we are calling a function titled formatDataForChart() that doesn't exist yet so let's make that. I will describe what this function does before pasting the code below. The first thing formatDataForChart will check is that this.state.Json did not receive null or empty data. This prevents a crash. Next, it creates 3 local arrays that we populate before then setting this.state equal to their data. It then loops through the received data and pushes the service name, count, and a randomly created color into the appropriate arrays. When this is complete, we call setState to assign the local arrays' data into the state data. Lastly, if the received data was null or empty, it sets the arrays to 0 and "no requests." This way, we can usefully display there were zero requests for this area, instead of a confusing response of "null."

```
formatDataForChart(){

if (this.state.localJson.rows.length > 0) {
```

```
var serviceCount = [];
 var serviceNames = [];
 var backColors = [];

//load the data into the appropriate state arrays
 for (var i = 0; i < this.state.localJson.total_rows; i++) {
 serviceNames.push(
 this.state.localJson.rows[i].service_name
 );
 serviceCount.push(
 this.state.localJson.rows[i].count,
 );
 backColors.push(
 this.getRandomColor()
 );
 }

this.setState({serviceNameCountNumbers: serviceCount});
 this.setState({serviceNameCountNames: serviceNames});
 this.setState({backgroundColors: backColors});
 }
 else if(this.state.localJson.rows.length ===0) {
 var serviceCountZero = [0];
 var serviceNamesZero = ['No Requests'];

this.setState({serviceNameCountNumbers: serviceCountZero});
 this.setState({serviceNameCountNames: serviceNamesZero});
 }//end else

}
```

Next, we will create a small class that generates random colors.

```
 //this random color generator is from: https://stackoverflow.com/questions/1484506/random-color-generator
 getRandomColor() {
 var letters = '0123456789ABCDEF';
 var color = '#';
 for (var i = 0; i < 6; i++) {
 color += letters[Math.floor(Math.random() * 16)];
 }
 return color;
 }
```

Now we will create a function called createChart. The format of this function comes from reading the documentation for react-chartjs-2

However, this is the first function we are creating that returns an HTML element. This is one of the fundamentals of react. As you'll see, the function has a return() method and inside of that is an HTML tag. In this case it's an HTML element with a lot of data, but it doesn't have to be. For example, from the react documentation:

```
function UserGreeting() {
  return <h1>Welcome back!</h1>;
}
```

Whenever that function UserGreeting is called, an h1 element with "Welcome back!"

Below is the code for our createChart function. We give it a className to apply CSS styling. *tip* in react, you must call it className and not classname.

```
createChart(){
 return (
 <div className="service_charts_by_year">
 <Doughnut
```

```
 ref='chart'
 data={{
 datasets: [{
 data: this.state.serviceNameCountNumbers,
 //label: this.props.yearToDisplay,
 backgroundColor: this.state.backgroundColors
 }],

// These labels appear in the legend and in the tooltips when hovering different
 arcs
 labels: this.state.serviceNameCountNames,
 }}
 options={{
 onClick:(evt, item) => {
 if(item.length>0) {
 this.props.handleServiceNameSelected(item[0]._model.label, item[0]._model.backg
roundColor);
 }
 },
 title: {
 display: true,
 fontSize: 18,
 text: "Click a service type below, then scroll down for more data."
 },
 legend: {
 onClick:(evt, item) => {
 this.props.handleServiceNameSelected(item.text, item.fillStyle);
 },
 display: true,
 position: "left",
 fullWidth: true,
 reverse: false,
 labels: {
```

```
 fontColor: '#262626'
 },
 },
 responsive: true,
 maintainAspectRatio: true
 }}

/>
 </div>
 );
 }//end createChart
```

Each class should have a render method itself. Since we instantiate this class in order to receive a chart, below is what we will use for this class:

```
 render() {
 return(
 this.createChart()
 );
 }
```

Lastly, we want this class to call the Philly 311 API on instantiation and return data. So, we need to use the factory method componentWillMount(). Create this function in the class. I like to put this near the top of the class after the constructor so I see it easily, but that placement is not required.

```
 componentWillMount(){
 this.getData();
 }
```

Now, we need to go back to app.js to address a few things to make this class work. We need to pass in the properties this class expects. If you search the class for props you'll find:

- this.props.baseURL
- this.props.handleServiceNameSelected(item[0]._model.label, item[0]._model.backgroundColor)
- this.props.handleServiceNameSelected(item.text, item.fillStyle)

Let's reopen app.js. At the top of app.js we need to import the class we just created. Add a new import statement:

```
import ServiceNameChart from './service_name_data';
```

In app.js we need prepare some things to work with our new chart.

1. We need to create a function to handle when a user clicks on a particular service name in our chart
2. We want to keep track in this app.js class file of what service name was clicked on
3. We want to keep track in this app.js class file of what color was created for that service name so we can use it elsewhere so the design looks consistent.

First, add two more variables to app.js's state.

```
service_name: " ",
chartColor: "",
```

Next, create a function called handleServiceNameSelected.

```
 handleServiceNameSelected(item, itemColor) {
 this.setState({service_name: item});
 this.setState({chartColor: itemColor});
 }
```

Now, we need to bind this handleServiceNameSelected method to our state. Use the following syntax in app.js's constructor, after this.state is complete:

```
this.handleServiceNameSelected = this.handleServiceNameSelected.bind(this);
```

At this point, app.js's state should look like this:

```
constructor(){
super();
this.state= {
service_name: " ",
chartColor: "",
yearToDisplay: 2014,
baseURL: "https://data.phila.gov/carto/api/v2/sql?q=SELECT",
totalRequests: 0
};
this.handleServiceNameSelected = this.handleServiceNameSelected.bind(this);
}
```

Ok! Let's try it out. In app.js's return method, we instantiate our new class like :

```
<ServiceNameChart
baseURL={this.state.baseURL}
handleServiceNameSelected={this.handleServiceNameSelected}
/>
```

Above, this is calling our ServiceNameChart class to instantiate it. The code before the closing tag (/>) is what is referred to as props. We are passing in two properties: baseURL and handleServiceNameSelected. It's important to note that there is no code making sure you are passing in something that makes sense. It's up to you to pass in what you want and call it what you want.

Now, where you place <ServiceNameChart… in the app.js return method will determine where it ends up on the website screen. I have it below our previous <h1>Philly 311 has handled… so it will show up below that. It's up to you to play around if you want it to be

above or below that.

tip there's a specific syntax required that I ran into a lot. The app's return method must wrap everything in a div. You can nest as many divs as you want inside, but it all does need to be inside one overall div.

Your site should load and show a graph like this. (colors may vary)

Archive 3 of the code at this point: https://drive.google.com/open?id=1nUWbqnhG3dRjCgwQwah3ovx15mUMRJA2

Year Chart

Next we will build a different chart to display data year by year. This will be a bar chart that receives the name of a service type, queries the Philly 311 API, and displays the total requests for that service type for each year.

As we did before, create a new file and title it src/yearChartData.js

Put the following import statements at the top of the file:

```
import React from 'react';
import './App.css';
import YearChart from './yearChart';
```

You might notice this class file is importing a class file called YearChart. We will build that a bit further down.

Create the class declaration like this:

```
class YearChartData extends React.Component {


} //end class


export default YearChartData;
```

Let's create this class's constructor method

```
    constructor(props) {
        super(props);
        this.state = {
            service_name: this.props.service_name,
            yearsOfData: this.props.yearsOfData,
            YearData:[0, 0, 0, 0],
            baseURL: "https://data.phila.gov/carto/api/v2/sql?q=",
            URLextension: "SELECT count(*) as NUM FROM public_cases_fc WHERE EXT
RACT(YEAR FROM requested_datetime)="
        };
    }
```

You can see that the following will be received properties, since they are referenced with this.props…

- yearsOfData
- service_name

Again, we see baseURL and a URLextension which are put together as part of this class's API call. There is one new state variable: YearData:[0, 0, 0, 0]. This array is where we will put the data received back from the API call. It is initialized to 0 as a preventative measure. In the API call, if no data is received back for a particular year, we can not setState for that array element, and this way still supply 0 instead of NULL to the chart.

Next, let's create the function to call the API:

```
getData(newServiceNameToSearch, yearToSearch, arrayIndexPoint) {

    var tempString = newServiceNameToSearch;
    var stringWithoutApos = tempString.replace("'s", "''s");

    tempString = stringWithoutApos.replace('&',  ' || & || ');

    var queryURL = this.state.baseURL;

    var tempURL = this.state.URLextension
        + yearToSearch
        + " AND service_name='"
        + tempString
        + "'"
    ;

    queryURL = queryURL.concat(tempURL);

    ///beeeeeeeeautiful compact async call
    (async() => {
        try {
            var response = await fetch(queryURL);
            var data = await response.json();

            let tempArray = this.state.YearData;
            tempArray[arrayIndexPoint] = data.rows[0].num;
            this.setState({YearData: tempArray});


        } catch (e) {
            let tempArray = this.state.YearData;
            tempArray[arrayIndexPoint] = 0;
            this.setState({YearData: tempArray});
```

```
                console.log("error")
            }
        })();
    } // end getData
```

This looks similar to the function we created for service_name_data.js, however, it takes 3 parameters:

1. newServiceNameToSearch – this will be the String that is passed in from app.js
2. yearToSearch – we will actually make 4 calls to getData and each time pass in the year to search with
3. arrayIndexPoint – this will be the index point for where we want to put the received data into this.state.YearData

Now let's create a small function that takes a "service_name" and calls getData for each year in this.state.yearsOfData.

tip when we instantiate this class, we will pass in an array that contains 2014, 2015, 2016, 2017. So, this will make a little more sense:

```
    //method to call the api and get the number.
    getYearDataForChart(service_name){

        for (var i =0; i<this.state.yearsOfData.length; i++){
            this.getData(service_name, this.state.yearsOfData[i], i);
        }
    }
```

Next, we will use a factory methods: componentWillReceiveProps. As you see in the documentation, if we want a class to re-render with updated information, we should begin that within componentWillReceiveProps.

tip just calling setState in some other function with new data and expecting that to trigger a re-render will not work. From the documentation: "`componentWillReceiveProps()` is

invoked before a mounted component receives new props. If you need to update the state in response to prop changes (for example, to reset it), you may compare `this.props` and `nextProps` and perform state transitions using `this.setState()` in this method."

```
componentWillReceiveProps(nextProps) {
    //setState with new properties;
    this.setState({service_name : nextProps.service_name})
    //call getData with the passed in data
    this.getYearDataForChart(nextProps.service_name);
}
```

What the function above does is set the state to the newly received service_name. Then, it calls our getYearDataForChart and passes in the newly received service_name.

Now, we will create the render method. We will instantiate a class we haven't made yet, and pass it what it will need:

```
render() {
    return <YearChart
            handleYearSelect={this.props.handleYearSelect}
            YearData={this.state.YearData}
            service_name={this.state.service_name}
            yearsOfData = {this.state.yearsOfData}
            chartColor={this.props.chartColor}
        />;
}
```

As you see, when this yearChartData class is instantiated, it returns an instantiation of YearChart. In this way, you can nest return statements to include other classes.

You also see above that we are passing from yearChartData to YearChart two props that

are not used in yearChartData. We are just passing them along: this.props.handleYearSelect and this.props.chartColor. We already discussed chartColor, but handleYearSelect is new.

Let's now create the class that will return YearChart.

Create a new file called src/yearChart.js and put the following import statements and class declaration:

```
import React from 'react';
import { Chart } from 'react-chartjs-2';
import './App.css';
import {Bar} from 'react-chartjs-2';


class YearChart extends React.Component {


} //end class


export default YearChart;
```

This class only needs two functions: one to create the chart, and one render function.

The code below is in the format that chart.js requires to create the bar chart:

```
createChart(){


      return (
          <div className="yearchart">
              <Bar
                  ref='chart'


                  data={{
                      datasets: [{
```

```
                    data: this.props.YearData,
                    label: this.props.service_name,
                    backgroundColor: this.props.chartColor,
                }],

                // These labels appear in the legend and in the tooltips w
hen hovering different arcs
                labels: this.props.yearsOfData,
            }} redraw
            options={{
                onClick:(evt, item) => {
                    if(item.length>0) {
                        this.props.handleYearSelect(item[0]._model.label);
                    }
                },
                title: {
                    display: true,
                    fontSize: 18,
                    text: "Distribution of requests related to "
                    + this.props.service_name
                    + " in all "
                    + (this.props.yearsOfData.length)
                    + " years."
                    +" Click a year below for month by month data."
                },
                responsive: true,
                maintainAspectRatio: false,
                legend: {
                    display: true,
                    position: "top",
                    fullWidth: true,
                    reverse: false,
                    labels: {
```

```
                            fontColor: '#262626'
                        },
                    },
                }}

            />
        </div>
    );
}
```

As you see above, the createchart function is using the properties we pass in from yearChartData to create the chart. Lastly, we need to put in the render function. Also this required function does is to call createchart, which returns the chart wrapped in a .

```
    render() {
        return(this.createChart());
    }
```

Now, in order to supply everything this class needs, we need to create two things in app.js

1. a function in app.js. Open up your app.js file. What we want is to know when a user clicks on a year in our YearChart bar graph. So, we need a way for that graph to relay back to it's parent what was clicked. The way to do this is to create a function to receive and handle that information.
2. a state variable called yearsOfData that is initialized to: [2014, 2015, 2016, 2017]

In app.js, create the following function:

```
    handleYearSelect(item) {
        this.setState({yearToSearch: item});
    }
```

Then, in app.js's contructor, after the state brackets, paste the following:

```
this.handleYearSelect = this.handleYearSelect.bind(this);
```

tip the way this works is that we pass this handleYearSelect function as a property to yearChartData, which then passes it as a property to YearChart. You can see it around line 30 in yearChart.js. It is called as an onClick function. You see it is passing back "item[0]._model.label" which, in this case, is way to find out what year was clicked.

In app.js's state add the following variable:

```
yearsOfData: [2014, 2015, 2016, 2017],
```

Finally, we want to try all this out and see if it shows up. If you remember, the only way to have elements show up on screen is to instantiate then. In app.js's render function, instantiate your YearChartData class and pass in all the required properties:
tip remember that wherever you put this instantiation, it needs to be within the overall in app.js's return method.

```
<YearChartData
    service_name={this.state.service_name}
    yearsOfData={this.state.yearsOfData}
    handleYearSelect={this.handleYearSelect}
    chartColor={this.state.chartColor}
/>
```

Go ahead and try that out. And… it didn't work. I forgot that we need to import these class files so they are available to app.js. At the top of app.js, we need to import our yearChartData class.

```
import YearChartData from './yearChartData';
```

We don't need to import the YearChart class because that doesn't need to be available to app.js. It needs to be available to yearChartData and we already imported it there.

At this point, your page should load a Doughnut chart and an empty bar chart. If you click on part of the bar chart or an icon the legend of the Doughnut chart, the Year chart should begin to make calls to the API and display bars. You will see a flickering effect as each call returns, alters that class's data, and the DOM is redrawn. I would like a way to eliminate that flickering, but I haven't found it yet.

Archive 4 of the project at this point: https://drive.google.com/open?id=1ABOjnqxWTtbD-JUL8z4K0c6cWYqXex2C

Month Chart

Before we go further, let's fix the css a bit. Delete the code in the app.css file and paste in the following css

```
.App {
    text-align: center;
}

.App-logo {
    animation: App-logo-spin infinite 20s linear;
    height: 80px;
}

.App-header {
    background-color: #222;
    height: 150px;
    padding: 20px;
    color: white;
}

.App-intro {
```

```
    font-size: large;
}

@keyframes App-logo-spin {
    from { transform: rotate(0deg); }
    to { transform: rotate(360deg); }
}



.agencylist {
    background-color:#F5F5F5;
    border: 2px solid;
    border-radius: 10px;
    border-color:#79bbff;
    padding:5px;
    list-style:none;
    margin: 5px 5px 5px 5px;

}

.agencyselect {
    border-top: 2px solid;
    border-bottom: 2px solid;
    border-color:#79bbff;
    padding:5px;
    list-style:none;
    margin: 5px 0px 5px 0px;
}



.color:hover
{
    background:#53a7ea;
```

```
}



.agency {
    margin: auto;
}

h2 {
    color:#004d4d;
}



.myButton {
    background-color:#79bbff;
    -moz-border-radius:6px;
    -webkit-border-radius:6px;
    border-radius:6px;
    border:1px solid #84bbf3;
    display:inline-block;
    cursor:pointer;
    color:#ffffff;
    font-family:Arial;
    font-size:15px;
    font-weight:bold;
    padding:6px 24px;
    text-decoration:none;
    margin: 1px 1px 1px 1px;
}
.myButton:hover {
    background-color:#378de5;
}
.myButton:active {
    position:relative;
```

```
    top:1px;
}

.yearchart {
    width:100%;
    min-height: 300px;
    margin: 0 auto;
}

.monthchart {
    width:100%;
    min-height: 300px;
    margin: 0 auto;
}

.service-chart-title{
    text-align: left;
}

.service-type-container{
    border-bottom: dotted;
    border-bottom-color: darkgray;
}
```

For our last major element we will create two classes that display a bar chart showing month by month data. When a user clicks on a service name in the doughnut chart, then a year in the bar chart, we will pass that year to a month chart which will show how many requests there were for each month of a particular year.

Start by creating two new classes:

1. src/monthChartData.js
2. src/monthChart.js

I will paste below the code for this class. You will see a few differences from the yearChartData.js class. For this class, we have a variable called “monthsOfData” which contains an array of months to use in our API call. Then, in the function called “getMonthDataForChart” we use that month array to make 12 calls to the API. Another difference is that we do not have an onClick function for this chart. If you develop this app further you can choose to add an onClick function to do something when the user clicks on a month.

```
import React from 'react';
import './App.css';
import MonthChart from './monthChart';

class MonthChartData extends React.Component {

    constructor(props) {
        super(props);
        this.state = {
            service_name: this.props.service_name,
            monthsOfData: ['January', 'February', 'March', 'April', 'May', 'June
', 'July', 'August', 'September', 'October', 'November', 'December'],
            MonthData: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            baseURL: "https://data.phila.gov/carto/api/v2/sql?q=",
            URLextension: "SELECT count(*) as NUM FROM public_cases_fc WHERE EXT
RACT(MONTH FROM requested_datetime)="
        }
    }

    componentWillReceiveProps(nextProps) {
        this.setState({service_name : nextProps.service_name});

        //call getData with the passed in data
        this.getMonthDataForChart(nextProps.service_name, nextProps.yearToSearch
);
```

```
    }

    //method to call the api and get the number.
    getMonthDataForChart(service_name, yearToSearch){
        //query 12 times, one for each month
        if(yearToSearch>0){
            for (var i =1; i<=12; i++){
                this.getData(service_name, i, yearToSearch);
            }
        }
        else {
            //reset the array to draw 0
            var tempArray=[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
            this.setState({MonthData: tempArray});
        }
    }
```

```
    getData(newServiceNameToSearch, monthNumberToSearch, yearToSearch) {

        var tempString = newServiceNameToSearch;
        var stringWithoutApos = tempString.replace("'s", "''s");
        tempString = stringWithoutApos.replace('&',  ' || & || ');
        var queryURL = this.state.baseURL; //todo fix this with new query

        var tempURL = this.state.URLextension
            + monthNumberToSearch
            + "AND EXTRACT(YEAR FROM requested_datetime)="
            + yearToSearch
            + "AND service_name='"
            + tempString
            + "'"
            ;
        queryURL = queryURL.concat(tempURL);
```

```
    ///beeeeeeeeautiful compact async call
    (async() => {
        try {
            var response = await fetch(queryURL);
            var data = await response.json();
            let tempArray = this.state.MonthData;
            tempArray[monthNumberToSearch-1] = data.rows[0].num;
            this.setState({MonthData: tempArray});
        } catch (e) {
            let tempArray = this.state.MonthData;
            tempArray[monthNumberToSearch-1] = 0;
            this.setState({MonthData: tempArray});
            console.log("error")
        }
    })();
} // end getData


    render() {
        return <MonthChart
            MonthData={this.state.MonthData}
            service_name={this.state.service_name}
            monthsOfData = {this.state.monthsOfData}
            chartColor = {this.props.chartColor}
            yearToSearch = {this.props.yearToSearch}
            />;
    }
} //end class

export default MonthChartData;
```

Our monthChart class is also nearly identical to the yearChart class. I will put the code below:

```
import React from 'react';
import { Chart } from 'react-chartjs-2';
import './App.css';
import {Line} from 'react-chartjs-2';

class MonthChart extends React.Component {

    createChart(){
        return (
            <div className="monthchart">
                <Line
                    ref='chart'
                    data={{
                        datasets: [{
                            data: this.props.MonthData,
                            label: this.props.service_name + " in " + this.props
.yearToSearch,
                            backgroundColor: this.props.chartColor,
                        }],
                        // These labels appear in the legend and in the tooltips
 when hovering different arcs
                        labels: this.props.monthsOfData,
                    }} redraw
                    options={{
                        title: {
                            display: true,
                            fontSize: 18,
                            text: "Distribution of requests related to "
                            + this.props.service_name
                            + " in the months of "
                            + this.props.yearToSearch
                            + "."
                        },
```

```
                    responsive: true,
                    maintainAspectRatio: false,
                    legend: {
                        title: this.props.yearToSearch,
                        display: true,
                        position: "top",
                        fullWidth: true,
                        reverse: false,
                        labels: {
                            fontColor: '#262626'
                        },
                    },
                }}
            />
        </div>
    );
}

    render() {
        return(this.createChart());
    }
} //end class

export default MonthChart;
```

As before, we need to go back to app.js and import our monthChartData class file as we did for yearChartData. Add the following import statement to app.js:

```
import MonthChartData from './monthChartData';
```

Also add the following variable to app.js's state:

```
yearToSearch: 2014,
```

Let's try it out! We can instantiate the class as before in app.js's render function. *tip* remember to put this within the overall that app.js returns.

```
<MonthChartData
service_name={this.state.service_name}
yearToSearch={this.state.yearToSearch}
chartColor={this.state.chartColor}
/>
```

Lookin' good. At this point, the app should look like this:

Archive 5 of the app at this point: https://drive.google.com/open?id=1ONMLW46tpF8v5JCASxf6r4pg5Xoe5byf

Final Small Changes

At this point I have a few small things I'd like to clean up. When you first load the app, above the year and month charts, the text reads: "Distribution of requests…" but there is no data displayed. I want to change to show more useful text if no service type, or year, have been clicked yet.

If we open yearChart.js add some code in the createChart function to check the state of this.props.service_name. If the value of it is still " ", indicating no service name has been clicked, we can display different text. Enter the following in the createChart function before the return statement:

```
var defaultText = "Choose a Service Type above"
var dataText = "Distribution of requests related to "
                    + this.props.service_name
```

```
                    + " in all "
                    + (this.props.yearsOfData.length)
                    + " years."
                    +" Click a year below for month by month data."
    var textToUse = defaultText


    switch(this.props.service_name === " ") {
        case true:
            textToUse = defaultText
            break;
        case false:
            textToUse = dataText
            break;
        default:
            textToUse = dataText
    }
```

Then, still in yearChart.js, within the “title” section, change text: to this:

```
title: textToUse,
```

For example, you can see this change in the gif below:

We can do the same in monthChart.js. Add the following code in monthChart.js in the createChart function, and before the return method:

```
    var defaultText = "Choose a Year above"
    var dataText = "Distribution of requests related to "
                        + this.props.service_name
                        + " in the months of "
                        + this.props.yearToSearch
                        + "."
```

```
    var textToUse = defaultText

    var defaultLabel = ""
    var dataLabel = this.props.service_name + " in " + this.props.yearToSearch
    var labelToUse = defaultLabel

    switch(this.props.yearToSearch == 0) {
        case true:
            textToUse = defaultText
            labelToUse = defaultLabel
            break;
        case false:
            textToUse = dataText
            labelToUse = dataLabel
            break;
        default:
            textToUse = dataText
    }
```

In the same way as above, change the text within the title section to this:

```
title: {
   display: true,
   fontSize: 18,
   text: textToUse
   },
```

Within the “datasets” section, change label: to this:

```
label: labelToUse,
```

So now, when we load the page but haven’t yet clicked on anything, the page should look like this. (your colors will vary)

One last change! You'll notice when you load the website it says "Welcome to React" in an <h2> tag. Change that to an <h3> tag with different text. I suggest this:

```
<h3>React App + Philly 311</h3>
```

Conclusion

Here is the final archive 6 of the code we created: https://drive.google.com/open?id=1sNUGHSi_0tlQrU6-gdl1gqVHF-KQjqQK

This final project should be the same as what you find at the original site: tps://philly-311-react.glitch.me/
I cleaned up, and improved the code as I recreated it here.

If you get stuck, I found the react documentation is excellent and helpful.

After learning React to create this site, my impression is that's best for quickly rendering many elements to the screen. If you have a less complicated site that isn't fetching data, React may be overkill.

good luck,
Bill Moriarty