LogRocket
Frontend Monitoring
FREE TRIAL
BLOG

Using D3.js with React: A complete guide

September 4, 2018 · 6 min read

Editor's note: *This React D3 tutorial was last updated in November 2020. For a closer look at the most recent stable release, check out our guide to using D3.js v6 with React.*

React and D3.js are JavaScript libraries that enable developers to create engaging, reusable data visualizations such as area charts, line graphs, bubble plots, and so much more.

Although React and D3.js is an extremely popular pairing among frontend developers, the two libraries can be challenging to use in tandem. In this tutorial, we'll show you how to use D3.js in React, discuss why you should use D3, and demonstrate how to make a chart in React with D3.

We'll cover the following in detail:

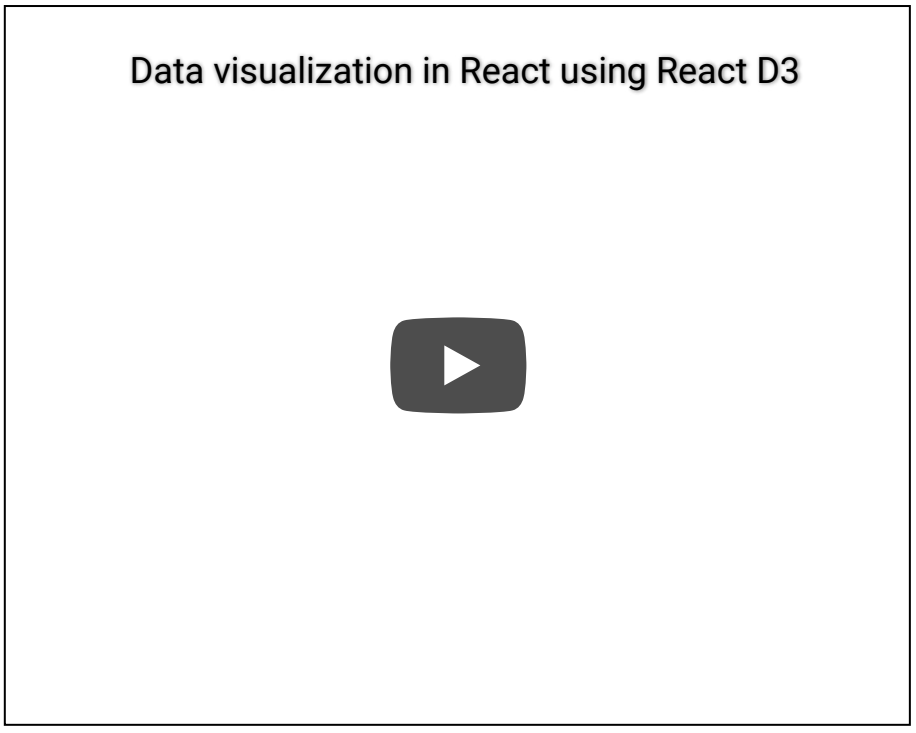
What are D3.js and React?

D3.js is a JavaScript library for creating dynamic, interactive data visualizations using HTML, CSS, and SVG. It binds data to the DOM and its elements, enabling you to manipulate visualizations by changing the data. The most recent release is D3.js v6.

While its focus on web standards enables you to harness the full capabilities of modern browsers without restricting yourself to a proprietary framework, D3.js and React are commonly used together to bring dynamic data visualizations to life.

React.js is an open-source frontend JavaScript library for building intricate user interfaces and UI components. It is a declarative, efficient, and flexible framework that allows you to build complex UIs composed of simple, reusable components. These components are capable of maintaining their state by themselves.

For a visual guide to help you get started with React and D3.js, check out the video tutorial below.



Why you should use D3.js

Data visualization helps you communicate information clearly and efficiently using shapes, lines, and colors. There are many tools available on the web, but D3.js has won the confidence of countless frontend developers, making it the de facto choice for data visualization in JavaScript.

D3.js is lightning-fast and supports large datasets and dynamic behaviors, enabling you to foster user interaction using animations and other eye-catching features.

How to use D3.js in React

D3.js and React can be challenging to use together because both libraries want to handle the DOM. They both take control of UI elements and they do so in different ways.

We'll show you how to get the most out of React and D3's distinct advantages by building a simple bar chart using the two libraries. First, we need to install React and D3.js.

Setting up React

To set up React, use the Create React App boilerplate. Run the following to install it globally on your local machine so that it can be reused:

```
npm install -g create-react-app
```

Next, create a new app using the `create-react-app` template:

```
create-react-app react-d3
```

Change the directory in the newly created project:

```
cd react-d3
```

Setting up D3.js

You can add the D3 library to your app either using the CDN or by installing via npm, as shown below:

```
npm install d3
```

Now, we are all set to start using D3.js to make data visualization in React.

To preview the app just created on your default browser, run the code below:

```
npm start
```

How to make a chart in React with D3.js

Open the created project with your favorite text editor and navigate to `src/App.js`.

This is the component that is currently rendered in the browser. We need to remove the content of the `render()` method so we can replace that with our own content.

In the `src` folder, create a new JavaScript file named `BarChart.js`. This is where we'll build the bar chart that will be rendered.

Setting up a bar chart with D3 and React

To start, add the following code to the file:

```
import React, {Component} from 'react';
import * as d3 from "d3";

class BarChart extends Component {

}

export default BarChart;
```

We'll use the `ComponentDidMount` lifecycle method to display the bar chart when the `BarChart` component is mounted in the DOM.

Add the following to the `BarChart` component:

```
class BarChart extends Component {

  componentDidMount() {
    this.drawChart();
  }
}
```

The `drawChart` is the method where we'll do all of our D3 magic.

Normally, when using D3.js without React, you do not have to put your D3.js code in a method, but this is important in React to ensure that the chart displays only when the component has been mounted on the DOM

Next, we'll create the `drawChart` method:

```
drawChart() {

  const data = [12, 5, 6, 6, 9, 10];

  const svg = d3.select("body").append("svg").attr("width", 700).attr("height",
300);

}
```

What's going on here?

First, we defined a variable, `data`, which contains the data we want to visualize.

Next, we defined an SVG using D3.js methods. We're using SVG because it's scalable — that is, no matter how large the screen is or how much you zoom in to view the data, it will never appear pixelated.

`d3.select()` is used to select an HTML element from the document. It selects the first element that matches the argument passed and creates a node for it.

In this case, we passed the `body` element, which we'll change later to make the component more reusable.

The `append()` method appends an HTML node to the selected item and returns a handle to that node.

The `attr` method is used to add attributes to the element. This can be any attribute you would normally add to the HTML element, such as `class`, `height`, `width` or `fill`.

We then appended an SVG element to the `body` element with a `width: 700` and `height: 300`.

Under the SVG variable we created, add the following code:

```
svg.selectAll("rect").data(data).enter().append("rect")
```

Just like the `select` method, `selectAll()` selects the element that matches the argument that is passed to it. That way, all elements that match the arguments are selected, not just the first.

Next, the `data()` method, is used to attach the data passed as an argument to the selected HTML elements.

Usually, these elements are not found because most visualizations deal with dynamic data and it is nearly impossible to estimate the amount of data that will be represented.

The `enter()` method rescues us from that bottleneck as it is used alongside the `append` method to create the nodes that are missing and still visualize the data.

Visualizing the data

So far we have created nodes for each data point. All that's left is to make it visible.

To make it visible we need to create a bar for each of those datasets, set a width and update the height of each bar dynamically.

The `attr` method allows us to use a callback function to deal with the dynamic data:

```
selection.attr("property", (d, i) => {})
```

`d` represents the data point value and `i` is the index of the data point of the array.

First, we need to set each data point at a specific point on the x- and y-axes of the bar chart. We use the `x` and `y` attributes to achieve this, where `x` represents the position of the bar along the x-axis (horizontally) and `y` represents the position of the bar along the y-axis.

Also, we need to set the width and height of each data point. The width of each data point is constant since the bars would be of the same width.

The height, on the other hand, depends on the value of each data point. We have to use the callback function to make the bar chart display the value of each data point.

We modify our SVG variable to become:

```
svg.selectAll("rect")
  .data(data)
  .enter()
  .append("rect")
  .attr("x", (d, i) => i * 70)
  .attr("y", 0)
  .attr("width", 25)
  .attr("height", (d, i) => d)
  .attr("fill", "green");
```

For the `x`, each index of the data point in the array is multiplied by a constant integer, `70`, to shift the position of each bar by 70.

`y` has a constant value, which we'll change soon.

The width also has a constant value of 65, which is less than the position of each element on the chart, to create a space between each element.

The height of the bar depends on the value of each entry in the data set.

Manipulating data with D3.js and React

Now we have created a bar chart. However, we have two issues:

1. The bars in the chart are small
2. The chart is also inverted

To resolve these issues, we'll multiply each datum by a constant value of, say, 10 to increase the size of each bar without affecting the data:

```
.attr("height", (d, i) => d * 10)
```

Next, we solve the issue of the bar being inverted, but before that let's understand why the chart is inverted in the first place.

The SVG position goes from top to bottom, so using a `y` attribute of `0` puts each bar at the top edge of the SVG element.

To fix this, subtract the height of each bar from the height of the SVG element:

```
.attr("y", (d, i) => h - 10 * d)
```

`(10 * d)` is the height we got from our previous calculation.

Putting it all together, the `BarChart` component will be:

```
  }

  drawChart() {
    const data = [12, 5, 6, 6, 9, 10];

    const svg = d3.select("body")
    .append("svg")
    .attr("width", w)
    .attr("height", h)
    .style("margin-left", 100);

    svg.selectAll("rect")
      .data(data)
      .enter()
      .append("rect")
      .attr("x", (d, i) => i * 70)
      .attr("y", (d, i) => h - 10 * d)
      .attr("width", 65)
      .attr("height", (d, i) => d * 10)
      .attr("fill", "green")
```

Bar Chart in it's glory

We now have a basic bar chart. Let's do a little extra and add labels.

Adding labels to a bar chart

To add labels, add the following code to the `drawChart` function:

```
svg.selectAll("text")
  .data(data)
  .enter()
  .append("text")
  .text((d) => d)
  .attr("x", (d, i) => i * 70)
  .attr("y", (d, i) => h - (10 * d) - 3)
```

This is similar to what we did for the bars but this time, `text` is appended instead.

The bar chart should now look like this:

How to make a chart reusable in React and D3.js

One of the important principles of React is to make components that are reusable.

To do this, we need to remove the provided data and then pass it to the component through props.

The width and height of the SVG will also be passed via props:

```
const data = [12, 5, 6, 6, 9, 10];
```

The above becomes:

```
const data = this.props.data;
```

And the width and height attribute change from:

```
const svg = d3.select("body").append("svg").attr("width", 700).attr("height",
300);
```

To:

```
const svg = d3.select("body").append("svg")
  .attr("width", this.props.width)
  .attr("height", this.props.height);
```

In our `App.js` file, we can now use the component and pass the data we want from the parent component.

```
class App extends Component {

  state = {
    data: [12, 5, 6, 6, 9, 10],
    width: 700,
    height: 500,
    id: root
  }

  render() {
    return (
      <div className="App">
        <BarChart data={this.state.data} width={this.state.width} height=
{this.state.height} />
      </div>
    );
  }
}
```

This way, we can reuse the bar chart anywhere we want in our React app.

Full visibility into production React apps

Debugging React applications can be difficult, especially when users experience issues that are difficult to reproduce. If you're interested in monitoring and tracking Redux state, automatically surfacing JavaScript errors, and tracking slow network requests and component load time, try LogRocket.

LogRocket is like a DVR for web apps, recording literally everything that happens on your React app. Instead of guessing why problems happen, you can aggregate and report on what state your application was in when an issue occurred. LogRocket also monitors your app's performance, reporting with metrics like client CPU load, client memory usage, and more.

The LogRocket Redux middleware package adds an extra layer of visibility into your user sessions. LogRocket logs all actions and state from your Redux stores.

Modernize how you debug your React apps — start monitoring for free.

Share this:

Twitter Reddit LinkedIn Facebook

Nosa Obaseki Follow

Front-end Dev currently building amazing products @theflutterwave 🐼🇳🇬

#react

11 Replies to "Using D3.js with React: A complete guide"

Nicole Says:
July 22, 2019 at 11:37 am

Reply

at “Putting it all together, the BarChart component will be:” ‘h’ and ‘w’ are undefined. I added “const h = 500; const x= 400;” under const data = [12, 5, 6, 6, 9, 10], and it worked. Overall, thanks for the tutorial it really helped guide my d3 work in react app.

Rich Goldman Says:
July 25, 2019 at 5:20 pm

Reply

this tutorial is broken and incomplete

BeBest Says:
August 8, 2019 at 2:57 am

Reply

This was very hepful. Thanks. But also, it would help if you posted the full working code on github or something. Also, can you explain why you used app.js to call drawChart instead of just rendering everyting in drawChart?

Ling Says:
August 12, 2019 at 3:49 pm

Reply

When I hook my app.js to a fetch, the chart doesn’t render. The initial call to componentdidmount for the chart gets no data, but when the data arrives and I set a new state, the new render has no effect on calling drawchart again. How would you recommend getting around this?

Paul Says:
August 30, 2019 at 9:46 pm

Reply

// Had to fix a few bugs. One possibility for working code below...

// App.js:

```
import React from ‘react’;
import BarChart from ‘./BarChart.js’;
import ‘./App.css’;
```

```
class App extends React.Component {

state = {
data: [12, 5, 6, 6, 9, 10],
width: 700,
height: 500,
id: “root”
}

render() {
return (

);
}
}

export default App;

// BarChart.js:

import React from ‘react’;
import * as d3 from “d3”;

class BarChart extends React.Component {
componentDidMount() {
this.drawChart();
}

drawChart() {
const data = this.props.data;
const svg = d3.select(“body”).append(“svg”)
.attr(“width”, this.props.width)
.attr(“height”, this.props.height);

const h = this.props.height;

svg.selectAll(“rect”)
.data(data)
.enter()
.append(“rect”)
.attr(“x”, (d, i) => i * 70)
.attr(“y”, (d, i) => h – 10 * d)
.attr(“width”, 25)
.attr(“height”, (d, i) => d * 10)
.attr(“fill”, “green”);
```

```
svg.selectAll(“text”)
.data(data)
.enter()
.append(“text”)
.text((d) => d)
.attr(“x”, (d, i) => i * 70)
.attr(“y”, (d, i) => h – (10 * d) – 3)

//selection.attr(“property”, (d, i) => {})
}

render(){
return
}

}

export default BarChart;
```

Rickard Peter Says:
September 30, 2019 at 11:38 pm

Reply

I can understand the article, but that is because i already know d3 and react. Please look at those suggestions above me and fix this article accordingly, because i think newbies will find this buggy and hard to follow, if just not working overall

Saravananselvamohan Says:
December 22, 2019 at 12:01 pm

Reply

Hi I just made some minor changes in the code and its working for me. Please find the code at https://github.com/saravananselvamohan/ReactD3.git

TylerPeters Says:
December 4, 2020 at 5:00 am

Reply

As a D3 noob, I appreciate any tutorial, but this article makes giant leaps over huge gaps.
A. After following the npm steps to install react, d3, and create the basic app

scaffolding, there is no Render() method in /src/App.js.
B. After walking us through creating BarChart.js, the author does not explain how to reference BarChart.js from /src/App.js.

António Says:
December 23, 2020 at 9:43 am

Reply

a bit disappointing that you guys are still promoting the old class pattern in this article, don't think that's helping anyone at this point in time.

Matt Angelosanto Says:
February 8, 2021 at 8:58 am

Reply

This post was originally published several years ago, and we just updated it a few months back. We've added an editor's note to clarify. Thanks for keeping us honest.

Prabakar Says:
December 25, 2020 at 8:46 pm

Reply

Thank You Saravanan, your link to the github source helped.

Leave a Reply

Enter your comment here...