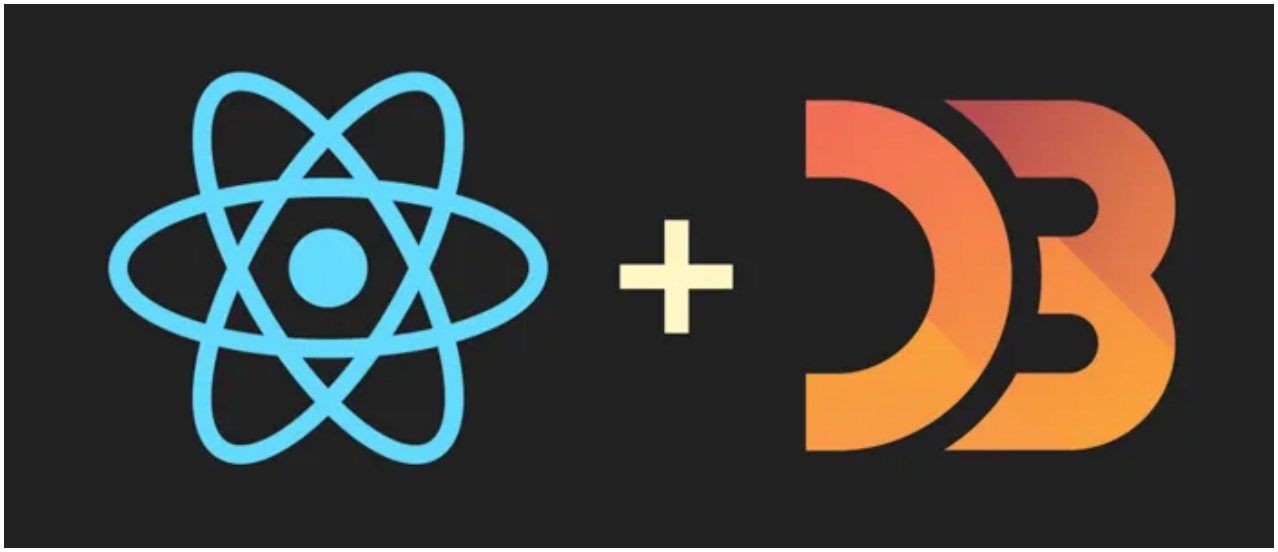
March 10, 2016 By Abhisek Jana — 30 Comments (Edit)

How to integrate React and D3 – The right way



React has been the choice for many developers in last year and will continue to grow in coming years. The main reason I liked React would be the portability of using it along with other frameworks specially in existing project. Other then the performance

improvement from virtual dom, React also excels in reusability and integration. In this **How to integrate React and D3 – The right way** tutorial we will take a look at integrating React with another very popular javascript component (d3.js), which is heavily used in visualization.

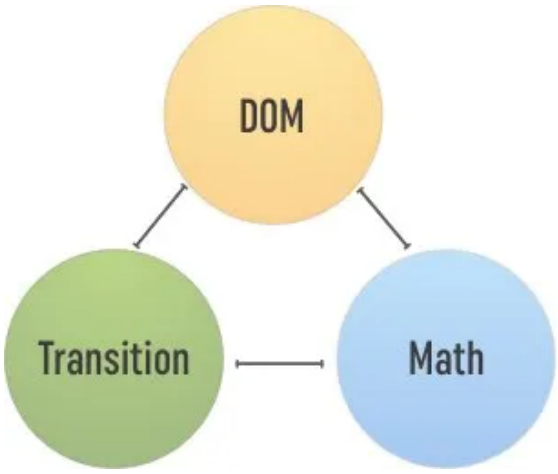
Why React + D3

The first question would be why we would need to even integrate React and D3. Even though D3 is a fantastic library it lacks in reusability and integration. You would need to put lots of additional effort in order to make a d3 chart reusable across your application. We will look into other advantages, but lets see how we can actually integrate React and D3.

How to integrate React + D3

At a high level D3 has following 3 functions:

1. DOM Update
2. Calculation (Math)
3. Transitions



When integrating React and D3, it would be best to have React to update the DOM (not D3). So **most** of the time we will use React to update the DOM however will be fully utilizing the Math functions provided by D3. We won't be able to use D3 transitions in React since we will have React update the DOM. In short, we will **not** be able to use `d3.select()` and `d3.transition()` from react.

Note: In very few scenarios we will still let D3 update the DOM since converting those functions in React could be overkilling.

Advantages of Integrating React and D3:

Let's come back to our why question again. Here are some facts, we will be covering them in our demo and example.

Reusability

As described earlier, the charts could be reused across the application without any additional coding. This would be a major reason for using React + D3

Update Chart

Updating the charts using React also will be very simple.We need to define the `props` and `state` properly in React to get the full benefit of this.We can also adapt **Flux Architecture** (We will talk about this in a future post) to extend the capabilities.

[the_ad id="763"]

Responsiveness

This is another very important functionality. Using React we can create responsive charts without even having to add any custom code for every chart. We will be using React's two way binding feature for this.

Shared Library/Structural Code/Maintainability/Extensibility/Scalability

Once you integrate React and D3 you would almost create a library of charts which can be easily shared and modified. You unit testing will also become easy. You can relate this to the Procedural vs Object Oriented Programming. D3 is like Procedural and React + D3 is like Object Oriented.

Disadvantages of Integrating React and D3:

D3 can be integrated with React by another way (Like any other library). Instead of let React update the DOM, you can get the current DOM node in React and use D3 to update the DOM. However this is **not** an efficient of way of integrating them together. Lets look at few Disadvantages of Integrating React and D3 below:

JSX

Since you will not be using D3 to update the DOM, you need to write equivalent JSX code to create the svg elements.

Transitions

D3 transitions functions **cannot** be used. However its probably best to avoid the Transitions altogether to start with, if they don't provide a real business value to your use case. You may try using React Animation however it will be a separate topic to discuss in future posts.Otherwise the only option is to use the core d3 library to perform the animation.

Enough of theory, let's look at the example of what we will be building. We will create a small dashboard using React and D3. You can find the live demo here.

Demo

The above charts are fully reusable and responsive. You can test by resizing the browser window. Click on the Donut or the Progress Chart to update the data as well. You can jump into the code right away, here is the code available in github.

View Code

Setup:

We will be using ES5/Babel and in browser compilation of JSX. You can refer my previous post on the setup of ES5 here.

Setup React JS Development Environment for ES5 and ES6

Setting up React JS Development can be sometimes confusing and time consuming. In this Setup React JS Development Environment for ES5 and ES6 tutorial we will go though some easier way of setting up the env. We have separate setup instructions for ES5 and ES5. The objective of this tutorial is to have you up ... Continue reading

A Developer Diary 5

JS Libraries:

We will be using following JS libraries, in addition to the setup files provided in the previous link (Setup React JS Development Environment for ES5 and ES6).

- bootstrap.css – We will use bootstrap grid for the layout.
- jquery.js (Version 2.2.1) – We will not use JQuery to update DOM, rather will use to add listener.
- d3.js

Index.html:

Here is the index.html. Find the full page in github. I am using bootstrap grid for the layout.

https://github.com/adeveloperdiary/react-d3-charts/blob/gh-pages/01_Visitor_Dashboard/index.html

app.jsx:

Below is the `app.jsx` file, we will be rendering three React components for the dashboard.

```
var Browser = React.createClass({
    render:function(){
        return (
```

Browser Share

```
        )
    }
});

var RetVisitors = React.createClass({
    render:function(){
        return (
```

Returning Visitors

```
        )
    }
});

var Visitors = React.createClass({
    render:function(){
        return (
```

Visitors to your site

```
        )
    }
});

ReactDOM.render(,document.getElementById("browser"));
ReactDOM.render(,document.getElementById("ret_visitors"));
ReactDOM.render(,document.getElementById("top-line-chart"));
```

Now lets look at the implementation one by one.

LineChart:

Here is the JSX code for the Line Chart. We have 1 Grid Component, 2 Axis Component, 1 path element which is the actual chart and Dot Component to create the circles for the data point.

Lets first start by creating just the line for the chart without the Axis, Grid and the Dots.

We will have the width,height and chartId defined as `propTypes`. Use the `getDefaultProps()` function to setup the default values. We will define the width in `this.state` since we want to use two way binding for the width. We will use this later to create responsive chart. So set the `this.state.width` to `this.props.width` in `getInitialState()` function.

[the_ad id="767"]

We will use fixed data set for time being. Create the `data[]` array. Then use the d3 math functions, `d3.time.scale()`, `d3.scale.linear()` and `d3.svg.line()` to create the xScale, yScale & line.

At the end, set `d={line(data)}` in the path element to create the chart.

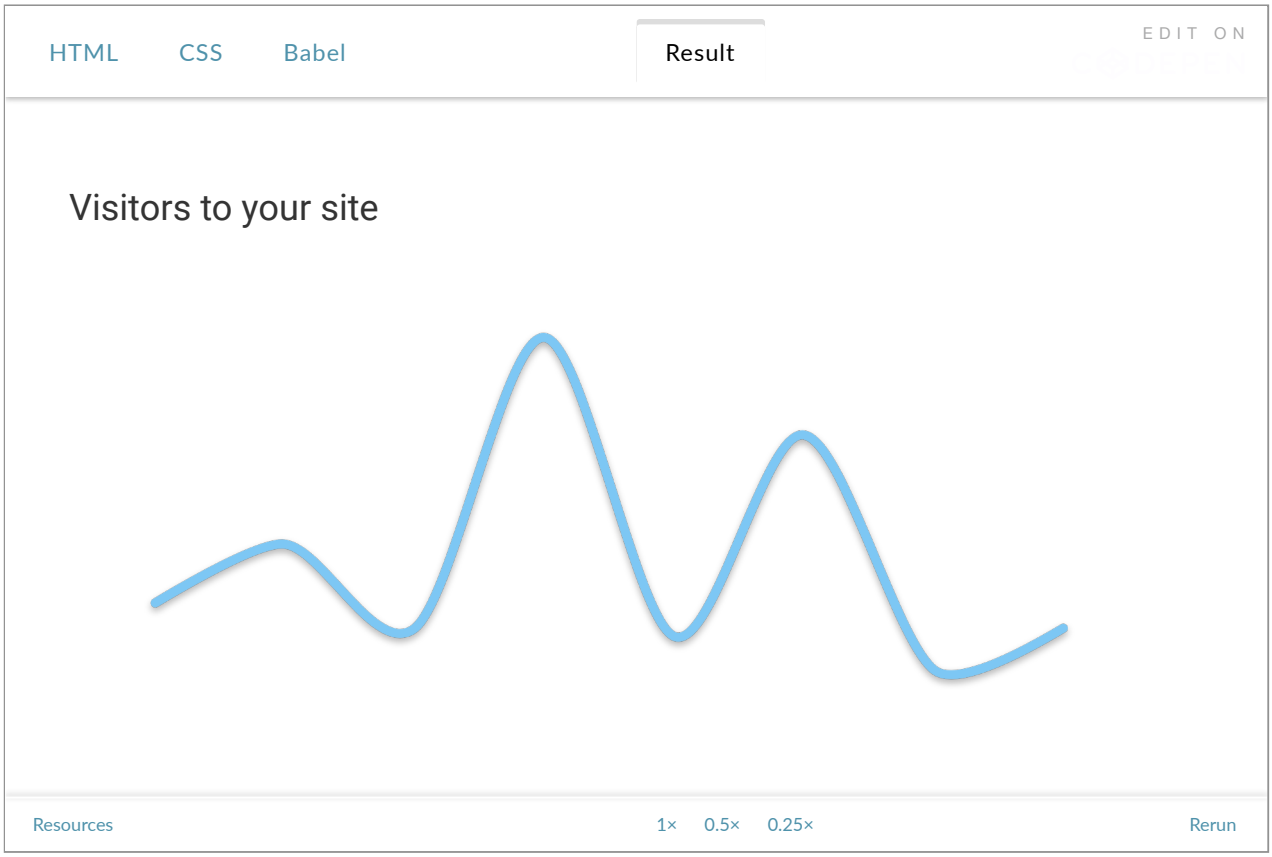
```
var LineChart=React.createClass({

    propTypes: {
        width:React.PropTypes.number,
```

```
            height:React.PropTypes.number,
            chartId:React.PropTypes.string
        },

        getDefaultProps: function() {
            return {
                width: 800,
                height: 300,
                chartId: 'v1_chart'
            };
        },
        getInitialState:function(){
            return {
                width:this.props.width
            };
        },
        render:function(){
            var data=[
                {day:'02-11-2016',count:180},
                {day:'02-12-2016',count:250},
                {day:'02-13-2016',count:150},
                {day:'02-14-2016',count:496},
                {day:'02-15-2016',count:140},
                {day:'02-16-2016',count:380},
                {day:'02-17-2016',count:100},
                {day:'02-18-2016',count:150}
            ];

            var margin = {top: 5, right: 50, bottom: 20, left: 50},
                w = this.state.width - (margin.left + margin.right),
                h = this.props.height - (margin.top +
    margin.bottom);

            var parseDate = d3.time.format("%m-%d-%Y").parse;

            data.forEach(function (d) {
                d.date = parseDate(d.day);
```

```
        });

        var x = d3.time.scale()
            .domain(d3.extent(data, function (d) {
                return d.date;
            }))
            .rangeRound([0, w]);

        var y = d3.scale.linear()
            .domain([0,d3.max(data,function(d){
                return d.count+100;
            })])
            .range([h, 0]);

        var line = d3.svg.line()
            .x(function (d) {
                return x(d.date);
            })
            .y(function (d) {
                return y(d.count);
            }).interpolate('cardinal');

        var transform='translate(' + margin.left + ',' +
margin.top + ')';

        return (
```

```
        );
    }
});
```

Here is the demo :



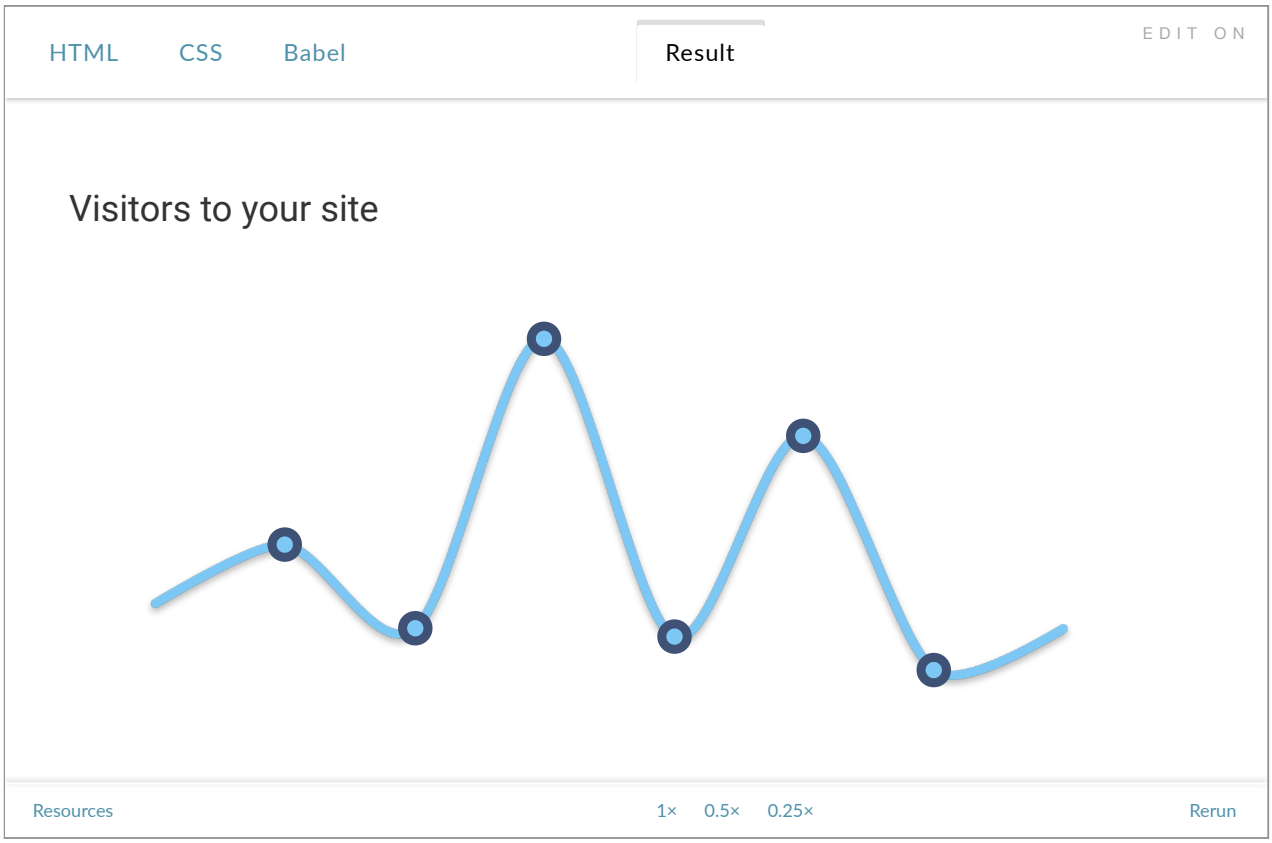
Next we will add the `Dots`. Lets create another class named `Dots`. Here we shall pass the xScale & yScale as function and the dataset to calculate the position of the circle. The below code is straight forward, let me know in case you have any questions.

```
var Dots=React.createClass({
    propTypes: {
        data:React.PropTypes.array,
        x:React.PropTypes.func,
        y:React.PropTypes.func

    },
    render:function(){
```

```
        var _self=this;

        //remove last & first point
        var data=this.props.data.splice(1);
        data.pop();

        var circles=data.map(function(d,i){

            return (
                {circles}

        );
    }
});
```

Now add the Dots Component in our SVG.

Now your output should look like this:



Adding Axis & Grid:

In order to add the Axis and Grid we will use D3 to update the DOM since the calculation for setting up the those in React could be complex. So this time we will use D3 to draw them.

First add the following to the `render()` method of the LineChart class.

```
var yAxis = d3.svg.axis()
            .scale(y)
            .orient('left')
            .ticks(5);

var xAxis = d3.svg.axis()
   .scale(x)
   .orient('bottom')
   .tickValues(data.map(function(d,i){
       if(i>0)
           return d.date;
   }).splice(1))
```

```
    .ticks(4);

var yGrid = d3.svg.axis()
    .scale(y)
    .orient('left')
    .ticks(5)
    .tickSize(-w, 0, 0)
    .tickFormat("");
```

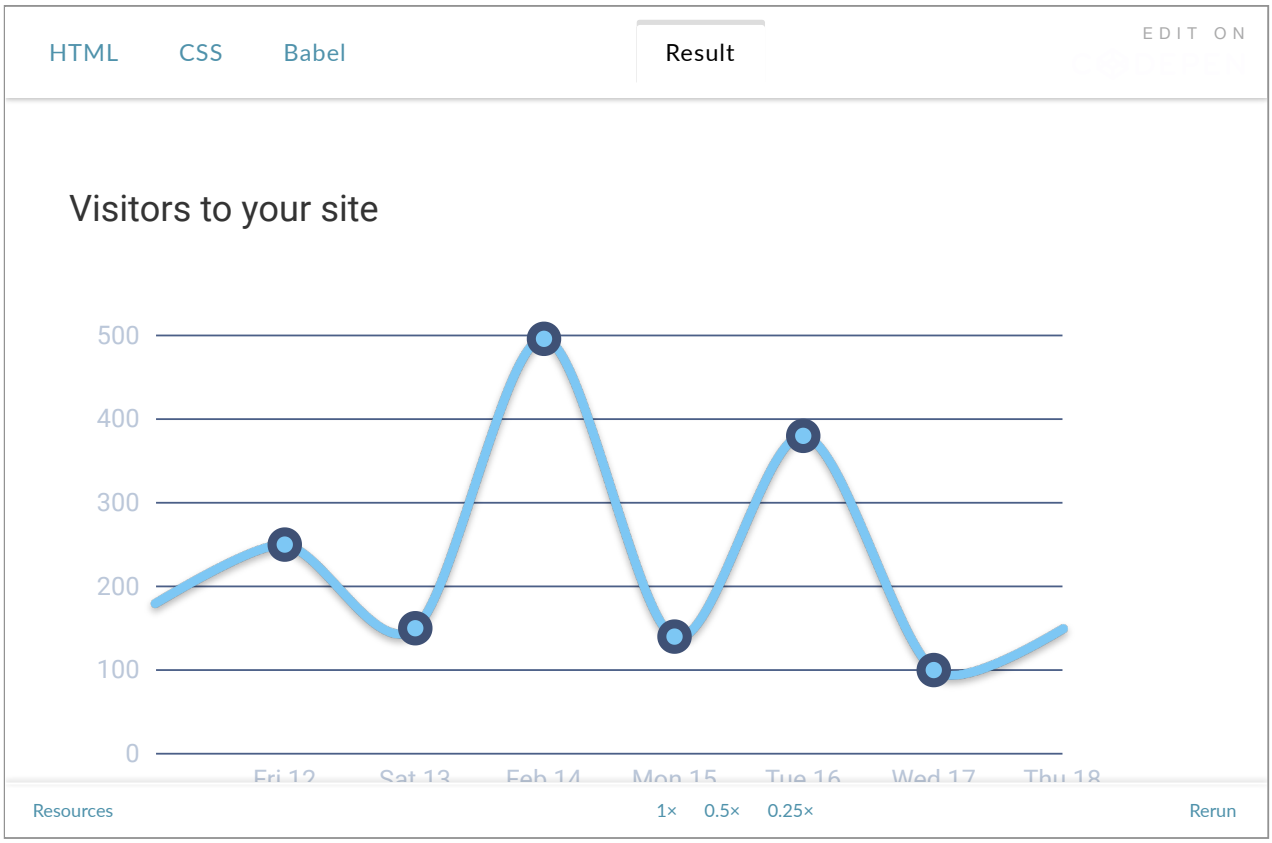
Then create the `Axis` & `Grid` classes. So here, we will call `renderAxis()` function from `componentDidUpdate()` and `componentDidMount()` method. in `renderAxis()` method we will call the `ReactDOM.findDOMNode(this)` to get the current dom element and then call `d3.select(node).call(this.props.axis);` function. Both the Axis and Grid classes are very same, in Grid class we will pass the `yGrid` function in the `props` and call the `d3.select(node).call(this.props.grid);`

```
var Axis=React.createClass({
    propTypes: {
        h:React.PropTypes.number,
        axis:React.PropTypes.func,
        axisType:React.PropTypes.oneOf(['x','y'])

    },

    componentDidUpdate: function () { this.renderAxis(); },
    componentDidMount: function () { this.renderAxis(); },
    renderAxis: function () {
        var node = ReactDOM.findDOMNode(this);
        d3.select(node).call(this.props.axis);

    },
    render: function () {

        var translate = "translate(0,"+(this.props.h)+")";

        return (
```

```
        );
    }

});

var Grid=React.createClass({
    propTypes: {
        h:React.PropTypes.number,
        grid:React.PropTypes.func,
        gridType:React.PropTypes.oneOf(['x','y'])
    },

    componentDidUpdate: function () { this.renderGrid(); },
    componentDidMount: function () { this.renderGrid(); },
    renderGrid: function () {
        var node = ReactDOM.findDOMNode(this);
        d3.select(node).call(this.props.grid);

    },
    render: function () {
        var translate = "translate(0,"+(this.props.h)+")";
        return (


        );
    }

});
```

Now lets add the Axis and Grid in the JSX of the `LineChart` `render()` function.

Here is the live demo of whatever we have completed so far.



Adding Tooltip:

Generally you can add a tooltip using simple html and css, however it would have some limitations. We will use a svg element in order to display the tooltip. `onMouseOver` and `onMouseOut` events from the `Dots` component should display the tooltip in this case. Let's add the events in the `circle` element.

We will call a function during `onMouseOver` and `onMouseOut` events, which needs to be passed from the parent React Component (`LineChart`).

Since we want to display the data points for a specific circle, we will use `html5` `data-` element to define them in the circle element itself. I have added `data-key` and `data-value` here.

```
var circles=data.map(function(d,i){

    return ()
});
```

Now lets pass the functions in the JSX.

We need to add the `showToolTip` and `hideToolTip` function to the `LineChart` Component. We should be able to display the tool tip from anywhere in the SVG, not just from one component, that's why we will define the event functions in the parent level. However let's create the ToolTip component first, then we will add the `showToolTip` and `hideToolTip` function to the LineChart Component.

Here is the `ToolTip` class. It receives the tooltip object which has the boolean value to show/hide the component and the required data fields (data-key & data-value) which we had added in the circle elements. The show/hide property will be controlled by the visibility attribute of svg. We also need to calculate the location of the tooltip based on the x & y position of the circle component.

If you look closely, I am calculating the position so that the tooltip can be displayed either above or below the circle based on how many pixels are available. You can easily perform similar calculation for left and right position.

The `polygon` element would create the small arrow using a triangle.

NOTE : this implementation of the Tooltip is not complete and I will have a more reusable version available in future tutorials.

```
var ToolTip=React.createClass({
    propTypes: {
        tooltip:React.PropTypes.object
    },
    render:function(){

        var visibility="hidden";
        var transform="";
        var x=0;
        var y=0;
        var width=150,height=70;
        var transformText='translate('+width/2+','+(height/2-
5)+')';
        var transformArrow="";
```

```
        if(this.props.tooltip.display==true){
            var position = this.props.tooltip.pos;

            x= position.x;
            y= position.y;
            visibility="visible";


            if(y>height){
                transform='translate(' + (x-width/2) + ',' + (y-
height-20) + ')';
                transformArrow='translate('+(width/2-20)+','+
(height-2)+')';
            }else if(y


                    {this.props.tooltip.data.key}
                    {this.props.tooltip.data.value+" visits"}


        );
    }
});
```

Now its time to add the `showToolTip` and `hideToolTip` function to the LineChart Component. We will use the event object here to capture the details from the calling circle element. First let's set the fill to `white`. then set the tooltip object in `this.state`. In the `hideToolTip` function, we will reset the fill attribute and set the default tooltip to `this.state`. This will hide the tooltip.

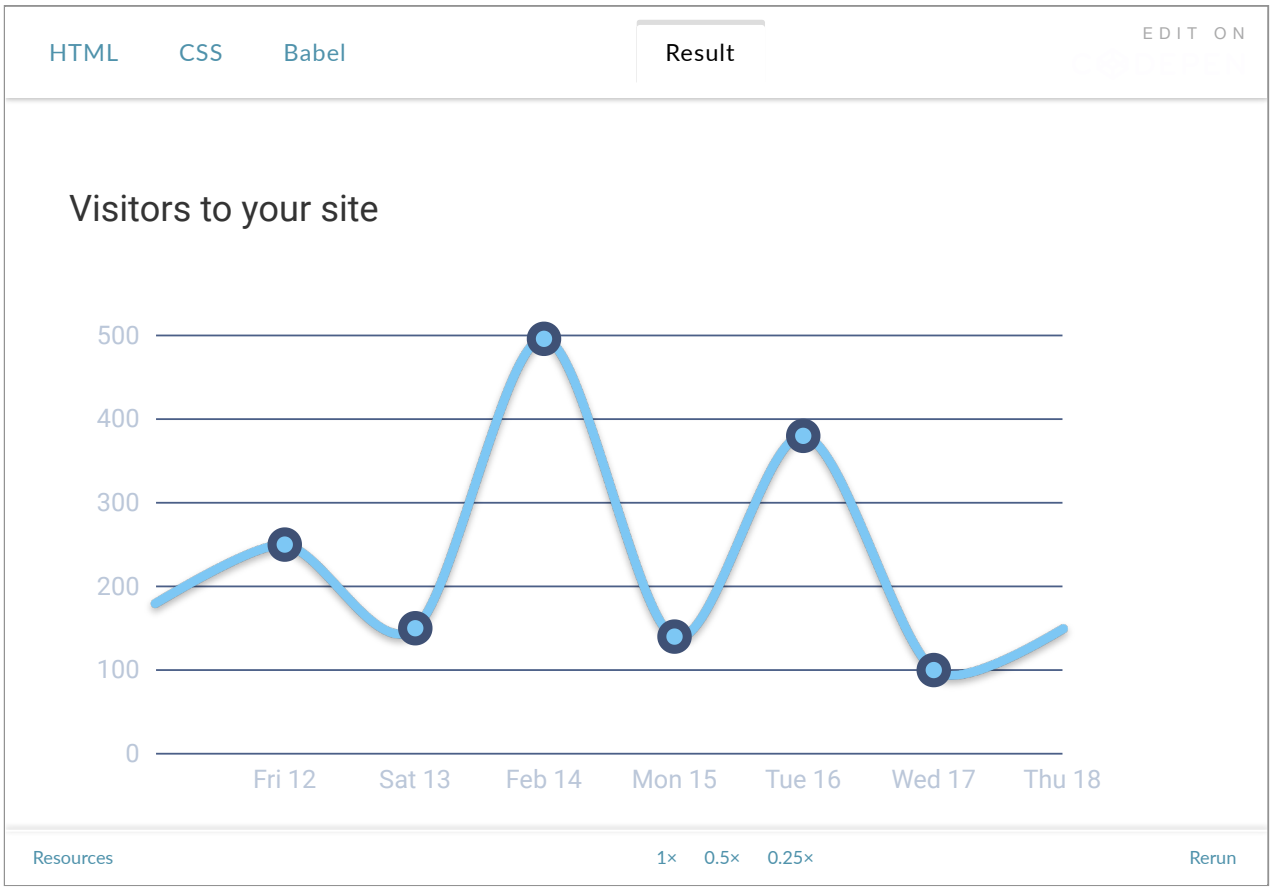
```
showToolTip:function(e){
    e.target.setAttribute('fill', '#FFFFFF');
```

```
        this.setState({tooltip:{
            display:true,
            data: {
                key:e.target.getAttribute('data-key'),
                value:e.target.getAttribute('data-value')
                },
            pos:{
                x:e.target.getAttribute('cx'),
                y:e.target.getAttribute('cy')
            }

            }
        });
    },
    hideToolTip:function(e){
        e.target.setAttribute('fill', '#7dc7f4');
        this.setState({tooltip:{ display:false,data:
    {key:'',value:''}}});
    }
```

We also need to update the `getInitialState()` function to add the default tooltip to `this.state`.

```
    getInitialState:function(){
        return {
            tooltip:{ display:false,data:{key:'',value:''}},
            width:this.props.width
        };
    },
```

Here is the demo of what we have created so far.



Add Responsiveness:

Now the only pending item we have is to add the Responsiveness to the chart. As I said before its very easy to make charts responsive. We already have the width defined as two way binding, now need to set the value of the width on browser resize event. We will use JQuery here to add the listener in `componentWillMount` and remove it in `componentWillUnmount` function. The `updateSize` function would get the current node and find the width using the JQuery `width()` function. Then we will set it in the `this.state.width`. This will trigger a re-render of the components.

We will also call the `updateSize` function from the `componentDidMount` in order to render the chart using current width. This will be helpful when your browser window width is already less then optimum.

```
componentWillMount:function(){

    var _self=this;

    $(window).on('resize', function(e) {
        _self.updateSize();
```

```
    });

    this.setState({width:this.props.width});

},
componentDidMount: function() {
    this.updateSize();
},
componentWillUnmount:function(){
    $(window).off('resize');
},

updateSize:function(){
    var node = ReactDOM.findDOMNode(this);
    var parentWidth=$(node).width();

    if(parentWidth
```

Mixin:

This code is common across all the charts. So I have created a mixin and added the `mixin` in the React class.

Note : At the time of writing this article, React ES6 does not support mixin.

```
var resizeMixin={
...
}
window.resizeMixin=resizeMixin;

// Add the following in LineChat Class
```

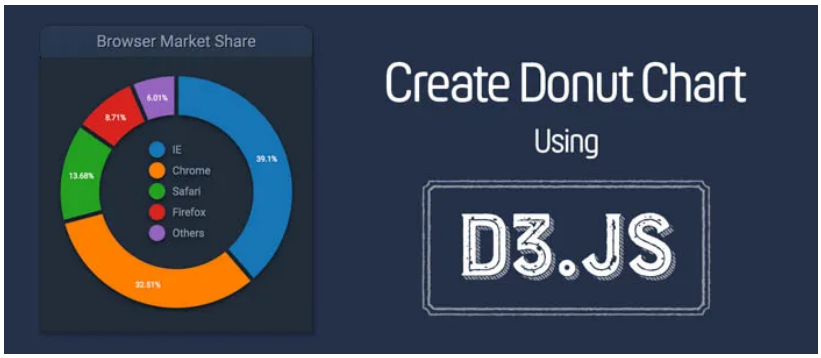
```
mixing:[resizeMixin]
```

You can see the final version here.

https://adeveloperdiary.github.io/react-d3-charts/01_Visitor_Dashboard/index.html

Donut Chart:

I have already provided a tutorial on how to create Donut Charts using D3. You can refer this post to get the basic idea of Pie Charts and Legends.



Create a simple Donut Chart using D3.js

We will learn how to Create a simple Donut Chart using D3.js . Even though the Pie chart is not very efficient in data visualization the Donut Charts are sometimes very helpful. We have learned how to Create a Simple Pie Chart using D3.js in our previous post. In this article we will take our ... Continue reading

A Developer Diary 5

Lets look at the render function of the Donut Chart below. We will have one component to render the Paths and another for the Legends. We are having the data as two-way binding and `onClick` event we are calling `updateData()` function. In the demo, click on the DonutChart to update the data. This is default behavior using React's two-way binding and no additional coding required.

```
render:function(){

    return (









    );
}
```

We will add the `ResizeMixin` to add the responsiveness to the chart. However I have customized the `DonutChartLegend` class to render differently when the width of the pages changes. You can change the window size of the demo and see how the legends are changing position.

Here is the full code of the DonutChart.jsx

```
var DonutChartPath=React.createClass({
    propTypes: {
        width:React.PropTypes.number,
        height:React.PropTypes.number,
        data:React.PropTypes.array,
        pie:React.PropTypes.func,
        color:React.PropTypes.func
    },
    componentWillMount:function(){

        var radius=this.props.height;

        var outerRadius=radius/2;
        var innerRadius=radius/3.3;

        this.arc=d3.svg.arc()
            .outerRadius(outerRadius)
            .innerRadius(innerRadius);

        this.transform='translate('+radius/2+','+radius/2+')';

    },
    createChart:function(_self){

        var paths =
(this.props.pie(this.props.data)).map(function(d, i) {

            return (

            )
        });
        return paths;
    },
```

```
        render:function(){

            var paths = this.createChart(this);

            return(

                    {paths}

            )
        }
    });

    var DonutChartLegend=React.createClass({
        propTypes: {
            width:React.PropTypes.number,
            height:React.PropTypes.number,
            data:React.PropTypes.array,
            pie:React.PropTypes.func,
            color:React.PropTypes.func
        },
        createChart:function(_self){

            var texts =
    (this.props.pie(this.props.data)).map(function(d, i) {

                var transform="translate(10,"+i*30+")";

                var rectStyle = {
                    fill:_self.props.color(i),
                    stroke:_self.props.color(i)

                };

                var textStyle = {
                    fill:_self.props.color(i)
                };
```

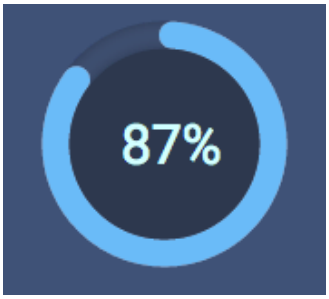
```
            return (


                    {d.data.name}

            )
        });
        return texts;
    },

    render:function(){

        var style={
            visibility:'visible'
        };

        if(this.props.width<=this.props.height+70){
            style.visibility='hidden';
        }

        var texts = this.createChart(this);
        var transform="translate("+
(this.props.width/2+80)+",55)";
        return(

                {texts}

        )
    }
});

var DonutChart=React.createClass({
    propTypes: {
        width:React.PropTypes.number,
        height:React.PropTypes.number,
        padAngle:React.PropTypes.number,
        id:React.PropTypes.string.isRequired
```

```
    },

    getDefaultProps: function() {
        return {
            width: 450,
            height: 250,
            padAngle:0
        };
    },
    getInitialState:function(){
        return {
            data:[],
            width:0
        };
    },

    mixins:[resizeMixin],

    componentWillMount:function(){

        this.pie=d3.layout.pie()
            .value(function(d){return d.count})
            .padAngle(this.props.padAngle)
            .sort(null);

        this.color = d3.scale.ordinal()

.range(['#68c8d7','#eccd63','#bb8cdd','#de6942','#52b36e','#bbc7d9']);

        var data = [
            { name: 'IE', count: 40 },
            { name: 'Chrome', count: 32 },
            { name: 'Safari', count: 14 },
            { name: 'Firefox', count: 9 },
            { name: 'Others', count: 6 }
        ];
```

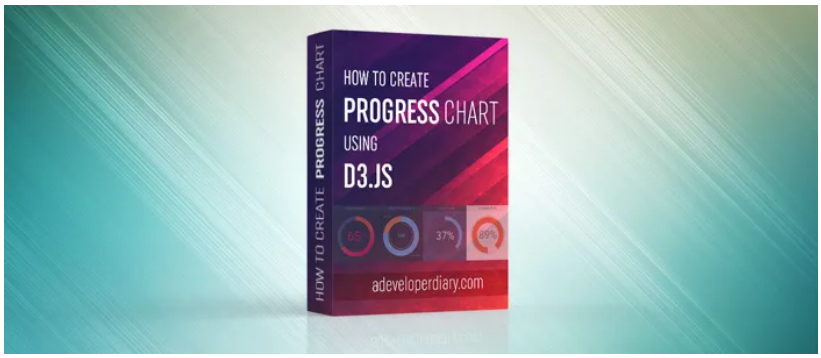
```
            this.setState({'data':data,width:this.props.width});
        },

        updateData:function(){
            var data = [
                { name: 'IE', count: Math.random() },
                { name: 'Chrome', count: Math.random() },
                { name: 'Safari', count: Math.random() },
                { name: 'Firefox', count: Math.random() },
                { name: 'Others', count: Math.random() },
                { name: 'Opera', count: Math.random() }

            ];

            this.setState({'data':data });
        },
        render:function(){

            return (









            );
        }
    });

    window.DonutChart = DonutChart;
```

Progress Chart:

We will adding the below progress chart.



Again, I have another post where I have explained how I have created this using D3. Please refer it.



How to create Progress chart using d3.js

We have learnt how to create donut chart using d3.js and we have also animated the chart. In this lesson we will now learn How to create Progress chart using d3.js. We will create four different types of progress chart. If you need to understand how to create basic donut chart using d3 then you ... Continue reading

A Developer Diary 3

Below is the JSX code for the Progress Chart. Apart from the regular SVG elements I have a `InsetShadow` React Component.

```
{this.state.percent*100+'%'}
```

SVG Shadow:

I have create a common React Component named `InsetShaow` to add `defs` in the `svg` element. Later I am using them in the filter property of the style attribute.

```
var InsetShadow=React.createClass({

    propTypes: {
        id:React.PropTypes.string,
        stdDeviation:React.PropTypes.string,
        floodColor:React.PropTypes.string,
        floodOpacity:React.PropTypes.string
    },
    render:function(){
```

```
        return(

        );
    }

});

window.InsetShadow=InsetShadow;
```

Now you have seen the usability of React in terms of reusability and two-way biding.

Here is the full code of the `ProgressChart.jsx`

```
var ProgressChart=React.createClass({
    propTypes: {
        width:React.PropTypes.number,
        height:React.PropTypes.number,
        chartId:React.PropTypes.string
    },

    getDefaultProps: function() {
        return {
            width: 200,
            height: 200,
```

```
            chartId: 'v_chart'
        };
    },
    getInitialState:function(){
        return {percent:0};
    },
    componentWillMount:function(){

        this.setState({percent:.87});
    },
    componentWillUnmount:function(){

    },
    updateData:function(){

        var value=(Math.floor(Math.random() * (80) + 10))/100;

        this.setState({percent:value});
    },
    render:function(){

        var color = ['#404F70','#67BAF5','#2d384d'];

        var outerRadius=(this.props.height/2)-10;
        var innerRadius=outerRadius-20;

        var arc=d3.svg.arc()
            .innerRadius(innerRadius)
            .outerRadius(outerRadius)
            .startAngle(0)
            .endAngle(2*Math.PI);

        var arcLine=d3.svg.arc()
            .innerRadius(innerRadius)
            .outerRadius(outerRadius)
            .cornerRadius(20)
            .startAngle(-0.05);
```

```
        var
transform='translate('+this.props.width/2+','+this.props.height/2+')';
        var style1={
            filter:'url(#inset-shadow1)'
        };
        var style2={
            filter:'url(#inset-shadow2)'
        };
        var styleText= {
            'fontSize': '40px'
        };
        return (


 {this.state.percent*100+'%'}









        );
    }
});

window.ProgressChart=ProgressChart;
```

BarChart:

The BarChart is quite self explanatory, so I am not going in detail here. Please find the code below.

```
var BarChart=React.createClass({
    getDefaultProps: function() {
        return {
            width: 300,
            height: 70,
            chartId: 'v_chart'
        };
    },
    getInitialState:function(){
        return {
            width:0
        };
    },
    mixins:[resizeMixin],

    render:function(){
        var data=[
            { month:'Jan', value:40 },
            { month:'Feb', value:50 },
            { month:'Mar', value:65 },
            { month:'Apr', value:60 },
            { month:'May', value:70 },
            { month:'Jun', value:55 },
            { month:'Jul', value:80 },
            { month:'Aug', value:55 },
            { month:'Sep', value:75 },
            { month:'Oct', value:50 },
            { month:'Nov', value:60 },
            { month:'Dec', value:75 }
        ];

        var margin={top:5,right:5,bottom:5,left:5},
        w=this.state.width-(margin.left+margin.right),
        h=this.props.height-(margin.top+margin.bottom);
```

```
        var
transform='translate('+margin.left+','+margin.top+')';

        var x=d3.scale.ordinal()
            .domain(data.map(function(d){
                return d.month;
            }))
            .rangeRoundBands([0,this.state.width],.35);

        var y=d3.scale.linear()
            .domain([0,100])
            .range([this.props.height,0]);

        var rectBackground=(data).map(function(d, i) {

            return (

            )
        });
        var rectForeground=(data).map(function(d, i) {

            return (

            )
        });

        return(
```

```
            );
    }

});

window.BarChart=BarChart;
```

Conclusion:

At the end, I will like to emphasis on some of the advantages we discussed earlier.

Reusability

By now you must be convinced that event single class/component created in React is reusable. This is a huge advantage when integrating D3 and react in enterprise level.

Update Chart

We haven’t actually written any code to update the chart. All we have done is to define the data in this.state to make it a two-way bindable. Rest has been taken care by react.

Responsiveness

Again, for this we added a mixin as a common code to all our charts and that did most of the tricks. Look into the BarChart.jsx and you won't find any additional code to make the chart responsive. This is huge.

Shared Library / Structural Code / Maintainability / Extensibility / Scalability

Even though I haven't put much effort on the code quality, the code is very well structured and can be easily unit tested. You can also expose these components as reusable charts and keep adding more features to it.

I hope this tutorial on How to integrate React and D3 – The right way would help you get a solid understanding of how we can use D3 with React.

Feel free to share your experience and feedbacks.

The entire codebase is available over github.

View Code

Update:

Please find the part 2 of this post here.

How to create reusable charts with React and D3 Part1

In this series we will continue working on our D3 components that we have created in our past lesson and make them fully reusable, extensible and maintainable. This How to create reusable charts with React and D3 Part1 would cover the basics and then we will dive into more complex scenarios. Prerequisites: You would need ... Continue reading

A Developer Diary 0

Related

How to create reusable charts with React and D3 Part1

How to create reusable charts with React and D3 Part2

How to create reusable charts with React and D3 Part3

In "D3.js" In "D3.js" In "D3.js"

Filed Under: D3.js, React JS Chart, React JS, React+d3

Tagged With: chart, d3, Dashboard, Donut Chart, integrate react and D3, Line

Comments

aditya says

March 22, 2016 at 5:35 pm

(Edit)

Hey, Thanks for the tutorial. I am quiet new to both D3 and React in general and i am having issues in just drawing the line for step1. I follow the code base till return but not able to display data in my React app.

Reply

A Developer Diary says
March 22, 2016 at 8:38 pm

(Edit)

Hi Aditya,
Could you please share the error that you are getting ? If possible share the code itself, so that I can take a look at it.

Thanks,
A Developer Diary

Reply

aditya says
April 5, 2016 at 9:05 am

(Edit)

Hi A Developer Diary,
Thanks for the reply. I was able to make the display of data work using D3 and React. But currently i am facing issues to make the chart responsive. I was able to do it in D3.js alone (https://jsfiddle.net/adityap16/11edxrnq/1/) , this is inspired from the following links top answer http://stackoverflow.com/questions/16265123/resize-svg-when-window-is-resized-in-d3-js. My problem is when i am integrating the solution in React my plots are becoming really small. Can you think of any reason why that be happening. Thanks again !

Reply

A Developer Diary says
April 5, 2016 at 11:56 am

(Edit)

Hi Aditya,
You need to probably play with the viewBox attribute.I tried and it did work for me.Looks like you are trying to scale SVG using the viewBox attribute, which is bit different than being responsive. If you are looking for zoom option then fine, otherwise you probably want to control how the chart is being displayed, like change position of the legend, reduce data point or reduce tick value etc.

Reply

aditya says
April 5, 2016 at 1:00 pm

(Edit)

Hi A Developer Diary,
Thanks for the reply. It appears that the ticks or the axis is not being controlled correctly. The ticks which seems omitted in full screen appears when you shrink the window. I have tried playing the viewbox attribute and the bottom padding but somehow its not working optimally. Also either it is aligned centrally or the y axis moves out of the window.

Reply

M says
May 11, 2016 at 8:35 am

(Edit)

Hello

I'm new to both D3.js and React.js, I basically took your code from github and tried to run it. Unfortunately, nothing showed up and there is no errors. I'm using webpack/babel to run jsx code.

Reply

A Developer Diary says
May 11, 2016 at 11:00 am

(Edit)

Hi M,
If you are not using Chrome, I would suggest to use chrome and find whether you are getting any error in the console. Here we are using in browser jsx compiler so we don't need to compile them before loading the page.

Let me know if you are seeing any error in the browser's console.

Reply

M says
May 16, 2016 at 2:28 am

(Edit)

I'm seeing this error
ReferenceError: Can't find variable: require

Reply

A Developer Diary says
May 16, 2016 at 10:08 pm

(Edit)

Hi M,
I have another tutorial on How to Setup React JS Development Environment for ES5 and ES6. Please find the steps here :

https://www.adeveloperdiary.com/react-js/setup-react-js-development-environment-for-es5-and-es6/

Reply

M says

May 23, 2016 at 8:35 am

(Edit)

Hi A Developer Diary

Quick update everything is working great. I just have one question, now I'm trying to use react with node.js to interact with mySQL. So if you have any examples that deal with such case it would be great to direct me to it. Also, I'm thinking of using AJAX to interact with node.js, is it a good approach ?

Reply

John says

May 16, 2016 at 7:49 pm

(Edit)

How do you changed the font size for the charts?

Reply

A Developer Diary says

May 16, 2016 at 10:05 pm

(Edit)

Hi John,
SVG supports font-size css property. Please take a look at the style.css in github. Here is a sample:

```
.axis text {
font-size: 14px;
}

.browser-legend{
font-size:14px;
opacity:.8;
}
```

Reply

Chris says
May 25, 2016 at 10:52 am

(Edit)

Hi, thank you for the tutorial I am implementing your code starting with the line chart into a react app I keep getting an error that says “h is not defined”. I am not sure where I am going wrong because I do have h defined.

Reply

A Developer Diary says
May 25, 2016 at 12:36 pm

(Edit)

Hi Chris,
May be h is not initialized, h depends on this.props.height. Please check whether you have defined height in getDefaultProps().

You can also find the sample working version in code pen. May be you can compare your version with this one.

http://codepen.io/adeveloperdiary/pen/PNNvpN

Let me know if you are still facing the issue. If you can, share the code base with me to take a look at.

Thanks,
A Developer Diary
Reply

Lars says
May 27, 2016 at 6:19 am

(Edit)

"However its probably best to avoid the Transitions altogether to start with, since they don't have real business value."

That's a ridiculous and sweeping statement. If everyone had a mindset like that we'd still be using ms-dos. Transitions and animations can play a crucial part in visualising and understanding data.

Reply

A Developer Diary says
May 27, 2016 at 10:52 am

(Edit)

Hi Lars,
I agree with you. I wanted to convey a different message however ended up with a controversial statement. I advocate Animation and Transition a lot and use them everywhere. Anyway what I wanted to point out really is, "However its

probably best to avoid the Transitions altogether to start with, if they don't provide a real business value to your use case."

Again, thanks for pointing this out and giving me an opportunity to update the statement.

Reply

lucho says

June 2, 2016 at 7:11 pm

(Edit)

Great tutorial.
You mentioned the animation in the beginning of the tutorial, is that something you could show an example?
"D3 transitions functions cannot be used." "You may try using React Animation however it will be a separate topic to discuss in future posts.Otherwise the only option is to use the core d3 library to perform the animation."
I'm using the core d3 library, but I'm not sure how to add the animations to the component.

Thanks

Reply

A Developer Diary says

June 3, 2016 at 11:11 am

(Edit)

Hi lucho,
Thanks for your feedback ! I will send you the details soon.

Thanks

Reply

Liran Brimer says
July 4, 2016 at 7:36 am

(Edit)

Hi,
Great tutorial, thanks!

My X Axis has 7 ticks (for each day of the week), and I want to display a vertical line on the nearest tick when the mouse is on the graph.

How can I transform the mouse event's clientX into the equivalent of d3.mouse(this)[0] (which i don't have access anymore using your method) ?

I want to use it for calculating the nearest tick in each OnMouseMove event, and then show appropriate tooltip whenever it occurs.

Thanks

Reply

Liran Brimer says
July 4, 2016 at 9:17 am

(Edit)

Ok i figured out that i can use register to the original "mouseover" d3 event from componentDidMount and then access d3.mouse(this) to get the coordinates:

var node = ReactDOM.findDOMNode(this);
d3.select(node).on('mouseover', function() { console.log('MOUSE IS IN ', d3.mouse(this)); });

but i still wonder if there is a way to do it from the onMouseOver event, without this spaghetti code. what do you think?

Reply

Liran Brimer says
July 4, 2016 at 9:18 am

(Edit)

Ok i figured out that i can register to the original “mouseover” d3 event from componentDidMount and then access d3.mouse(this) to get the coordinates.

var node = ReactDOM.findDOMNode(this);
d3.select(node).on(‘mouseover’, function() { console.log(‘MOUSE IS IN ‘, d3.mouse(this)); });

but i still wonder if there is a way to do it from the onMouseOver event, without this spaghetti code. what do you think?

Reply

JulienD says
July 26, 2016 at 9:59 am

(Edit)

Thanks a lot. I think I have tried *every* way of making d3 work with React, and this was really the most intuitive and I got my bar chart done in less than 30 minutes.

Actually a bit more because D3 was upgraded to v4, and I had to make a few changes to scales to have it working again. Do you plan to upgrade your example in a new branch ? I could send a PR with my changes to the BarChart at least.

N.B. The method described there
http://oli.me.uk/2015/09/09/d3-within-react-the-right-way/
also worked fine. A different point of view: more reusable code (can reuse the d3 code as such without React), but adds a dependency and apparently sacrifices a bit of performance.
Reply

Alex says
September 19, 2016 at 10:13 pm

(Edit)

Hey I noticed that the third date along the bottom of the screen is wrong. It says Feb instead of Sun. I am curious why this is. I tried using different values than d.date but I get an error. any idea why we must use d.date in this situation? It tells me there is a problem with the transform if I try to change it to something else

Reply

Jan says
September 26, 2016 at 4:46 am

(Edit)

The visitors graph is broken in the last version of Google Chrome.

Reply

Jan says
September 26, 2016 at 4:56 am

(Edit)

In the last version of Internet Explorer all the graphs are broken.

Reply

Jan says
September 27, 2016 at 9:32 am

(Edit)

It looks like it is a bug of Google Chrome. The last update breaks the elements that use these CSS attributes inside :

-webkit-filter
-filter

Reply

jrayhan says
April 12, 2017 at 11:03 pm

(Edit)

There is some change
the line is not show
can you update your blog?

Reply

Justin Hardin says
May 3, 2017 at 5:43 pm

(Edit)

Great Tut! FWI the line graph and the Pie chart are not showing in Safari because of a bug in Safari http://stackoverflow.com/questions/36705323/why-is-filterdrop-

shadow-causing-my-svg-to-disappear-in-safari

Reply

Marie G says

July 30, 2017 at 9:58 am

(Edit)

Thanks for this tutorial, helped me get my first chart in react running. needs to update to v4 though, nonetheless good job and great tutorial.

Reply

Vishal says

September 27, 2017 at 5:23 pm

(Edit)

Hello,

Thank you such a nice tutorial. However when I try to run, it gives me some errors:

Uncaught Error: Element type is invalid: expected a string (for built-in components) or a class/function (for composite components) but got: undefined. Check the render method of $Browser$.

Reply

Leave a Reply

Logged in as Abhisek Jana. Edit your profile. Log out? Required fields are marked *

Comment *

Post Comment

This site uses Akismet to reduce spam. Learn how your comment data is processed.