A Developer Diary
{about:"code learn and share"}

April 6, 2016 By Abhisek Jana — 1 Comment (Edit)

How to create reusable charts with React and D3 Part2

In this **How to create reusable charts with React and D3 Part2** we will create Area and Line charts using multiple reusable React Components. We will use the knowledge we gained from our previous article here.

Limitation in Previous Implementation

We have already learnt how to integrate React and D3 in our previous lesson. So the objective here is how to take that one step further and create declarative and reusable charts. In our previous post, we had one React Element which was responsible to draw the chart, all we had to do is to create the React element. Here is the code that we had written so far.

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

Visitors to your site

```
            )
        }
});
```

There was bit of configuration (id,width etc) however, the previous implementation completely lacks declarative and reusable features. In case we want to plot two or more line charts together then there is no way to configure that. We also want to pass the name of all the css classes and turn off and on many features in a chart.

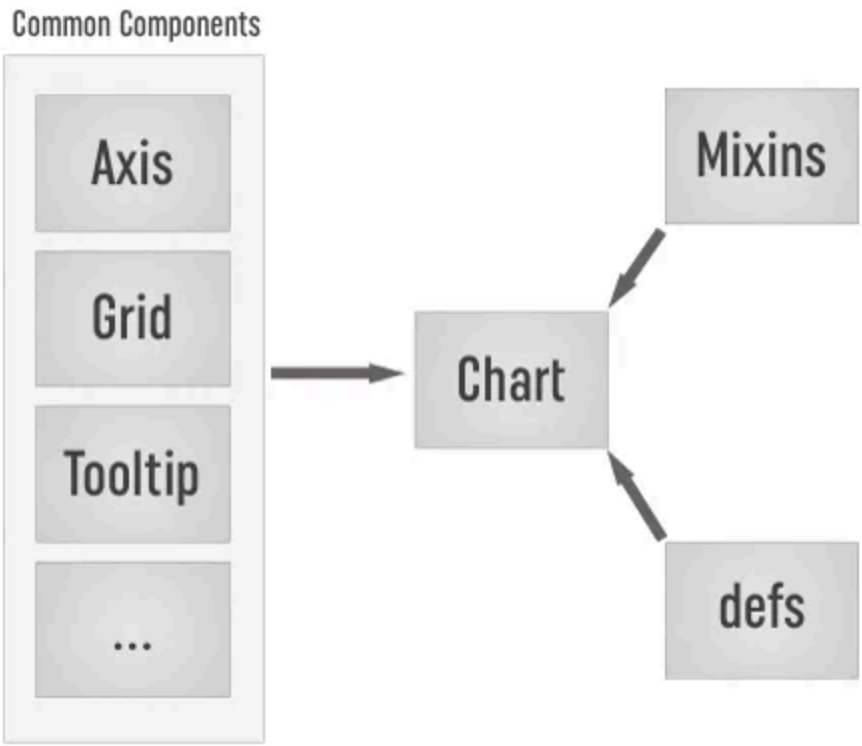
Objective

So we should be able to declare components we want to have in a chart. We don't want to modify JavaScript, however should be able to define them in HTML itself. We will be creating the charts as defined below, where we can specify whether we need a grid or not, we can add multiple charts dynamically and also able to control the axis.

Create Reusable Time Area Chart

The main reason I am adding Time and calling it as Time Area Chart is the xAxis. Lets talk about how we can structure our React components. As most of you are pretty smart in coding I won't go through line by line code here, however I will talk about all the design elements and concept. You should be able to find the rest by going through the code. You can also ask me questions in case of any confusion.

We will create `Axis`, `Grid`, `Tooltip` as reusable common React Components. Then we will have Mixins for responsiveness and defs for all the svg graphical elements like filter, gradient etc. Then we will add all these dynamically based on the elements provided by the html (JSX).



Axis

Here is the React Axis component we will be creating. This is straight forward and most of the code is generic. We are still using D3's built in function to

create the Axis.

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

orient:React.PropTypes.oneOf(['left','top','right','bottom']),
        className:React.PropTypes.string,
        tickFormat:React.PropTypes.string,
        ticks:React.PropTypes.number
    },

    componentDidUpdate: function () { this.renderAxis();
},
    componentDidMount: function () { this.renderAxis();
},
    renderAxis: function () {

       var _self=this;
       this.axis = d3.svg.axis()
            .scale(this.props.scale)
            .orient(this.props.orient)
            .ticks(this.props.ticks);

        if(this.props.tickFormat!=null &&
this.props.axisType==='x')

this.axis.tickFormat(d3.time.format(this.props.tickFormat));

        var node = ReactDOM.findDOMNode(this);
        d3.select(node).call(this.axis);
```

```
    },
    render: function () {

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

        return (


        );
    }

});
```

Grid

The React Grid Component is also very similar.

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

orient:React.PropTypes.oneOf(['left','top','right','bottom']),
        className:React.PropTypes.string,
        ticks:React.PropTypes.number
    },

    componentDidUpdate: function () { this.renderGrid();
},
    componentDidMount: function () { this.renderGrid();
```

```
},
    renderGrid: function () {

        this.grid = d3.svg.axis()
            .scale(this.props.scale)
            .orient(this.props.orient)
            .ticks(this.props.ticks)
            .tickSize(-this.props.len, 0, 0)
            .tickFormat("");

        var node = ReactDOM.findDOMNode(this);
        d3.select(node).call(this.grid);

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


        );
    }

});
```

Create D3 Time Area Chart

Lets create the main component of our chart. First, let's take a relook at our `D3TimeAreaChart` component. We have defined many child elements such as `yGrid`, `xAxis` and `area`, however they shouldn't be creating any direct HTML output.We need to read them inside our `D3TimeAreaChart` component and build the chart by ourself.

So at first we need to read all the child elements and transform them to appropriate React component. For an example, we should replace `yGrid` element with a `Grid` React element which will intern create the grid.

Since we know we are going to create an Area chart, we can create `xScale`, `yScale` and `area`. So lets create a function named `createChart()`. Then call it from the `render()` function. You should be able to recognize this code, otherwise please refer my previous post on How to Integrate React & D3.

```
createChart:function(_self){

    this.w = this.state.width - (this.props.margin.left +
this.props.margin.right);
    this.h = this.props.height - (this.props.margin.top +
this.props.margin.bottom);

    this.xScale = d3.time.scale()
        .domain(d3.extent(this.props.data, function (d) {
            return d[_self.props.xData];
        }))
        .rangeRound([0, this.w]);

    this.yScale = d3.scale.linear()
```

```
        .domain([0,d3.max(this.props.data,function(d){
            return
d[_self.props.yData]+_self.props.yMaxBuffer;
        })])
        .range([this.h, 0]);

    this.area = d3.svg.area()
        .x(function (d) {
            return this.xScale(d[_self.props.xData]);
        })
        .y0(this.h)
        .y1(function (d) {
            return this.yScale(d[_self.props.yData]);
        }).interpolate(this.props.interpolations);

    this.transform='translate(' + this.props.margin.left
+ ',' + this.props.margin.top + ')';
},
```

Next, we need to go through all the children and create component accordingly. We will utilize the `this.props.children` property to read the children. If the `this.props.children` is an array we will loop through the array and call a function named `createElement()`. `createElement` function should return `JSX` or React Component. Then we just need to specify the elements variable in the JSX. Remember, elements is an array of JSX or React Components.

```
render:function(){
    this.createChart(this);
    var elements;
    var _self=this;

    if(this.props.children!=null) {
        if (Array.isArray(this.props.children)) {
```

```
elements=this.props.children.map(function(element,i){
                return _self.createElements(element,i)
            });
        }else{

elements=this.createElements(this.props.children,0)
        }
    }

    return (









    );
}
```

Now we need to write the `createElement()` function. We have a switch conditional statement here, based on the element type (`element.type` should give us the name of the element we created, eg – xGrid, xAxis, area etc). So based on the element we will create appropriate React element and pass the required props.

Note : `…this.props` or `…element.props` should pass all the available property to the child.

In case of area being passed, we will create the path svg element and configure it with the data. In case you don't know how to create area charts in d3, it's exactly same as the line chart, we need to create a `path` element, the `d` attribute need to be populated using the `area(data)` function we created earlier. The `data[]` array has been extracted from the data that was passed in the `D3TimeAreaChart`.

```
createElements:function(element,i){
    var object;
    var _self=this;

    switch(element.type){

        case 'xGrid':
            object=;
            break;

        case 'yGrid':
            object=;
            break;

        case 'xAxis':
            object=;
            break;

        case 'yAxis':
            object=;
            break;

        case 'area':

            var data=[];
```

```
            for(var k=0,j=0;k;
            break;

    }
    return object;
}
```

So thats all, I have added the `resizeMixin` we created earlier and defined the `getDefaultProps()` and `getInitialState()` function to set the default values. Notice, the elements are created as they are defined in the parent element. This is very important, since you want to manage the layover by yourself by chaining the position.
I am having few lines to code to generate the `dataArea[]` array. I am using the `moment.js` for this, rest of it is simple.

```
var dataArea=[];

for(var i=0,j=0;i<15;++i,++j){

    var d={day:moment().subtract(j, 'days').format('MM-
DD-YYYY'),count:Math.floor((Math.random() * 30) +
5),type:'A'};
    d.date = parseDate(d.day);
    dataArea[i]=d;
}
for(var i=15,j=0;i<30;++i,++j){

    var d={day:moment().subtract(j, 'days').format('MM-
DD-YYYY'),count:Math.floor((Math.random() * 40) +
20),type:'B'};
    d.date = parseDate(d.day);
    dataArea[i]=d;
}
```

```
for(var i=30,j=0;i<45;++i,++j){

    var d={day:moment().subtract(j, 'days').format('MM-
DD-YYYY'),count:Math.floor((Math.random() * 50) +
30),type:'C'};
    d.date = parseDate(d.day);
    dataArea[i]=d;
}

var margin={
    top: 20, right: 30, bottom: 20, left: 50
};
```

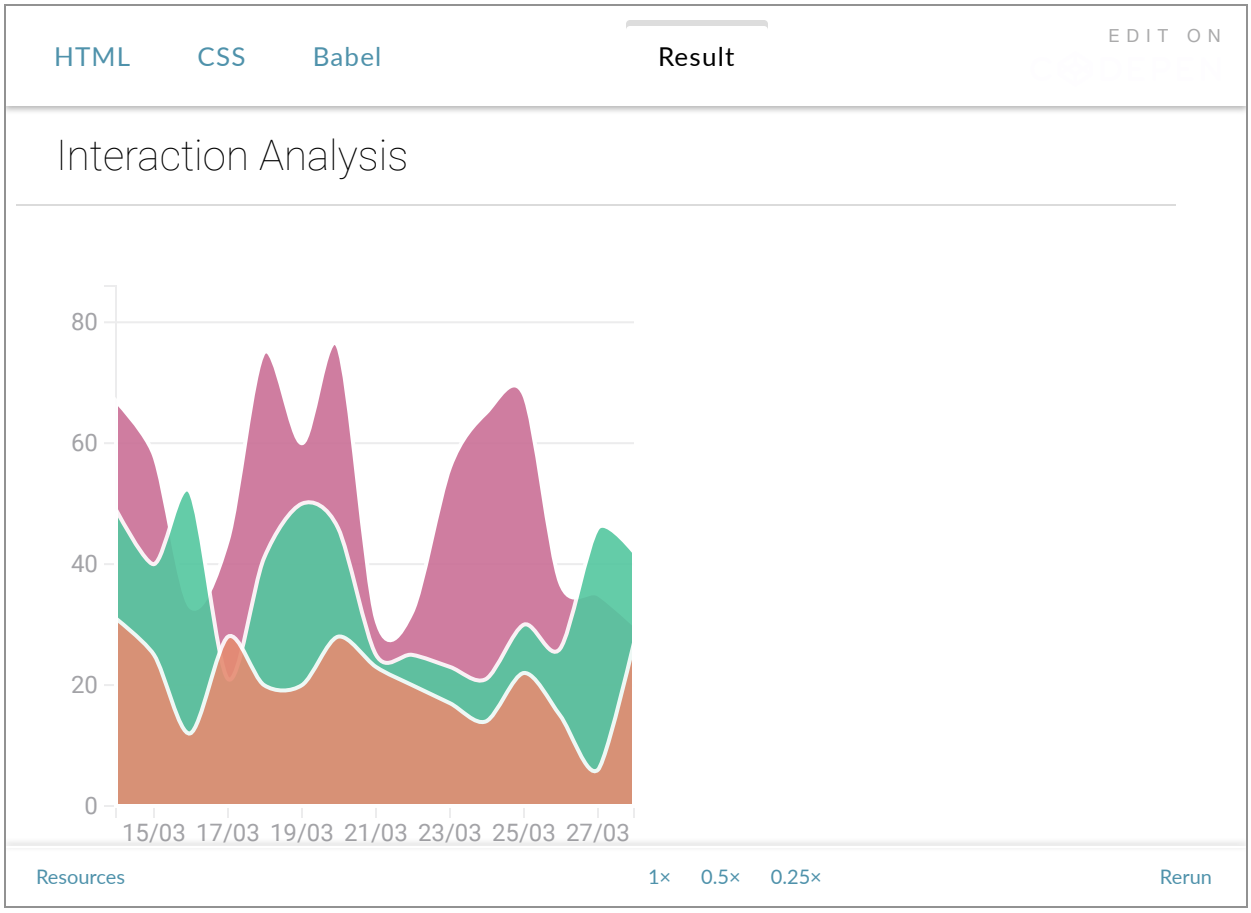
Here is the structure of the `dataArea` (only 2 data sets has been displayed below).

```
[
  {
    "day": "04-06-2016", "count": 13, "type": "A",
"date": "2016-04-06T04:00:00.000Z"
  },
  {
    "day": "04-05-2016", "count": 14, "type": "A",
"date": "2016-04-05T04:00:00.000Z"
  },
.
.
  {
    "day": "04-06-2016", "count": 40, "type": "B",
"date": "2016-04-06T04:00:00.000Z"
  },
  {
    "day": "04-05-2016", "count": 54, "type": "B",
```

```
"date": "2016-04-05T04:00:00.000Z"
  },
.
.

  {
    "day": "04-06-2016", "count": 70, "type": "C",
"date": "2016-04-06T04:00:00.000Z"
  },
  {
    "day": "04-05-2016", "count": 79, "type": "C","date":
"2016-04-05T04:00:00.000Z"
  }
]
```


Here is the demo of the just the Area Chart.



Create D3 Time Line Chart

We shall now add the `D3TimeLineChart` very similar way we added the above area chart. Here is the JSX element. There are few additional component we have here, such as gradient for the `area`, `dots` and `tooltip`.Notice how we can now combine both area and line chart together just by defining them as child elements.

```
    {/**/}
```

Dots Component

Since we are highlighting each data point by a circle (dot), lets create a common component named dots so that we can use across. We have added a prop to remove the first and last data point, you can make many customization like this as needed and define them. This dot component is responsive to tooltip so we have also defined them here, very similar to our previous example (please refer How to Integrate React & D3).

```
var Dots=React.createClass({
    propTypes: {
```

```
        data:React.PropTypes.array,
        xData:React.PropTypes.string.isRequired,
        yData:React.PropTypes.string.isRequired,
        x:React.PropTypes.func,
        y:React.PropTypes.func,
        r:React.PropTypes.string,
        format:React.PropTypes.string,
        removeFirstAndLast:React.PropTypes.bool
    },
    render:function(){

        var _self=this;

        //remove last & first point

        var data=[];

        if(this.props.removeFirstAndLast){
            for(var i=1;i)
        });

        return(

                {circles}

        );
    }
});
```

We will also add the ToolTip as another component. Here is the code. Explanation is available in the previous post (please refer How to Integrate React & D3).

```
var ToolTip=React.createClass({
    propTypes: {
        tooltip:React.PropTypes.object,
        bgStyle:React.PropTypes.string,
        textStyle1:React.PropTypes.string,
        textStyle2:React.PropTypes.string,
        xValue:React.PropTypes.string,
        yValue:React.PropTypes.string

    },
    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.xValue +" :
"+this.props.tooltip.data.key}
                    {this.props.yValue +" :
"+this.props.tooltip.data.value}


        );
    }
});
```

Add Defs

Since we will be using gradient, we need to add common React components for that.

```
var Gradient=React.createClass({

    propTypes: {
        id:React.PropTypes.string,
        color1:React.PropTypes.string,
        color2:React.PropTypes.string

    },
    render:function(){
        return(
```

```
        );
    }

});
```

Draw the chart

Now its time to create the chart, since we have all the components ready. Like previous example, we will create a function named `createChart()` to add `xScale`, `yScale`, `area` & `line`.

```
createChart:function(_self){

    this.w = this.state.width - (this.props.margin.left +
this.props.margin.right);
    this.h = this.props.height - (this.props.margin.top +
this.props.margin.bottom);

    this.xScale = d3.time.scale()
        .domain(d3.extent(this.props.data, function (d) {
            return d[_self.props.xData];
        }))
        .rangeRound([0, this.w]);

    this.yScale = d3.scale.linear()
        .domain([0,d3.max(this.props.data,function(d){
            return
d[_self.props.yData]+_self.props.yMaxBuffer;
        })])
        .range([this.h, 0]);

    this.area = d3.svg.area()
```

```
            .x(function (d) {
                return this.xScale(d[_self.props.xData]);
            })
            .y0(this.h)
            .y1(function (d) {
                return this.yScale(d[_self.props.yData]);
            }).interpolate(this.props.interpolations);

        this.line = d3.svg.line()
            .x(function (d) {
                return this.xScale(d[_self.props.xData]);
            })
            .y(function (d) {
                return this.yScale(d[_self.props.yData]);
            }).interpolate(this.props.interpolations);

        this.transform='translate(' + this.props.margin.left
+ ',' + this.props.margin.top + ')';
}
```

Next we call this from render function. However the render function has few more additional details than last time since we need to add the SVG defs element before the group element. So we need to ignore defs while calling the createElements function. Then, we need to loop through the `this.props.children` and create `defs[]` based on all the defs passed. As of now we have only one, named gradient defined. We need to defined `{defs}` before the `g` element in the JSX since we need to create them separately. Rest of the code is exactly same as previous.

```
render:function(){
    this.createChart(this);
```

```
    var elements;
    var defs;
    var _self=this;

    if(this.props.children!=null) {
        if (Array.isArray(this.props.children)) {

elements=this.props.children.map(function(element,i){

                if(element.type!="defs")
                    return
_self.createElements(element,i)
            });

            for(var i=0;i
```

); }

Here is the `createElements` function. You should be able to create Mixins for this. I have not optimized that much to make it easy to understand. We have the path & dots as a new option here, rest are same.

```
createElements:function(element,i){
    var object;
    switch(element.type){
        case 'dots':
```

```
            object=();
            break;

        case 'tooltip':
            object=;
            break;

        case 'xGrid':
            object=;
            break;

        case 'yGrid':
            object=;
            break;

        case 'xAxis':
            object=;
            break;

        case 'yAxis':
            object=;
            break;

        case 'area':
            object=;
            break;
        case 'path':
            object=;
            break;

    }
    return object;
}
```

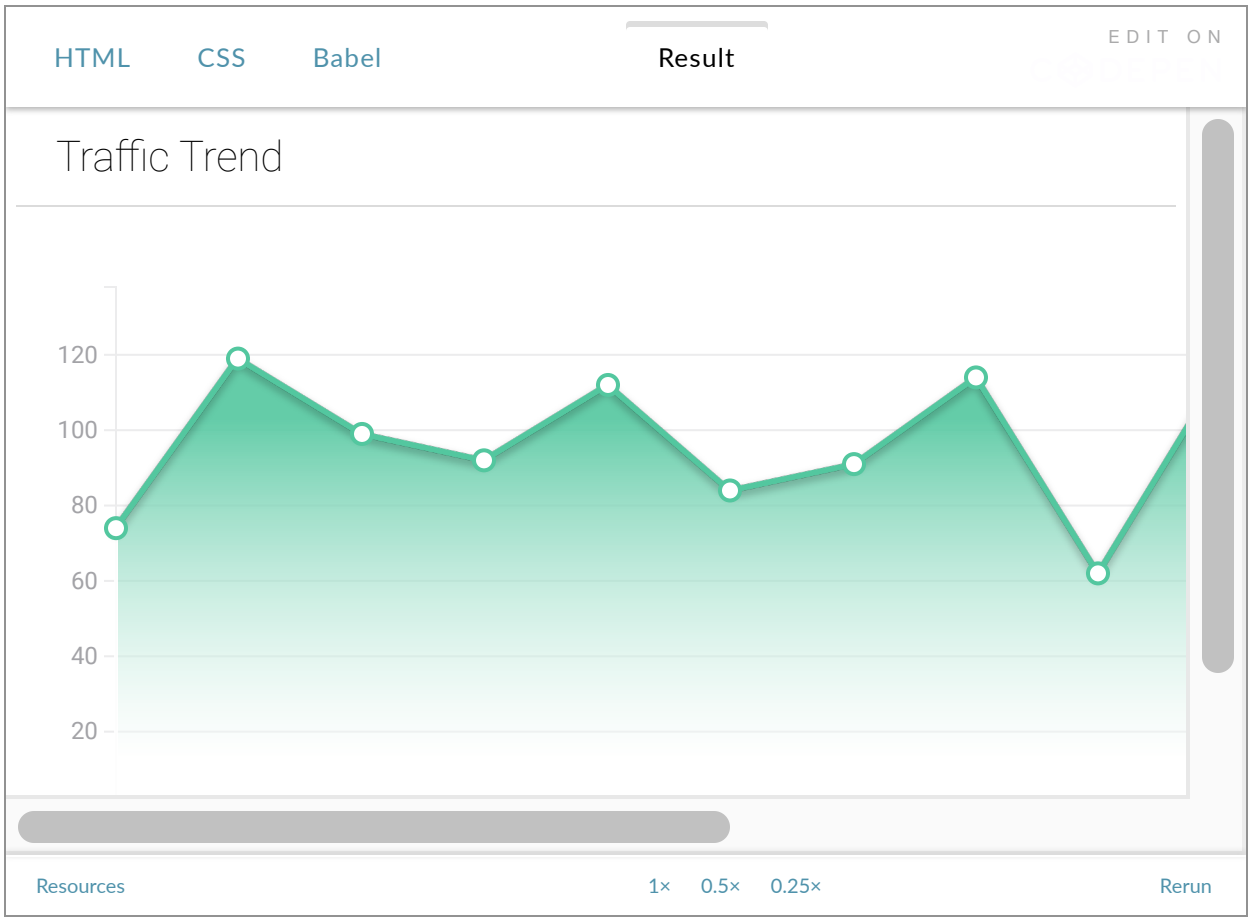
We would need a new method named `createDefs()`. Lets add it.

```
createDefs:function(element,i){
    var object;
    switch(element.type){
        case 'gradient':
            object=();
            break;
    }
    return object;
}
```

Again I have added some code to generate the data[]. Here is the output of the data[] that we are creating.

```
[
  {
    "day": "04-06-2016",
    "count": 88,
    "date": "2016-04-06T04:00:00.000Z"
  },
  {
    "day": "04-05-2016",
    "count": 58,
    "date": "2016-04-05T04:00:00.000Z"
  }
  .
  .
  .
]
```

At the end, we have to add `getDefaultProps()`, `getInitialState()`, `showTooltip`, `hideTooltip` and the `resizeMixin`. Here is the output.



Conclusion

We have gone through many different concept in this How to create reusable charts with React and D3 Part2. I hope this will provide a better idea and guidance. You can find the full demo here. See you again in the final part (part3) where we will complete the dashboard. Feel free to contact me in case of any disconnect.

Code (so far)

Code

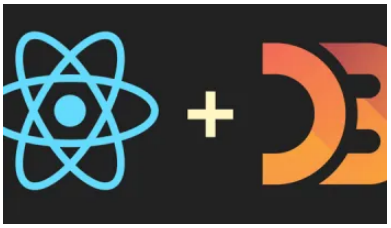
Demo (so far)

Demo

Related

How to create reusable charts with React and D3 Part1
In "D3.js"

How to create reusable charts with React and D3 Part3
In "D3.js"



How to integrate React and D3 – The right way
In "D3.js"

Filed Under: D3.js, React JS Tagged With: Area Chart, Code, d3, d3.js, example, integrate react and D3, Line Chart, React, step by step, tutorial, Visualization

Comments

Eric Van Horenbeeck says
February 17, 2017 at 7:59 pm

(Edit)

Hello Abhisek,
In various parts of your code you copy: ” var _self=this; ”
I understand that you want to preserve the original ‘this’ value but it is not clear to me why. Can you explain the underlying logic?
Thank you,
Eric

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.