A Developer Diary

{about:"code learn and share"}

Home Data Science Java JavaScript jBPM Tools Tips

About

April 7, 2016 By Abhisek Jana — 3 Comments (Edit)

How to create reusable charts with React and D3 Part3

Welcome to the How to create reusable charts with React and D3 Part3, this would be the final part of our series. We will first focus on adding the Donut Chart, then Stacked Bar Chart. Afterwards we will make the range filter (date range) to reload data. This last part should be simple and easier to follow along.

Create Donut Chart

The definition of the Donut chart would be very simple. We will define few attribute like custom `color` `scale`, `enable3d` etc. We will also have a child legend element to add the legends.

If you are new to Donut Chart, I have created the same chart using d3. Please refer the following tutorial.

Create 3D Donut Chart using D3.js

We will learn how to Create 3D Donut Chart using D3.js in this tutorial. We will use simple color function to add the perspective of third dimention. This chart looks great and can be used in any application. Please refer my previous article Create a simple Donut Chart using D3.js where we have created a … Continue reading

A Developer Diary 0

There are three section to the Donut Chart and we will create a React Component for all of them.

- DonutChartShadow
- DonutChartPath
- DonutChartLegend

Create DonutChartShadow

We can provide a 3d view of the chart by adding a shadow. We will dynamically calculate the color of the shadow so that once you set a color scale, your shadow would change automatically with it. I am using `d3.hsl()` built-in function to generate the shadow colors. Here is the code.

```
var DonutChartShadow=React.createClass({
    propTypes: {
        width:React.PropTypes.number,
        height:React.PropTypes.number,
        data:React.PropTypes.array,
        pie:React.PropTypes.func,
        color:React.PropTypes.func,
        innerRadiusRatio:React.PropTypes.number,
        shadowSize:React.PropTypes.number
    },

    getDefaultProps: function() {
        return {
            shadowSize:10
        };
    },

    componentWillMount:function(){

        var radius=this.props.height;

        var
outerRadius=radius/this.props.innerRadiusRatio+1;
        var innerRadius=outerRadius-
this.props.shadowSize;

        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) {

            var c=d3.hsl(_self.props.color(i));
            c=d3.hsl((c.h+5), (c.s -.07), (c.l -.10));

            return (

            )
        });
        return paths;
    },

    render:function(){

        var paths = this.createChart(this);

        return(

                {paths}

        )
    }
});
```

Create DonutChartPath

The Path and Shadow components are almost alike and we should combine them in one component. I have kept them separate here for simplicity however you should combine them.

```
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,
        innerRadiusRatio:React.PropTypes.number
    },
    componentWillMount:function(){

        var radius=this.props.height;

        var outerRadius=radius/2;
        var
innerRadius=radius/this.props.innerRadiusRatio;

        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}

        )
    }
});
```

Create DonutChartLegend

Here is the code to generate the legends. We are controlling the visibility of the legends based on the width of our chart. In case it narrows, we will hide the legends.

```
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,
        label:React.PropTypes.string,
        radius:React.PropTypes.number
    },
    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[_self.props.label]}


            )
        });
        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 legendY=this.props.height/2-
this.props.data.length*30/2;

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

                {texts}

        )
    }
});
```

Donut Chart

Now its time to put together our Donut chart. We will create the `pie` layout, define the color scale and set the state in the `componentWillMount()` method.

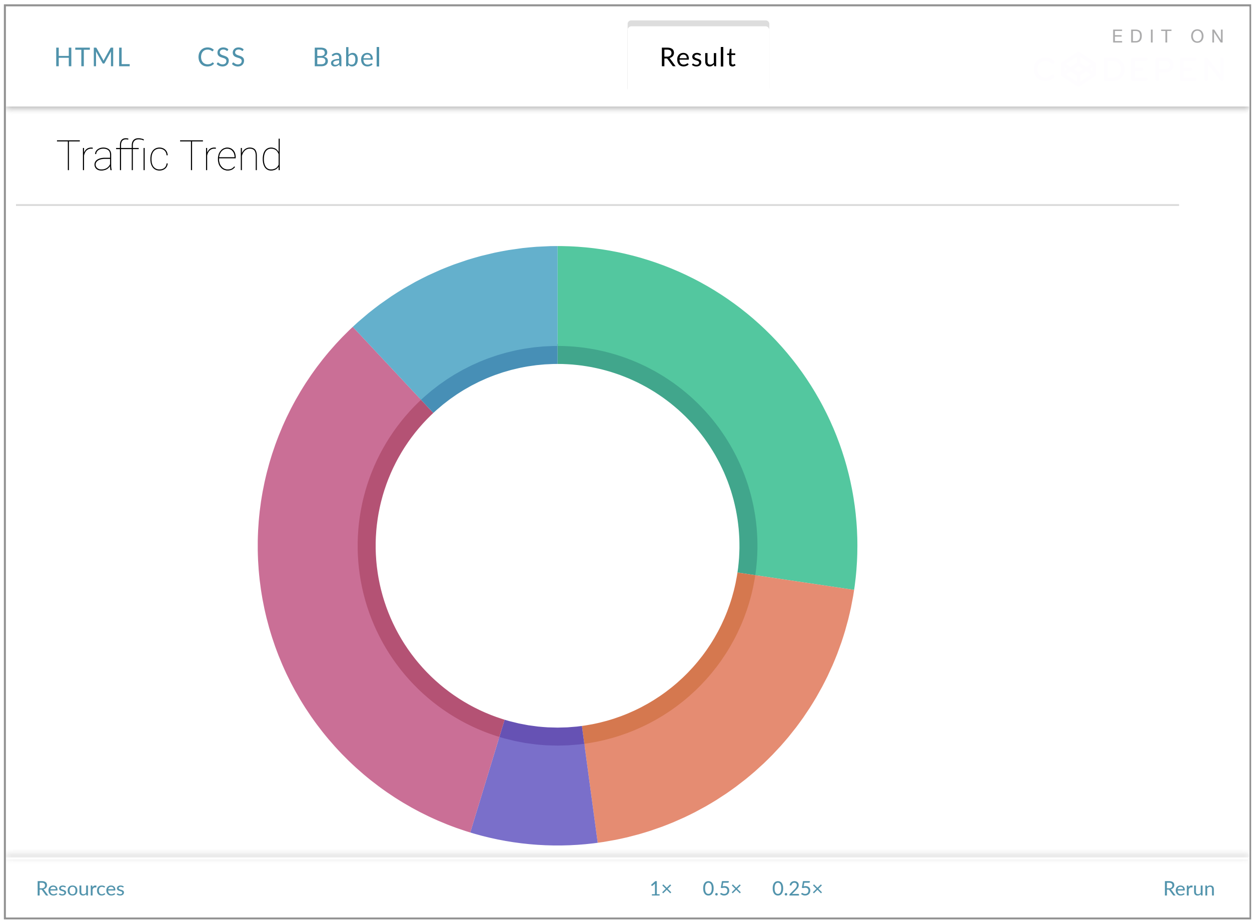
```
componentWillMount:function(){

    var _self=this;

    this.pie=d3.layout.pie()
        .value(function(d){return d[_self.props.point]})
        .padAngle(this.props.padAngle)
        .sort(null);

    this.color = d3.scale.ordinal()
        .range(this.props.color);

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

Inside the `render()` function we will follow these steps:

- Add the `DonutChartShadow` if the `enable3d` property has been set to true.
- Add the `DonutChartLegend` in the legend child element has been provided.
- Add the `DonutChartPath` to JSX directly.

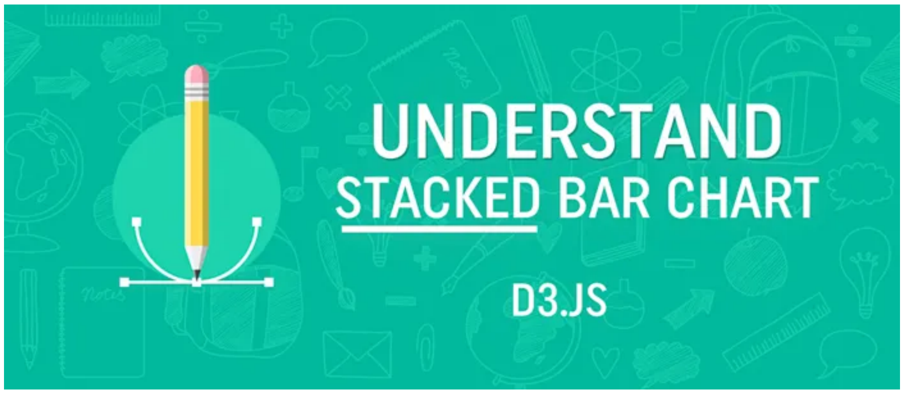
```
render:function(){

    var shadow;
    if(this.props.enable3d){
        shadow=(   );
    }

    var legend;

    if(this.props.children!=null){
        if(!Array.isArray(this.props.children)){
            if(this.props.children.type==='legend'){
                legend=();
            }
        }
    }

    return (
    );
}
```

We are randomly generating some data. Here is the demo.



Create Stacked Bar Chart

In case you are new to Stack Chart, I would recommend you to go through my earlier post on creating Stacked Bar Chart. This tutorial provides step by step guidance on how to create a stacked bar chart.



How to Create Stacked Bar Chart using d3.js

In this How to Create Stacked Bar Chart using d3.js post we will learn not only to code but the mathematical calculation behind creating a stacked bar chart using d3. Even if you have probably copy pasted a working version the code, I strongly recommend you to go though this tutorial in order to get … Continue reading

A Developer Diary 15

Here is our `StackChart` JSX element. We will define data, axis and grid. Based on the data it will automatically plot the chart. We will have a special two color scheme mode to have each bar a different color. Otherwise each category will be having a different color.

In the `createChart()` function we will define the `d3.layout.stack()` and transform our JSON data. We will also create the `xScale` and `yScale`.

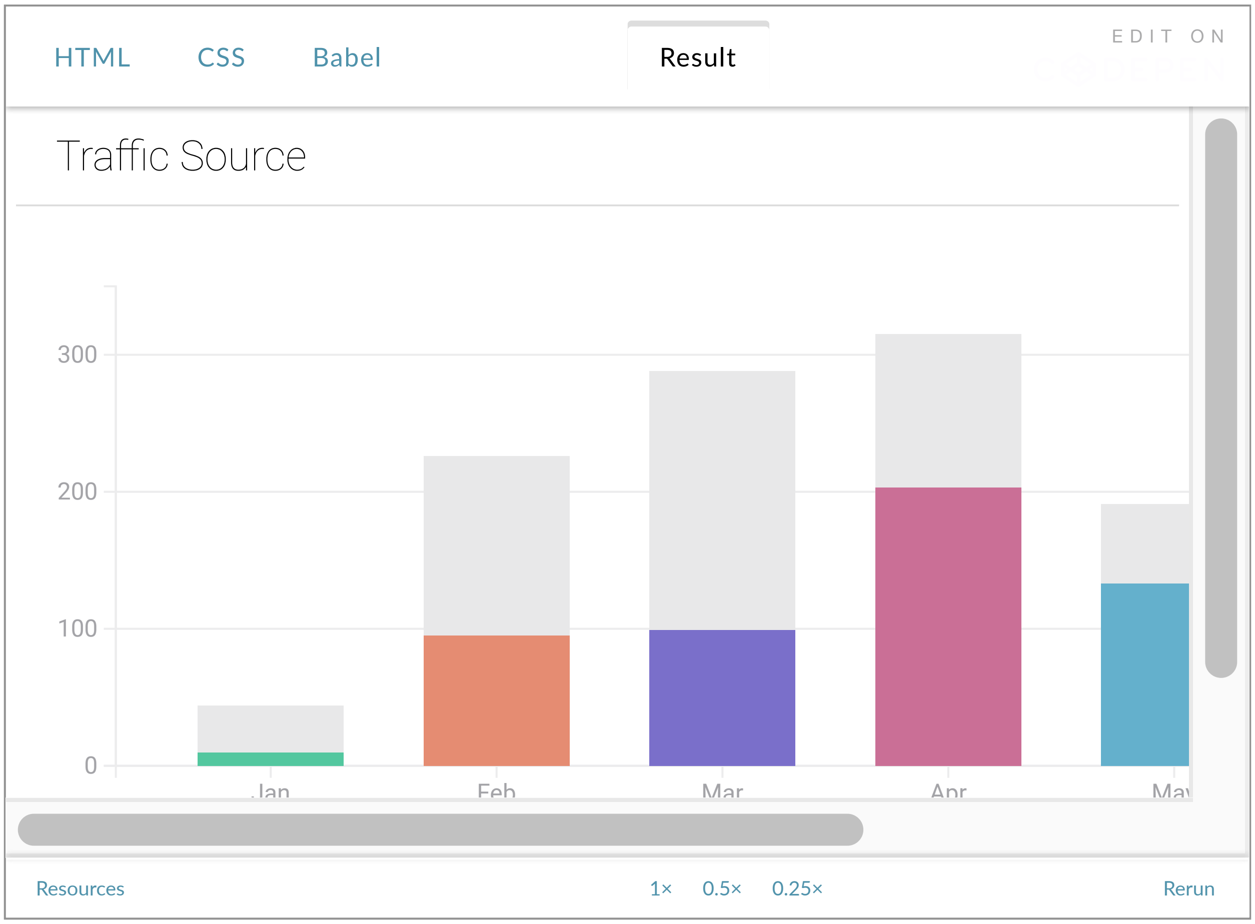
```
createChart:function(_self){
    if(this.props.color.length>0){
        this.color=d3.scale.ordinal()
            .range(this.props.color);
    }else{
        this.color=d3.scale.category20();
    }

    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.stacked = d3.layout.stack()
(_self.props.keys.map(function(key,i){
        return _self.props.data.map(function(d,j){
            return {x: d[_self.props.xData], y: d[key] };
        })
    }));

    this.xScale = d3.scale.ordinal()
        .rangeRoundBands([0, this.w], .35)
        .domain(this.stacked[0].map(function(d) { return
d.x; }));

    this.yScale = d3.scale.linear()
        .range([this.h, 0])
        .domain([0,
d3.max(this.stacked[this.stacked.length - 1], function(d)
{ return d.y0 + d.y; })])
        .nice();

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

```
},
```

The `render()` function should look familiar. We are first creating all the elements (axis & grid), however we are adding the bars separately. We need to go though the stacked array and create the `rect` element. We have our `twoColorScheme` defined here to change the color accordingly.

```
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)
        }
    }

    var bars=_self.stacked.map(function(data,i){

        var rects=data.map(function(d,j){

            var fill="";

            if(_self.props.twoColorScheme) {
                fill = _self.color(j);
                if (i > 0) {
                    fill = "#e8e8e9";
                }

            }

            return ();

        });

        var fill;
        if(!_self.props.twoColorScheme){
            fill=_self.color(i);
        }

        return
            {rects}

    });

    return (
    );
}
```

Again we have code to generate random data. I am not showing the data structure intentionally since my previous post on how to create stacked bar chart has it all. Let's go straight to the demo.



Add Range Filter

We will finish our Dashboard by adding the range filter functionality. We need to modify the Range Component. We have one Range component in each charts and a global one on top. We should refresh all the data by clicking on the global one, however the Chart's header range only should impact the specific chart's data.

- We have two states, 7 Day and 30 Days. So need a boolean(`defaultSelection`) to hold the state of this to values, false indicates 7 Days.

- If we call the global range as master, then if `master =true` we shall generate an event to have all the listeners (Charts) reset itself.
- In case the `master=false`, then we shall call the function named loadData passed as prop by passing the `defaultSelection` value.

Here is the code of the Range Class.

```
var Range=React.createClass({
    propTypes: {
        loadData:React.PropTypes.func,
        defaultSelection:React.PropTypes.bool,
        master:React.PropTypes.bool
    },
    getDefaultProps: function() {
        return {
            defaultSelection:false,
            master:false
        };
    },
    getInitialState:function(){
        return {
            defaultSelection:false
        };
    },

    componentWillReceiveProps: function(newProps) {

        if (newProps.defaultSelection !=
this.state.defaultSelection) {
            this.setState({defaultSelection:
newProps.defaultSelection});
        }
    },

    componentWillMount:function(){

this.setState({defaultSelection:this.props.defaultSelection});
    },

    toggleSection:function(){
        if(this.props.master){

            eventEmitter.emitEvent("reload",
[!this.state.defaultSelection]);
        }else {

this.props.loadData(!this.state.defaultSelection);
        }

this.setState({defaultSelection:!this.state.defaultSelection});
    },

    selectColor:function(){
        if(this.state.defaultSelection){
            this.fill7='#e58c72';
            this.fill30='#8f8f8f';
        }else{
            this.fill30='#e58c72';
            this.fill7='#8f8f8f';
        }
    },

    render:function(){

        this.selectColor();
        return(



                    ●
                    7 days


                    ●
                    30 days



        );
    }
});
```

So while defining the Range JSX element from each Chart's header, we will pass a function which would reload the data for that chart as the `loadData` prop. Our random data generation code would go inside this reload (in this case `reloadBarData()`) function.

In the `MainRangeSelection`, which is the global one, we just set the `master` prop value to true.

```
var MainRangeSelection=React.createClass({
    render:function(){
        return(
        );
    }
});
```

We will use `EventEmitter` to create our custom events. When the Range class emits the event named `reload` (below is the snippet), we can listen to it and update all the charts as required.

```
toggleSection:function(){
    if(this.props.master){

        eventEmitter.emitEvent("reload",
[!this.state.defaultSelection]);
    }else {

this.props.loadData(!this.state.defaultSelection);
    }

this.setState({defaultSelection:!this.state.defaultSelection});
},
```

We need to add the `addListener()` and `removeListener()` method to the `componentWillMount` and `componentWillUnmount` respectively. We should implement this not just for the charts, but for the cards as well. We probably want to create a mixin for this as well.

```
componentWillMount:function(){
    this.reloadBarData();
    this.reloadPieData();
    eventEmitter.addListener("reload",this.reloadData);

},
componentWillUnmount:function(){

eventEmitter.removeListener("reload",this.reloadData);

},
reloadData:function(defaultValue){
    this.reloadBarData(defaultValue);
    this.reloadPieData(defaultValue);
},
```

Here is the final version of the code.

Code

Final demo.

Demo

Conclusion

In this series we have go through many important topics of both React and D3. We have created reusable, customizable , extensible , responsive and configurable D3 charts in React.We have use this.props.children to define our components. The EventEmitter is a very helpful lightweight library. We have broken down our components into multiple reusable React classes and used them across the demo. At the end I hope this articles will help you to understand the concepts on How to create reusable charts with React and D3. Feel free to post your feedback, suggestions or questions in the comment section.

Related

How to create reusable charts with React and D3 Part2
In "D3.js"

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

Create Pie Charts using D3.js
In "D3.js"

Filed Under: D3.js, React JS | Tagged With: Charts, Code, Component, d3, d3.js, EventEmitter, example, integrate react and D3, JavaScript, Programming, React, React JS, React+d3, Reusable, step by step, Visualization

Subscribe to stay in loop

* indicates required

Email Address *

Subscribe

Comments

Prashanth says
July 9, 2016 at 9:26 am

(Edit)

Awesome 3 part series.

Have you implemented these dashboards using angular and d3js?

Reply

A Developer Diary says
July 11, 2016 at 12:41 am

(Edit)

Hi Prashanth,
Thanks for your feedback !

In order to use D3.js with Angular(1.x) Directive we need to use the watch function, which is not very efficient. I will have some of the tutorials posted soon.

Thanks

Reply

Samuel says

August 15, 2016 at 3:14 pm

(Edit)

Thanks a lot, your entire blog is great. I'm working with D3 and Angular and It seems easier work with D3 + React.

Reply

Leave a Reply

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

Comment *

Post Comment