

---

# DKAN Documentation

*Release*

**DKAN**

**Feb 24, 2017**



---

## Contents

---

<b>1</b>	<b>Table of Contents</b>
----------	--------------------------

<b>3</b>
----------



React Dash is a library for quickly building custom data visualization dashboards based on re-usable components.

- Chart Components Using NVD3
- Choropleth Maps in geoJson and topoJson
- HTML Components
- Custom Filtering
- Custom Data Handling
- CSV Integration
- DKAN API Integration



## Getting Started

### Starting a project with the boilerplate module

The *react-dash-boilerplate* project provides a starter project and a development environment for doing react-dash development. It's a good place to start.

- Make sure that you have npm installed on your system
- Install the react-dash-boilerplate project and its dependencies:

```
git clone https://github.com/NuCivic/react-dash-boilerplate.git
cd react-dash-boilerplate
npm install
npm run init
npm run start
```

- Visit <http://localhost:5000> on your system - if everything was successful, you should see an example dashboard
- Look at our [Developer's Guide](#) for next steps to customize your dash!

### Run the example project that ships with the react-dash project

If you can't get the boilerplate project to install, just install the react-dash library and use the example project as a starting place.

- Make sure that you have npm installed on your system
- Set up an npm project

```
npm init
npm install react-dash
npm run start
```

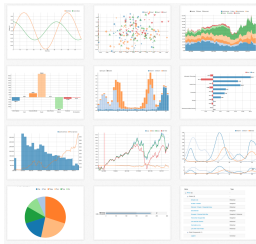
- Look in the `examples/` folder for the project source
- Take a look at our [Developer's Guide](#) for next steps to customize your dash!

## Building a DKAN project

Visit the [Drupal Module](#) page for more on Drupal Dkan development

## Features

### NVD3 Chart Components



NVD3 is a powerful Javascript library for doing advanced data visualizations. To see examples of what is possible with nvd3, visit <http://nvd3.org/examples/index.html>

### Filters

Filters allow end-users to filter and compare data. Common use cases include:

- Filter data by year
- Filter data by department
- Compare entities

#### Select Filter

#### Autocomplete Filter



Checkbox Filter

Map Components

Data Table

Metric Component

Text component

## Developer Guide

### Overview

Currently, there are two formats available for defining dashboards:

#### JSX

You can define a dashboard using familiar JSX syntax. Following is a complete example of a simple dashboard defined in JSX:

npm run jsx-example will load the following dashboard:

```
/**
 * This example shows how to build a basic dashboard using available components
 * and basic React JSX syntax.
 */
import 'bootstrap/dist/css/bootstrap.min.css';
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import { Dashboard, Region, Card, Metric, Chart } from '../src/ReactDashboard';

class App extends Component {
  render() {
    return (
      <Dashboard title="A Frivolous Dashboard Writ in JSX" doFilterRouting={false}>
        <link rel="stylesheet" type="text/css" href="https://maxcdn.bootstrapcdn.com/
        bootstrap/3.3.7/css/bootstrap.min.css" />
        <div className="row">
          <Card cardStyle="metric" className="col-md-4">
            <Metric
              caption='Caption A'
              data={[1]}
              background='#687a99'
              iconClass='fa fa-bed'
            />
          </Card>
          <Card cardStyle="metric" className="col-md-4">
            <Metric
              caption='Caption B'
              data={[2]}
              background='#689994'
              iconClass='fa fa-bomb'
            />
          </Card>
        </div>
      </Dashboard>
    );
  }
}
```

```

    </Card>
    <Card cardStyle="metric" className="col-md-4">
      <Metric
        caption='Caption C'
        data={[3]}
        background='#8f6899'
        iconClass='fa fa-bathtub'
      />
    </Card>
  </div>
  <div className="row">
    <Card cardStyle="chart" className="col-md-6">
      <Chart
        header="Foo"
        iconClass="fa fa-cloud"
        key="chart1"
        data={[{x: 'foo', y: 10}, {x: 'bar', y: 20}, {x: 'bax', y: 30}]}
        settings={
          {
            type: 'pieChart',
            height: 300
          }
        }
      />
    </Card>
    <Card cardStyle="chart" className="col-md-6">
      <Chart
        header="Eeny"
        iconClass="fa fa-cogs"
        key="chart2"
        data={[{x: 'Eeny', y: 1122}, {x: 'Meeny', y: 2220}, {x: 'Miney', y: 910}]}
        settings={
          {
            type: 'pieChart',
            height: 300,
          }
        }
      />
    </Card>
  </div>
</Dashboard>
)
}
}

document.addEventListener('DOMContentLoaded', function(event) {
  ReactDOM.render(<App/>, document.getElementById('root'));
});

```

## Javascript Settings object

In many cases it is useful to define dashboard configuration in JSON - allowing for database storage, programmatic creation, etc. Here is the above Dashboard defined as a Javascript object.

`npm run js-example` will load the following dashboard configuration at `localhost:3000`:

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import { settings } from './settings';
import { Dashboard } from '../../src/ReactDashboard';
import 'fixed-data-table/dist/fixed-data-table.min.css';

document.addEventListener('DOMContentLoaded', function(event) {
  ReactDOM.render(<Dashboard {...settings}/>, document.getElementById('root'));
});
```

## Complete Dashboard Application

The app at `/examples/app.js` runs an example of a more robust application with datahandling, filtering, etc.

`npm run start` will from the react-dash repository will run the application.

## Boilerplate project

The boilerplate module includes the above application in a complete development environment suitable for doing dashboard development.

```
git clone https://github.com/NuCivic/react-dash-boilerplate.git
cd react-dash-boilerplate
npm install
npm run init
npm run start
```

This will spin up the example application, and you can use it as a starting point for developing your own app.

## Building the Dashboard with Webpack

When development is done, you need to build the dashboard.

### standalone project

Standalone allows you to output a javascript bundle that you can embed in any website. If you are using the boilerplate project, run `npm run build_standalone`. If you are building from the example folder of the react-dash library run `npm run build`. It is also possible to provide an additional webpack configuration file that suits your development or production needs. Consult the [webpack documentation](#).

### dkan project

From inside of the `react_dashboard/app` directory run `npm run build_dkan`. This will output the compiled javascript and css bundles in the proper location for the `react_dashboard` module to load them. See the [react\\_dashboard docs](#) for greater detail.

## additional integrations

Interested in integrating another platform? Written an integration? File an [issue](#)

## Application Entry Point - app.js

```

1 import React, { Component } from 'react';
2 import ReactDOM from 'react-dom';
3 import { settings } from '../settings';
4 import { Router, Route, browserHistory } from 'react-router';
5 import { Dashboard, Dataset } from '../src/ReactDashboard';
6 let _settings;
7 import { omit } from 'lodash';
8
9 // get settings object from global, if available
10 if (typeof expressDashSettings !== "undefined") {
11   _settings = expressDashSettings;
12 } else {
13   _settings = settings;
14 }
15
16 // Extend Dashboard with our data fetch logic
17 // @TODO - this is a good case for a higher order function as mariano discussed
18 class Dash extends Dashboard {
19   getDashboardData(_appliedFilters) {
20     let dashData = Object.assign({}, this.state.data);
21     let dataKeys = Object.keys(this.props.dataResources);
22     let appliedFilters = _appliedFilters || Object.assign({}, this.state.
    ↪appliedFilters);
23
24     dataKeys.forEach(dataKey => {
25       let filters = this.getFilters(dataKey, appliedFilters);
26
27       this.fetchBackend(this.props.dataResources[dataKey]).then(data => {
28
29         // Note that, because of the shape of our data and the need for custom
    ↪processing, that we do our filtering AFTER we fetch. In other cases (The DKAN Dash
    ↪implementation for example), we use the appropriate filters to update our API
    ↪calls to return filtered data. The details of implementation are up to you, but we
    ↪suggest you stick with the appliedFilters pattern, which maintains the proper top
    ↪down data flow
30         dashData[dataKey] = this.applyFilters(data.hits, filters);
31         if (Object.keys(dashData).length === dataKeys.length) {
32           this.setState({data: dashData, isFetching: false});
33         }
34       }).catch(e => {
35         console.log('Error fetching dashboard data', e);
36       });
37     });
38   }
39
40   /**
41    * Use backends to fetch data and query the result
42    */
43   fetchBackend(fetcher) {
44     return new Promise((resolve, reject) => {
45       let dataset = new Dataset(omit(fetcher.fetchData, 'type'));
46       let queryObj = this.state.queryObj;
47       this.setState({isFetching: true, dataset: dataset});
48       dataset.fetch().then((data) => {
49         this.state.dataset.query(queryObj).then(queryRes => {
50           resolve(queryRes);

```

```

51     })
52     }).catch(e => {
53       reject(e);
54     });
55   });
56 }
57
58 // A bit of a trivial example of how to use filters to return filtered data
59 applyFilters(data, filters) {
60   let _data = data.slice(0), filterVal;
61
62   // perform required filtering based on filters obj
63   // In this case we need to compare the year value with the YearMonth value as 
64   ↪strings
65   if (filters && filters.length > 0) {
66     filters.forEach(f => {
67       if (f.field === 'YearMonth') {
68         filterVal = f.value[0].value || f.value[0];
69         _data = _data.filter(row => {
70           return row.YearMonth.toString().indexOf(filterVal.toString()) >= 0;
71         })
72       }
73     });
74   }
75   return _data;
76 }
77 }
78
79 // Now wrap the Dashboard so we can pass Routing info from the App
80 class MyDashboard extends Component {
81   render() {
82     let z = {};
83     z.appliedFilters = (this.state) ? this.state.appliedFilters : {};
84     const props = Object.assign({}, this.props, z, _settings);
85     return <Dash {...props}/>
86   }
87 }
88
89 // Wrap Dashboard component in router
90 class App extends Component {
91   render() {
92     return (
93       <div id="router-container">
94         <Router history={browserHistory}>
95           <Route path='*' component={MyDashboard} />
96           <Route path='/react-dashboard' component={MyDashboard} />
97         </Router>
98       </div>
99     )
100   }
101 }
102
103 // Now put it in the DOM!
104 document.addEventListener('DOMContentLoaded', function(event) {
105   ReactDOM.render(<App/>, document.getElementById('root'));
106 });

```

React dash is a library for building apps, not an app itself! We need to provide a data handling framework, and we do that in our `app.js`. Take a look at `/examples/app.js` and follow along with the explanations below.

## Extend Dashboard

### getDashboardData

We extend the dashboard to provide an implementation of the `getDashboardData` method.

Note that we assume that data is segmented into *dataResources*. Each *dataResource* or *dataKey* contains a discreet set of data. *dataResources* should be defined as an array *settings.js / props*. Each *dataKey* can contain data required for fetching given data. We leave the implementation details up to you.

The dashboard is initialized with `state.isFetching = false`. `getDashboardData` should set this state parameter to false when all data has returned.

In the example we loop through each *dataResource* and call *fetchBackend* which sets the response object to `state.data[dataKey]`

Also, `getDashboardData()` should set `state.isFetching = false` once all data has been returned.

As per normal REACT, the `setState` call will trigger a re-render of the dashboard, updating components as needed.

Note that we use a custom *applyFilters* method that maps current application state (*appliedFilters*) onto our data fetching code. You can use this opportunity to apply filter parameters to an API call, to remap current dashboard data, etc.

## Configuring the dashboard - settings.js

`settings.js` defines a declarative configuration for our react-dash.

A simple example:

```
import 'bootstrap/dist/css/bootstrap.min.css';

export var settings = {
  title: 'React-Dash Javascript Settings Example',
  doFilterRouting: false,
  regions: [
    {
      id: 'metrics-row',
      className: 'row',
      children: [
        {
          type: 'Metric',
          caption: 'Caption A',
          cardStyle: 'metric',
          iconClass: 'fa fa-bed',
          data: [1],
          background: '#687a99',
          className: 'col-md-4',
        },
        {
          type: 'Metric',
          caption: 'Caption A',
          cardStyle: 'metric',
          iconClass: 'fa fa-bed',
          data: [2],
        }
      ]
    }
  ]
}
```

```

        background: '#689994',
        className: 'col-md-4',
    },
    {
        type: 'Metric',
        caption: 'Caption A',
        cardStyle: 'metric',
        iconClass: 'fa fa-bed',
        data: [3],
        background: '#8f6899',
        className: 'col-md-4',
    }
]
},
{
    id: 'charts-row',
    className: 'row',
    children: [
        {
            type: 'Chart',
            cardStyle: 'chart',
            header: 'Foo',
            iconClass: 'fa fa-cloud',
            className: 'col-md-6',
            key: 'c1',
            data: [{x: 'foo', y: 10}, {x: 'bar', y: 20}, {x: 'baz', y: 30}],
            settings: {
                type: 'pieChart',
                height: 300
            }
        },
        {
            type: 'Chart',
            cardStyle: 'chart',
            header: 'Eeny',
            iconClass: 'fa fa-cloud',
            className: 'col-md-6',
            key: 'c2',
            data: [{x: 'eeny', y: 110}, {x: 'meeny', y: 920}, {x: 'mo', y: 430}],
            settings: {
                type: 'pieChart',
                height: 300
            }
        },
    ],
    // start DataTable Example
    {
        type: 'DataTable',
        header: 'Mi titulo',
        data:
        [
            [
                {foo: 1, bar: 2},
                {foo: 11, bar: 21},
                {foo: 111, bar: 222},
                {foo: 1, bar: 2},
            ]
        ],
        settings: {

```

```

        table: {
            rowHeight: 40,
            width: 800,
            maxHeight: 300,
            headerHeight: 40
        },
        columns: {
            flexGrow: 1,
            width: 150,
        },
        cells: {
            height: 40,
            width: 500,
        }
    },
    // end DataTable Example
}
]
}
}

```

## Regions

### Bootstrap rows

Regions are rendered into divs, allowing you to add bootstrap `row` class, and thus to organize the dashboard into rows using the responsive grid.

### Conditional Rendering

If `region.multi = true` then you can use conditional rendering to render the region. If this is the case, then the region should contain an `elements` object. The `elements` object has keys which each contain a different array of elements. The multi-region should also define a `datahandler`. The `datahandler` returns a string which is used as the key to chose an array from the `elements` object.

@@TODO add example

## Data Handling

React-dash can be instantiated without any data processing, simply by passing data as an object, in the specified format (see format, below). In this way, you can implment the react-dash as a view-layer of an application, or as a client to a service that provides data in the specified formats. The dashboard currently provides a basic framework for fetching and processing data.

## Overview

All components that extend the [Base Component](#), including the [Dashboard Component](#) can receive data in three ways, depending on how the component (or the dashboard) is configured:



## Raw Data

Raw data is passed via the `data` prop. If the data is in the correct format, as specified by the component specification, it will be rendered as is.

example:

```
{
  type: 'chart',
  settings: {
    x: 'val',
    y: 'key'
    // ...
  },
  data: [
    [{key: 'a', val: 1}, {key:'a', val: 2}, {key: 'a', val: 3}]
  ]
}
```

Here data is a an array containing a single series which represents variable a as a linear progression. Note that the series is contained as an array, as expected by most of our components. See [Data Format](#).

## Data Handlers (dataHandlers)

In React Dash, data handlers act similarly to reducers. They are given the dashboard's data and state, and reduce the input to a data object which is passed to a component as `props.data`.

See [Data Handlers](#)

## Backends

@@NOTE: The backends are kind of broken in v0.5 sorry example:

```
{
  // ... other component configuration
  data: {
    type: 'backend',
    backend: 'CSV',
    url: '/path/to/your.csv'
    // delimiter: '\t' // optionally specify a
  }
}
```

Data can be fetched using one of the existing data backends. Currently, react dashboard supports the following backends:

- CSV

## Data Handlers

Data handlers are similar in principle to Redux reducers. They take as input the global dashboard state, and the dashboard data, and return a data object which is passed as `props.data` to the component which defined the handler.

## **datahandler definition**

*Datahandlers* are defined as props at the component level - `props.datahandlers` is defined as an array of objects, where each object consists of a *name* property, as well as any number of additional properties, which are passed to the datahandler function as properties of the *handler* argument.

Consider:

```
// settings.js:
{
  type: 'Metric',
  caption: 'My Cool Metric',
  dataHandlers: [
    {
      name: 'getRandomMetric'
    },
    {
      name: 'multiplyByFactor',
      factor: 2
    },
    {
      name: 'multiplyByFactor',
      factor: 4
    }
  ]
}
```

Take a look at `examples/customDatahandlers.js` for an example implementation.

## **datahandler arguments**

### **componentData**

Data set on the component as `this.state.data` - this could come from a fetch call, be passed as props, or through some novel method on a custom component.

### **dashboardData**

Also referred to as `globalData` - this is the data available to the entire dashboard

### **handler**

The handler as defined in `settings.js`. Any properties set on the handler are available as `handler.foo`, etc

### **e**

Filters use the *e* property to capture the javascript event and pass it along for use in the handler

## appliedFilters

A global property of the dashboard which indicates what filters are applied at the global level. This object is of the form:

```
{
  filterValueOne: ["val1", "val2"],
  filterValueTwo: ["val3"]
}
```

## pipelineData

If datahandlers are chained, then *pipelineData* will be the return value of the previous datahandler in the chain. See [chaining](#)

## chaining

If the component.props.dataHandlers array has more than one datahandler then the return value from the first handler will be passed as *pipelineData* to subsequent handlers, in this way composition of components is possible, etc.

## Existing Data Handler Libraries

### common

Includes a library of common data transformations. @TODO document each handler, but for now you can look in `src/datahandlers/common.js`

### nvd3

A library of data handling functions for working with NVD3 chart data. @TODO document each handler, but for now you can look in `src/datahandlers/NVD3.js`

## Data Format

In most cases, data is considered as an array (`[]`). Multiple series of data can be represented as an array of arrays:

```
[
  [ {key: 1, val: 2}, {key: 1, val: 1}, {key: 1, val: 5} ],
  [ {key: 2, val: 4}, {key: 2, val: 5}, {key: 2, val: 7} ],
  // ...
]
```

Data for a [Metric Component](#) could be represented as `[1234]` where 1234 is the value passed to the metric.

Note that in most places we assume that single series and even single scalar values will be represented within an array.

This is not a hard and fast rule - [components](#) define their own data formats, but *dataHandlers* will make some assumptions about data, so it is good to follow these conventions.

## Backends

### NVD3

NVD3 provides the primary graphing engine in react-dash, via the `react-nvd3` module.

NVD3 has it's own, sometimes confused, opinions about the shape that data should come in. *React-dash* provides a few basic adaptors that allow us to transform data from our own, sometimes confused, data format. This allows us to keep data in our preferred format until the very last minute, when NVD3 needs it.

*Note* We considered implementing this in the Chart component, and hiding all of this from the library user, but thought it was better (if a bit more laborious) to allow the developer greater flexibility with the data.

The adaptors are implemented and used like any other `datahandler`:

### NVD3.returnChartSeries

Assume that our data is represented by two series of data in our preferred `format`:

```
[
  [ {a: 1, b: 1}, {a: 2, b: 2 }, {a:3, b: 2} ], // series one
  [ {foo: 1, bar: 1}, {foo: 2, bar: 2 }, {foo:3, bar: 2} ], // series two
]
```

Then the config in settings.js will look like this:

```
{
  type: 'Chart',
  data: [], // put data here
  settings: {
    // ... put settings here
  },
  dataHandlers: [
    {name :}
  ]
}
```

`NVD3.returnChartSeries` is suitable for use for all nvd3 chart types (I think) except `pieChart`, which uses:

### NVD3.toPieChartSeries

Similar to `returnChartSeries`. Expects the same input format but the output format is different.

## Data Backends

Data handling is up to you, but we are working on some backends to address common use cases.

### CSV

Coming soon...

## DKAN

Coming soon...

## CartoDB

Coming soon...

## Actions

In order to change dashboard state, components need to fire actions. Actions are handled at the Dashboard level and can update the state. For instance, filters fire actions that update `state.appliedFilters`. The `appliedFilters` object is passed to `dataHandlers` and can be used for filtering data or making decisions about component state.

All components that inherit from *BaseComponent* have a method called *emit*. *Emit* triggers actions and an *onAction method* that is automatically called when an action is fired from any component.

It's worth mentioning the *emit method* returns a regular javascript object. By convention it should have an *actionType* but the rest is up to you.

### action payload

Actions deliver a payload. The payload has the following attributes:

- **actionType** This determines which *onAction* hook is fired by the dashboard. You can define new *onAction* cases by extending the dashboard in your app. Filter components should define an *actionType*.
- **field** Field value defines the key on the `appliedFilters` object (`state.appliedFilters[field]`). It is often useful for the *field* value to correspond to a column in your data (for instance, filter by *year* field, where *year* is a column in your data)
- **value** In all of our current filter implementations, this is an array of objects:

```
[
  {label: '2012', value}
]
```

- **filter props** As a matter of convention we include all filter props as attributes to the payload. Some of these attributes are used to determine how to apply the filters. See [Filter](#)

**##appliedFilters** The react-dash has a *state.appliedFilters* object which encapsulates the current dashboard configuration. The object looks like:

```
appliedFilters: {
  fieldName1: ["foo"],
  fieldName2: [1, 10, 100]
  // ...
}
```

Applied filters should be used to filter data in your app, update application state, etc.

See also the section on [Filter](#)

## Dashboard Filtering

The user interface is represented through filters components. Filters trigger actions on the Dashboard which update `state.appliedFilters`. This triggers a re-render, allowing the Dashboard to respond to the updated application state - by re-fetching data, filtering existing data, reconfiguring the Dashboard etc.

### Filter

#### props

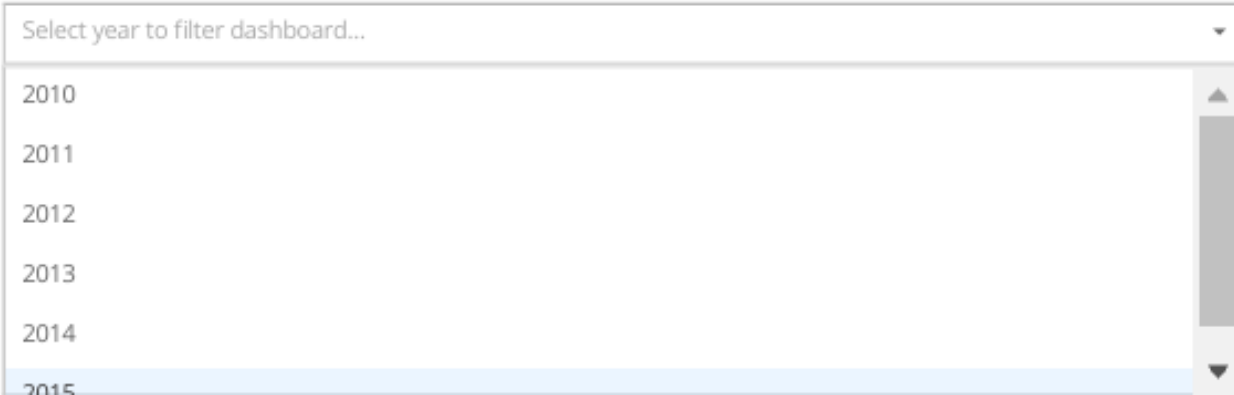
Filters take the following props: **type** The component type. Filter components should extend `BaseFilter` component. **field** This represents a key that will be used on the `state.appliedFilters` object. Depending on the implementation, it can also represent a field in the data. **action**: NOT IMPLEMENTED. (This will allow for sort or other filter actions in the future) **willFilter** An array of strings, each string corresponds with a `dataResource` or `dataKey`. This property indicates which part of the dashboard data the filter will be applied to. **disabledBy** An array of strings, where each string corresponds to a `field` value of another filter. When that field value is present in `appliedFilters`, the filter's html element will have the attribute `disabled=true`. **data / dataHandlers** As with all components, data can be passed as an object literal, or calculated via `dataHandlers`. It should be an array of arrays, where each array is a series of key value pairs. **initVal** An initial value for the filter.

### Example

Consider this filter definition:

```
{
  type: 'Autocomplete',
  className: 'col-md-6',
  name: 'specialty-autocomplete',
  id: 'specialty-autocomplete',
  className: 'specialty-autocomplete',
  field: 'YearMonth',
  action: 'filter', // sort / groupBy / etc
  willFilter: ['climateData'], // array of dkanDataResources keys that filters affect
  data: [
    [
      { label: '2010', value: '2010' },
      { label: '2011', value: '2011' },
      { label: '2012', value: '2012' },
      { label: '2013', value: '2013' },
      { label: '2014', value: '2014' },
      { label: '2015', value: '2015' },
    ]
  ],
  placeholder: 'Select year to filter dashboard...'
}
```

This will create the following select box:



If the user selects a year, the dashboard's `state.appliedFilters` object will be updated as follows:

```
state : {
  // ...
  appliedFilters: {
    YearMonth: ['2010']
  }
}
```

Out `dataHandlers` have access to the `appliedFilters` values and can filter data accordingly:

```
Object.keys(appliedFilters).forEach(k => {
  if (k === "year" && appliedFilters[k].length > 0) {
    _data = _data.filter(row => {
      return _inYear(row, appliedFilters[k]);
    })
  }
});
```

See also *Actions*.

## Components

### Autocomplete Component

Autocomplete uses the *react select component* <https://github.com/JedWatson/react-select>. As a result all the *react select* configurations can be passed in the element configuration.

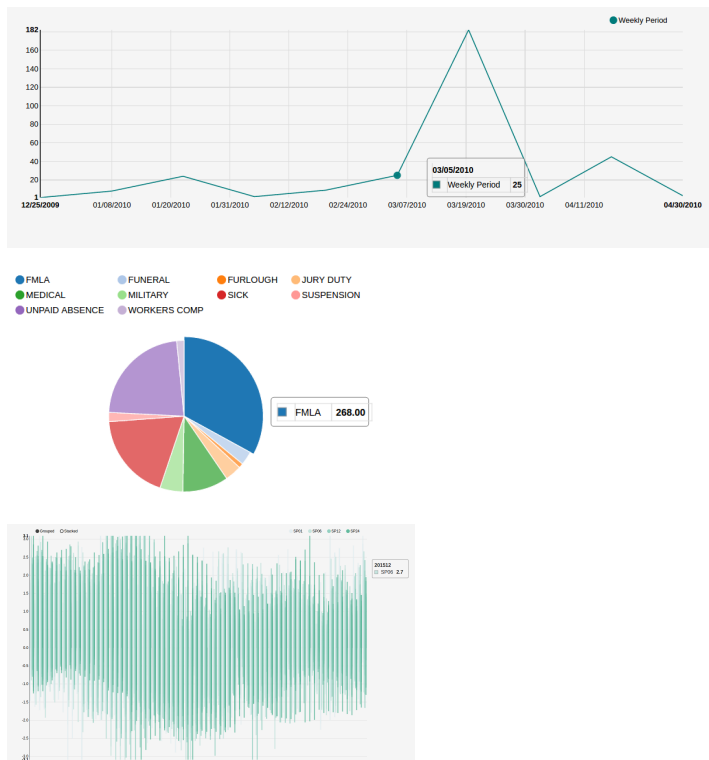
```
{
  type: 'Autocomplete',
  name: 'some-name',
  multi: true,
  url: 'http://localhost:3004/options?q={{keyword}}',
},
```

#### Available settings

- **willFilter:** An array of values where each value refers to a `dataResource` which the filter applies to
- **disabledBy:** An array of values where each value is the *field* attribute of another filter. If the referenced filter is enabled, the current filter will be disabled.
- **multi:** you can enable multi-value selection by setting `multi` to `true`.

- **field:** an arbitrary name, or key, for the filter - this will be the key for `state.appliedFilters[field]`. It is often helpful for the field name to correspond to a column variable in your data.
- **data:** an array with options (e.g.: [{ value: 'one', label: 'One' }])
- **dataHandlers:** Alternately, you can use a data handler to return the component data - this is useful if your autocomplete options are derived from dashboard data

## Chart Component



**Chart** component is a wrapper of the *react-nvd3* library ([@@LINK](#)), which is also a wrapper of the *nvd3* chart library. That means all the charts and options available in *nvd3* are also available in this component.

```
{
  header: 'Top',
  type: 'GACart',
  iconClass: 'glyphicon glyphicon-tree-conifer',
  settings: {
    id: 'lineChart2',
    type: 'discreteBarChart',
    x: 'label',
    y: 'value',
    height: 340,
    margin: {
      left: 38
    },
    color: ['#EA7E7E']
  },
  dataHandlers: ['getChartDaqta']
}
```



Notice that all the chart configuration goes inside the settings object.

If the x and y columns on your data already have the names you want, then you don't need to specify the x and y settings.

#### Available settings

- **settings** Settings are passed the React NVD3 module. See [React NVD3 documentation](#)
- **data** If raw data is being passed, data should be formatted as per the NVD3 data requirements which vary based on chart type. See the [NVD3 documentation and examples](#) which outline the proper shape for data.
- **dataHandlers** If you are using the react-dash internal data handling, make sure to pass data to one of the [NVD3 Data Handlers](#) as the final step of your data handling pipeline

## Dashboard Component

### Public Methods

#### method *componentWillMount*

Calls *getDashboard* when Dashboard is initially loaded.

#### method *componentDidUpdate*

Sets `state.isFetching` to true. Calls `getDashboardData()`.

#### method *getDashboardData*

This is an abstract method. Your Dashboard should implement its own *getDashboardData* method which sets a data object to the dashboard in the following way:

```
// fetch data
setState({data: data});
```

When all data has been returned, `getDashboardData()` should set `state.isFetching = false`

#### method *getFilters*

Given a *key*, determine which filters should be applied to `state.data[key]`;

#### method *getChildData*

Apply component *dataHandlers* and return data. Will return `props.data` if data is set as props on component.

## Data Table

This is a fork of facebook's [fixed-data-table](#) product.

Filter
10

agency	count	description	month	year	category	county	adult_or_ju
Saint Croix Tribal P	184	null	Jan	2010	Grand Total	Tribal	Total
Saint Croix Tribal P	16	All Other (Except Tr	Oct	2010	Other	Tribal	Total
Saint Croix Tribal P	13	All Other (Except Tr	May	2010	Other	Tribal	Total
Saint Croix Tribal P	13	All Other (Except Tr	June	2010	Other	Tribal	Total
Saint Croix Tribal P	13	All Other (Except Tr	Sept	2010	Other	Tribal	Total
Saint Croix Tribal P	12	All Other (Except Tr	April	2010	Other	Tribal	Total
Saint Croix Tribal P	11	All Other (Except Tr	Aug	2010	Other	Tribal	Total
Saint Croix Tribal P	9	All Other (Except Tr	Dec	2010	Other	Tribal	Total
Saint Croix Tribal P	8	All Other (Except Tr	July	2010	Other	Tribal	Total
Saint Croix Tribal P	7	All Other (Except Tr	Nov	2010	Other	Tribal	Total

1
2
3
4
5
6
7
8
9
10
»
»

Here's an example of the configuration:

```

type: 'DataTable',
header: 'Mi titulo',
data:
[
[
{foo: 1, bar: 2},
{foo: 11, bar: 21},
{foo: 111, bar: 222},
{foo: 1, bar: 2},
]
],
settings: {
table: {
rowHeight: 40,
width: 800,
maxHeight: 300,
headerHeight: 40
},
columns: {
flexGrow: 1,
width: 150,
},
cells: {
height: 40,
width: 500,
}
}
,

```

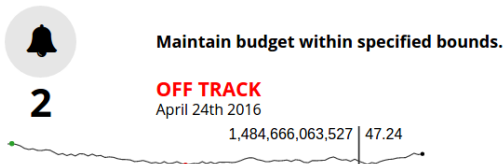
## Available settings

- *hideControls* Hide the controls
- **hideFilterHeader** Hide the filter controls

- **settings:**
  - **settings.table:** allows to configure all the properties for a table
  - **settings.columns:** allows to configure all the properties for columns
    - \* **overrides:** allows to override configurations for the column name number used as key.
  - **settings.cells:** allows to configure all the properties for cells
  - **overrides:** allows to override configurations for the cell in the row number used as key.
  - **settings.hideControls:** Hide row-numbers select in table header..
  - **settings.hideFilterHeader:** Hide filter box in table header.

## Goal Component

**React Dash** allows you to define goals to accomplish and are measured against the data. Goals be displayed by *increase, decrease, maintain or measure*.



```
{
  type: 'GAGoal',
  title: '',
  caption: 'number of schools enrollments',
  link: 'http://tootherplace.com',
  icon: 'glyphicon-gbp',
  startDate: '03/24/2016',
  endDate: '04/24/2016',
  startNumber: 0,
  endNumber: 200,
  action: 'increase',
  background: 'white',
  // trackStatus: 'function',
  tolerance: [
    {from: 0, to: 2, label: 'On Track', color: 'green'},
    {from: 2, to: 5, label: 'Needs Improvement', color: 'orange'},
    {from: 5, to: Infinity, label: 'Off Track', color: 'red'},
  ],
  spline: {
    height: 50,
  },
  fetchData: {type: 'function', name: 'getData'},
  metric: 'getRandomMetric'
}
```

### Available settings

- **caption:** caption text using in the component. Only plain text is allowed.
- **link:** a url to redirect users when they click in the goal.

- **startDate:** date when you start to measure your goal
- **endDate:** date when you needs to reach the goal.
- **startNumber:** amount of units you start with.
- **endNumber:** amout of units you want to reach.
- **action:** the action you want to accomplish.

There are 6 possible values:

- *increase*: your goal is to increase the number of units. If the number of units are equal or greater than the `endNumber` then goal is on track.
- *decrease*: your goal is to decrease the number of units. If the number of units are equal or lower than the `endNumber` then goal is on track.
- *maintain\_above*: this action is very similar to the increase action except `startNumber` and `endNumber` should be set at the same number.
- *maintain\_below*: this action is very similar to the decrease action except `startNumber` and `endNumber` should be set at the same number.
- *measure*: in this case you don't want to reach a goal but just display a mesure.
- *tolerance*: allow you to define a tolerance to define the status of your goal.

Let's take a look at the above example. In that case if your deviation is between 0 and 2 then the *OnTrack* label will be displayed because the first item of tolerance will be selected.

Deviation is computed by projecting the number of units based on the *startDate*, *endDate* and *endNumber* and using a linear function. You can override the *getTracker* and the *trackStatus* functions if this projection doesn't fit with your needs.

- *spline*: you can choose to additionally show a spline chart below the goal. If you choose to display the goal then you can set an object with the configuration needed to display the spline (e.g.: height).

## Loader

**Loader** allows components to display a loader while they are fetching data.

```
class MyComponent extends BaseComponent {
  render() {
    return (
      <Loader isFetching={this.state.isFetching}>
        <MyComponent>
          //...
        </MyComponent>
      </Loader>
    );
  }
}
```

As soon as *state.isFetching* is true then all the components inside and will display.

*ifFetching* is passed to components as props from the Dashboard `state.isFetching`

## Markup

Markup component allows you to embed arbitrary html markup within your react dashboard layout.

For example - a static html list:

```
{
  type: 'Markup',
  content: '<div>\n
    <ul>\n
      <li>FOO</li>\n
      <li>BAR</li>\n
      <li>BAZ</li>\n
    </ul>\n
  </div>'
}
```

### Available settings

- **content:** the html content to display.

## Metric Component



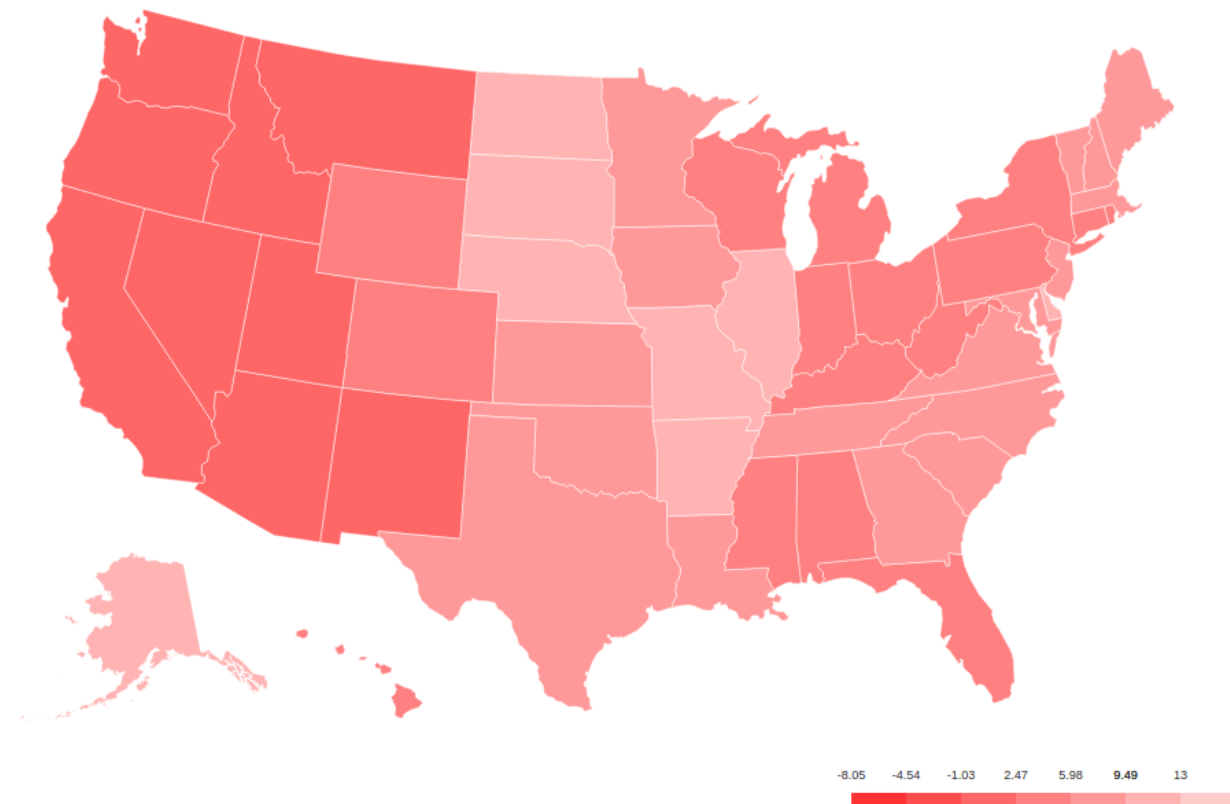
**Metrics** are intended to display a single value to the end-user.

```
{
  type: 'Metric',
  cardStyle: 'metric',
  iconClass: 'fa fa-clock',
  background: '#9F3E69',
  data: ['Provided Value'],
  caption: 'New Users',
}
```

### Available settings

- **background:** the background color to be used for this metric.
- **caption:** a description to be displayed
- **cardStyle:** REQUIRED: must be 'metric'
- **iconClass:** font-awesome icon class
- **data:** a value for the metric. It should be a scalar value contained within an array
- **dataHandlers:** an array containing dataHandler object(s)
- **options:** an array with options (e.g.: [{ value: 'one', label: 'One' }])

## Choropleth Maps



@@TODO UPDATE - !!!this is out of date!!! The **Choropleth** element provides a choropleth map (also known as a “heat map”) and a legend. The component uses a set of functions (*choroplethFunctions*) to map domain data to map polygons. The following elements are required to generate the Choropleth:

### Map Data

Map data provides features suitable for rendering a d3 map. Two formats are supported: **topojson** and **geojson**.

### Domain Data

Domain data provides the statistical data necessary to calculate the levels of the choropleth. As with all components, this can be provided by the *globalData* parameter, or fetched via a custom function or using any of the available backends.

### Configuration object

```
{
  type: 'Choropleth',
  format: 'geojson',
  fetchData: {
    url: './data/apollo-parsed-1737-325_0.csv',
  }
}
```

```

    type: 'backend',
    backend: 'csv',
    // delimiter: '\t'
  },
  id: 'Choropleth',
  dataKeyField: 'Zone',
  dataValueField: 'Total Observers',
  geometryKeyField: 'name',
  geometry: './data/zones.geojson', // topojson or geojson
  projection: 'equiarectangular',
  scaleDenominator: .7,
  borderColor: '#000000',
  noDataColor: '#F3F3F3',
  dataClassification: 'equidistant',
  legend: {
    // startColor: 'red',
    // endColor: 'yellow',
    classesCount: 5,
    palleteKey: 'GnBu',
    pallete: ['#f0f9e8', '#bae4bc', '#7bccc4', '#43a2ca', '#0868ac'],
    domainStartValue: '',
    domainEndValue: '',
  }
  // customMin: '',
  // customMax: '',
  // topologyObject: 'counties'
}

```

## Settings

- **format:** *[string]* type of geometry file to be used. Actually geojson and topojson geometries are supported.
- **geometry::** *[string]* path to either a geojson or topojson file.
- **geometryKeyField** (geojson): *[string]* name of the property in the geometry file that will be used to join the domain data with the proper polygon.
- **dataKeyField:** *[string]* field in the domain data that will be used to join join the domain data with the proper polygon.
- **dataValueField:** *[string]* field in the domain data to calculate the levels of the choropleth.
- **projection:** *[string]* the projection to draw the geometry. Available projections can be found at <https://github.com/d3/d3/wiki/Geo-Projections>.
- **scaleDenominator:** *[number]* a number to scale the map according to an arbitrary factor - experiment to find the best result
- **borderColor:** *[string]* border color for each shape in the geometry
- **noDataColor:** *[string]* shape color when no data is available in a given polygon.
- **startColor(linear scale):** *[string]* color mapped to the lowest value in the domain data.
- **endColor(linear scale):** *[string]* color mapped to the highest value in the domain data.
- **dataClassification:** *[string]* kind of scale to be used for data classification. Linear and Equidistant scales are supported.
- **legend**
  - **classesCount** the number of ranges to divide the domain data into

- **pallette** An array of css colors to represent the choro[pleth gradient]

## Theming

### Bootstrap grid

React Dash uses bootstrap responsive grid. Full docs are [here](#)

Layout is defined as an array of regions. Each region represents a bootstrap row, as follows:

```
regions: [
  {
    className: 'row',
    children: [
      {
        type: 'Metric',
        value: 'A',
        className: 'col-4-md'
      },
      {
        type: 'Metric',
        value: 'B',
        className: 'col-4-md'
      },
      {
        type: 'Metric',
        value: 'B',
        className: 'col-4-md'
      }
    ]
  }
]
```

Note that we're just using the bootstrap classes to define the responsive grid.

### cards

Cards allow you to use pre-defined themed layouts at the component level. See [Cards](#). To enable card layout, add *cardStyle* prop to your component in settings.js:

```
{
  type: 'Chart',
  cardStyle: 'chart',
  header: 'Card renders headers!',
  oconClass: 'fa fa-clock'
  // .... your settings follow
}
```

Components rendered as Cards (ie, components with *cardStyle* prop) can accept the following props: **cardStyle** Provides css class for additional styling **iconClass** Font awesome icon class will render icon to the left of any header text (use format *fa fa-icon*) **header** Render header text

### custom css

The `index.html` file in the examples project loads `static/custom.css`. Add custom css here.



## inline styles

Define a style object in `settings.js`:

```
{
  type: 'yourComponentType',
  style: {height: '100%', maxWidth: '60%', fontFamily: '"Times New Roman", Georgia, ↵
↵Serif'}
```

## Implementation Examples

- [Hours Worked Performance Dashboard](#)
- [Current Library Example](#)
- [UCR Arrest Data Dashboard](#)

## Ecosystem

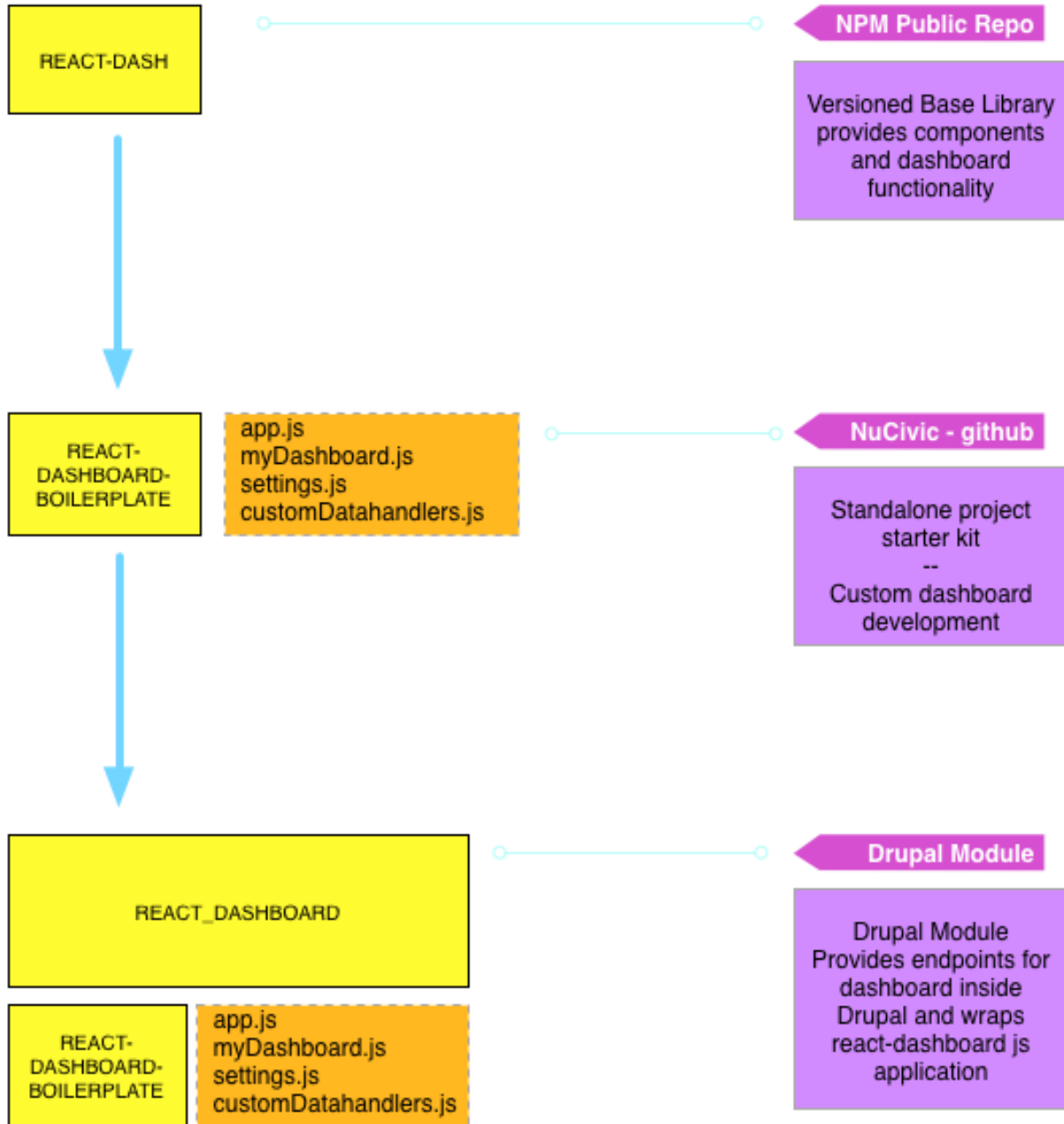
React-dash is an npm library that serves as a toolkit for doing rapid prototyping and development.

In order to further speed development, we also provide a [boilerplate module](#) with which to build standalone projects.

For Drupal and DKAN development, we provide a [Drupal Module](#) which allows you to easily embed a compiled dashboard into a drupal page, provides menu callbacks, and other useful features.



## REACT DASH -- DKAN ECOSYSTEM



## Contribute to React Dash

Visit our Github Page to:

- Report a bug

- Submit a patch or pull request
- Suggest an improvement
- Show us what you have built!