DKAN Documentation

Release

DKAN

Jan 20, 2017

Contents

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

CHAPTER 1

Table of Contents

1.1 Getting Started

1.1.1 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-dashboard-boilerplate project and its dependencies:

```
git clone https://github.com/NuCivic/react-dashboard-boilerplate.git
cd react-dashboard-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!

1.1.2 Copying the example project from the react-dash repo

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

- Make sure that you have npm installed on your system
- Clone the react-dash repository

```
git clone https://github.com/NuCivic/react-dashboard.git
```

- Install dependencies and start the development server

```
npm install
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!

1.1.3 Building a DKAN project

Visit the Drupal Module page for more on Drupal Dkan development

1.2 Developer Guide

1.2.1 Overview

The best way to start a react-dash project is to install the boilerplate module. See *Getting Started*.

Once the project is installed, you will have a directory structure that looks something like this:

```
package.json
node_modules/
webpack.config
index.html
dist/
  bundle.js
  bundle.css
src/
 app.js
 settings.js
 customDatahandlers.js
 static/
   custom.css
   your.img
```

The most important files here are in the `src/` directory. Not counting static files (image resources, css, etc), you should only need to modify the following files:

app.js

This file contains the boilerplate code to load a dashboard component into the root element of your index.html file. It is possible, but not necessary, to do initial preparatory work here, for instance fetching dashboard data (see Initializeing Dashboard Data)

settings.js

This file contains a javascript object with a declarative configuration for the dashboard. Here we define the dashboard components and settings.

customDatahandlers.js

A library of functions that we use to preprocess data.

1.2.2 Building the Dashboard

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

standalone project

Standalone allows you to ouput 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

1.2.3 Application Entry Point - app.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';
import { settings } from './settings';
import { Router, Route, browserHistory } from 'react-router';
import { Dashboard } from '../src/ReactDashboard';
let _settings;
if (typeof expressDashSettings != "undefined") {
   _settings = expressDashSettings;
} else {
  _settings = settings;
}

// We extend the Dashboard so we can pass Routing info from the App
class MyDashboard extends Component {
  render() {
    let z = {};
    z.appliedFilters = (this.state) ? this.state.appliedFiltersi : {};
    const props = Object.assign({}, this.props, z, _settings);
    return <Dashboard {...props}/>
  }
}

// Wrap Dashboard component in router
class App extends Component {
  render() {
    return (
      <div id="router-container">
        <Router history={browserHistory}>
          <Route path='*' component={MyDashboard} />
          <Route path='/react-dashboard' component={MyDashboard} />
        </Router>
      </div>
    )
  }
}

// Now put it in the DOM!
document.addEventListener('DOMContentLoaded', function(event) {
    ReactDOM.render(<App/>, document.getElementById('root'));
});
```

If you need to do custom initialization of data that cannot be accomodated with the existing backends, do it here. @@TODO Add example of custom fetch code here

1.2.4 Configuring the dashboard - settings.js

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

Lets look at a few examples.

A NOTE ABOUT ENVIRONMENTS The examples assume that you are using the react-dashboard-boilerplate module or a similar development environment which will load the examples from a settings.js file.

If you are using gist files with react-dash-server you will need to remove the export statement and convert the javascript settings object to JSON.

EXAMPLE 1 - Simple pie chart with header

Our hello world example consists of a single pie chart with some static values.

```
export var settings = {
  title: 'Hello World',
  components: [
    {
      type: 'Chart',
      cardStyle: 'Chart',
      header: 'My Pie Chart',
      data: [[{x: 1, y: 40}, {x: 2, y: 40}, {x: 3, y: 20}]],
      dataHandlers: ['NVD3.toPieChartSeries'],
      settings: {
        type: 'pieChart',
        x: 'x',
        y: 'y',
        height: '600'
      },
    }
  ]
}
```

Note that we use a datahandler to convert from our standard data format (an array of series). This is to account for the data 'shape' that nvd3 expects for its pie chart. See NVD3 Examples for a better understandoing of how NVD3 expects data to be formatted.

EXAMPLE 2 - Regions

```
export var settings = {
  title: 'Hello World',
  components: [
    {
      type: 'Chart',
      cardStyle: 'Chart',
      header: 'My Pie Chart',
      data: [[{x: 1, y: 40}, {x: 2, y: 40}, {x: 3, y: 20}]],
      dataHandlers: ['NVD3.toPieChartSeries'],
      settings: {
        type: 'pieChart',
        x: 'x',
        y: 'y',
        height: '600'
      },
    },
    {
      type: 'Region',
      className: 'row', // row class is used by twitter Bootstrap
      children: [
        {
          type: 'Metric',
          cardStyle: 'Metric',
          iconClass: 'fa fa-level-up',
          className: 'col-md-4', // col class used by twitter Bootstrap
          caption: 'Test A',
          data: ['A'] // an arbitrary value for our example
        },
        {
          type: 'Metric',
          cardStyle: 'Metric',
          iconClass: 'fa fa-level-down',
          className: 'col-md-4',
          background: '#53ACC9',
          caption: 'Test B',
          data: ['B']
        },
        {
          type: 'Metric',
          cardStyle: 'Metric',
          iconClass: 'fa fa-fire',
          caption: 'Test C',
          background: '#C97053',
          className: 'col-md-4',
          data: ['C']
        }
      ]
    }
  ]
}
```

We're starting to introduce more dashboard functionality here, including the use of the Bootstrap grid, by using regions, which have `className: 'row'` and children which have `className: 'col-md-4'`. More on the Metric component here More on theming and bootstrap grid here More on the Region component here

Fecthing Data and Beyond

For now, use the /examples/settings.js file as a guide to understand some of the more complex applications of the dashboard. It includes various examples of fetching and handling data, different chart configurations etc.

1.2.5 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.

Backends

example:

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

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

- CSV

Custom Data Handlers

Data handlers allow you to write custom code to determine how to generate component data or dashboard data.

An example of a component that uses a data handler to generate its data:

in settings.js:

```
{
  // ...other component configuration
  dataHandlers: [
    {
      name: getCustomData,
      fields: ['field1', 'field2']
    }
  ]
}
```

in customDatahanders.js

```
getCustomData: function (componentData, dashboardData, handler, e, appliedFilters,␣
↪pipelineData) {
  let _data = dashboardData;
  let fields = handler.fields;
  return  _.data.map(row => {
    let newRow = {};
    fields.forEach(field => {
        newRow[field] = row[field];
    });
    return newRow;
  });
}
```

In this example, we tell our component to use a custom datahandler called getCustomData. We use an array of field names to select a subset of the dashboard's global data. See datahandler settings object and datahandler paramaters below:

datahandler settings object

Datahandlers are defined in `settings.js` as an object with a unique `name` paramter. Dot notation can be used in the name to provide structure to your library of datahandlers, if needed.

All paramters except for `name` will be passed to the datahandler function as paramaters to the `handler` argument, and can be used example:

```
// in settings.js
{
  type: 'metricComponent'
  dataHandlers: [
    {
      name: 'anotherCustomeHandler',
      arg1: {foo: 'bar', bar: 'baz'},
      arg2: [1,2,3],
      // etc - any valid javascript/json data can be passed to the data handler
    }
  ]
}
```

These attributes can now be used within `anotherCustomHandler` by accessing `handler.arg1`, `handler.arg2`, etc.

datahandler paramaters

Data handler functions receive the following paramaters:

- **componentData** - any data set on the calling component
- **dashboardData** - also known as globalData, data set at the top level of the dashboard
- **handler** - the handler object as defined in settings.js. All paramaters, except the required `name` paramter will be properties of this object, accessible inside the handler's scope
- **e** - if the datahandler is called after an action, the jevascript event which fired the action. Useful for handling filter events and user interactions which update data. See Actions
- **appliedFilters** - Any filters which have been applied on the dashboard. See Actions
- **pipelineData** - If the component has defined an array of datahandlers, subsequent datahandlers will be passed the return value from the previous handler, otherwise `undefined`. See *chaining*

chaining

Data handlers can be chained, in which case the return value from each handler is passed to the following handler in the chain as `pipelineData`. A trivial example follows:

```
// assume that globalData is as follows:
{
    seriesOne: [{key: 1, val: 1}, {key: 1, val: 2}, {key: 2, val:2 }],
    seriesTwo: [{key: a, val: 11}, {key: b, val: 2}, {key: c, val:6 }]
    // ...
}

// settings.js
{
  type: 'Metric',
  dataHandlers: [
    {
      name: 'getSeriesByIndex',
      index: 'seriesOne'
    },
    {
      name: 'addOne',
      x: 'val'
    },
    {
      name: 'double',
      x: 'val'
    }
  ]
}
```

```
// customDatahandlers.js
/**
 * returns keyed data from globalData
 **/
function getSeriesByIndex(componentData, dashboardData, handler, e, appliedFilters,␣
↪pipelineData) {
  return [globalData[handler.index]]
}

/**
 * Adds one to all values
 **/
function addOne(componentData, dashboardData, handler, e, appliedFilters,␣
↪pipelineData) {
  // check for pipeline data first, or use componentData if exists
  let _data = pipelineData || componentData || [];
  _data.map(series => {
    series.forEach(row => {
      row[handler.x] = row[handler.x]*2
    });
    return series;
  })
}
```

I'll leave it as an excercise for you to implement the `double` dataHandler :)

registering data handlers

In order for everything to work, data handlers must be registered as follows:

```
// customDatahandlers.js
import DataHandler from 'react-dash';

function exampleHandler(componentData, dashboardData, handler, e, appliedFilters,␣
↪pipelineData) {
  // ... your handler code
}

DataHandler.set('exampleHandler', exampleHandler);

// OR:
let handlers = {
  handler1: function(componentData, dashboardData, handler, e, appliedFilters,␣
↪pipelineData) {
    // ... your code
  },

  handler2: function (componentData, dashboardData, handler, e, appliedFilters,␣
↪pipelineData) {
    // ... your code
  },

  // ...
}

for (let k in handlers) {
  DataHandler.set(k, customDataHandlers[k]);
}
```

@@TODO Provided Data Handlers

Global Data

Data which is set to the top-level Dashboard component is passed to all components as a `globalData` prop. It is also available inside of data handlers as the `dashboardData` argument. Data is assigned to the dashboard as a whole in the same way as it is assigned components, using data, backends, or customDatahandlers.

@@TODO EXAMPLE HERE

Data Handlers

In order to facilitate the custom handling of data we have introduced the concept of datahandlers. Datahandlers are functions that transform data - they can accept arbitrary paramaters, and have access to the following arguments:

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:

‘‘javascript // 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 deffined 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 pf the form:
```javascript
{
    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.

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 mertric.

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 userd like any other datahandler:

NVD3.returnChartSeries

Assume that our data is represeted 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.

1.2.6 Data Backends

CSV

1.2.7 Actions

Sometimes you need to tell other components about a change that happened in your dashboard. For example, a change in the underlying dashboard data after adding a new selection in the autocomplete.

This is handled through **actions**.

All components 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.

@@TODO update / verify example

```
// Component emitting a change
onClick(){
  this.emit({
    actionType: 'CHANGE',
    data: data
  });
}

// Component receiving a change
onAction(action){
  switch(action.actionType){
    case 'CHANGE':
      // Do some in
      break;
  }
}
```

1.2.8 Filter components

Filters

Filters allow data to be filtered based on user input, application state, or other custom logic. Filters use dom events and custom data handlers to provide filtered data.

Component-level filters

Filters can be used to allow user input which controls the data at the component level. Filters use dataHandlers, along with user input, to determine how to filter component data. Filters are configured as follows

Filter Paramaters

Filter paramaters are serialized to the url, allowing the dashboard to be loaded with a set of filters already applied. The url query string is serialized according to the following scheme:

`http://yoursite.com/dashboard/cid1=key1_val1&cid_1=key1_val2&cid2=key2_val3`

```
{
  cid1: {
    key1 : ['val1', 'val2']
  },
  cid2: {
    key2 : val2
  }
}
```

Components recieve their `ownParams` as props. So for copoment with *cid1*:

`component.props.ownParams = { key1:  ['val1', 'val2'] }`

```
//@@TODO
```

Dashboard-level filters

@@TODO Autocomplete / Actions / data handlers

Theming

Dashboard-level theming

The **React Dash** comes with default styles, but you can also customize them by importing a stylesheet.

```
// file: entry point
// standalone.js or dkan.js
import 'stylesheets/custom.css'
```

Currently you can use either a *css* or a *sass* file. You can also add import sentences inside to split the files. It's good to have a separate stylesheet for each component you are overriding.

Cards

@@TODO clarify If a *cardStyle* property is specified, the component will be rendered inside a car div.

Componentlevel theming

Components can take a style object as follows:

```
style: {backgroundColor: 'red', fontSize: '1em', margin: '1em'}
```

1.2.9 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.

Usually you won't need to extend this component. Autocomplete has standard behavior and is highly configurable.

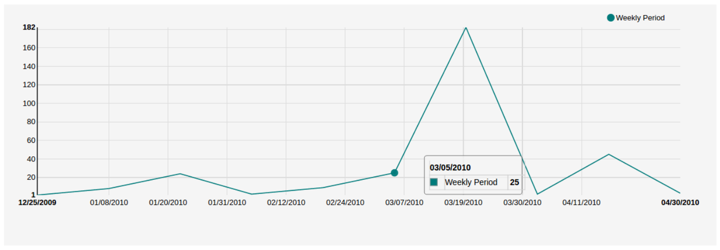
```
{
  type: 'Autocomplete',
  name: 'some-name',
  multi: true,
  url: 'http://localhost:3004/options?q={{keyword}}',
},
```

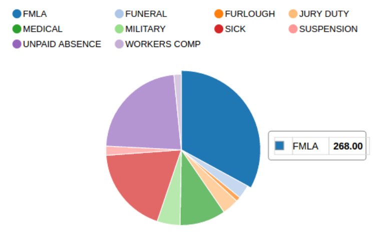
Available settings

- **url:** url to fetch the options base on the keyword you typed in the input.
- **multi:** you can enable multi-value selection by setting multi to true.
- **name:** an arbitrary name.
- **options:** an array with options (e.g.: [{ value: 'one', label: 'One' }])

Base Component

Chart Component





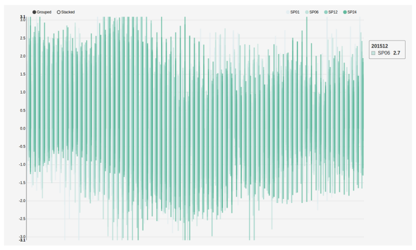


Chart component is a wrapper of the *react-nvd3* library, which is also a wrapper of the *nvd3* chart library. That meanas all the charts and options available in nvd3 are also available in this component.

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

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

id, type, fetchData and height are mandatory.

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.

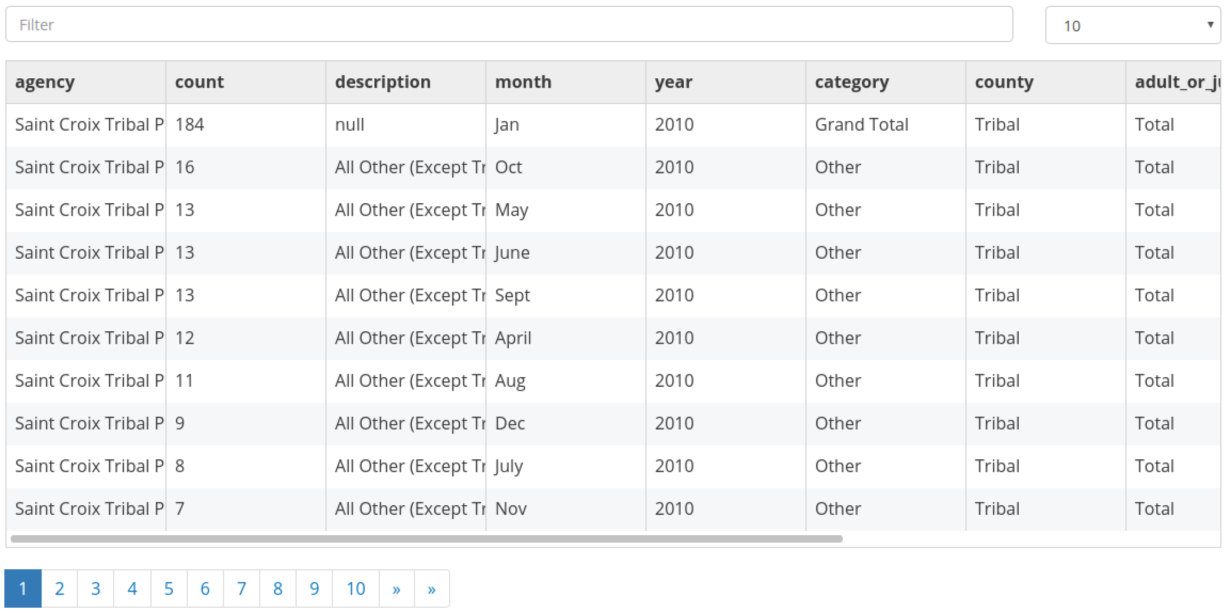
Available settings

- **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 requriments which vary based on chart type. See the NVD3 documentation and examples which oultine 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

Data Table

DataTable component provides a way to browse, filter, search and display datasets to end-users.



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

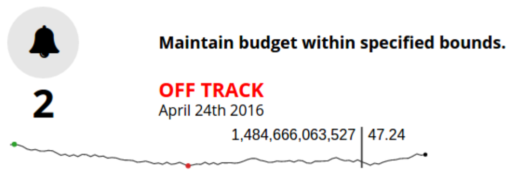
```
{
  type: 'GATable',
  header: 'Mi titulo',
  fetchData: {
    type:'backend',
    backend: 'csv',
    url: 'http://demo.getdkan.com/node/9/download',
  },
  settings: {
    table: {
      rowHeight: 40,
      width: 800,
      maxHeight: 300,
      headerHeight:40
    },
    columns: {
      flexGrow: 1,
      width: 150,
      overrides: {
        a1: {
          flexGrow: 0.5
        }
      }
    },
    cells: {
      height: 40,
      width: 500,
      overrides: {
        1: {
          height: 40
        }
      }
    }
  }
},
```

Available settings

- **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 messure 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. If you create a completely new component (it inherits either from *Component* or *BaseComponent*) then you can use it in this way:

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

      </Loader>
    );
  }
}
```

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

If you are extending from the *BaseComponent* and using the *fetchData* property to fetch resources then the *isFeching* state is handled for you.

If you aren't using *fetchData* to fetch resources then you need to switch this variable manually.

Markup

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

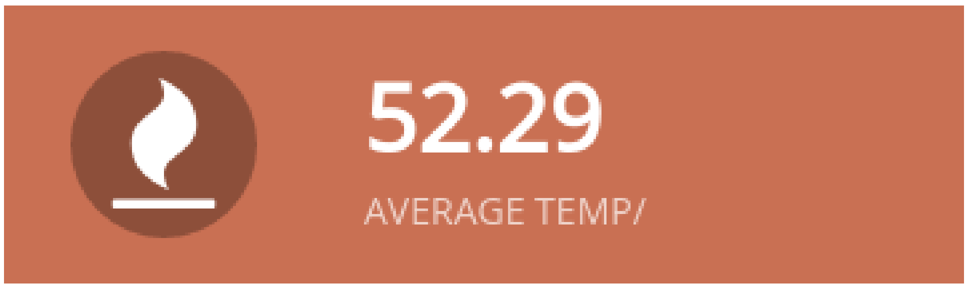
For example - a static html list:

```
{
  type: 'Markup',
  content: '<div>\
              <ul>\
                  <li>FOO</li>\
                  <li>BAR</li>\
                  <li>BAZ</li>\
              </ul>\
          </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
- **fetchData:** fetch datat callback
- **dataHandlers:** an array containing dataHandler object(s)
- **options:** an array with options (e.g.: [{ value: 'one', label: 'One' }])

Multi Component

The **Multi Component** provides a starting point for developing component rendering schemes that depend on logic to determine which components to render. The Multi component expects the following settings:

- **elements** a keyed array that defines a set of elements. of the format:

```
    elements:
        a: [
            {//... a component},
            {//... another component},
            {//... etc}
        ],
        b: [
            {//... just one component} // still use an array to define a single␣
↪component
        ]
```

Child components should contain a 'key' value which is unique, and allows React to keep track of lists of children.

- **initialSelect** the key value to load as the initial set of elements (for example, given the above elements array 'a')

In addition to these settings, the implementation of the Multi component should define the following methods:

render This will render the component. The render method can define a UX element to control this method can call `this.renderChildren()` in order to render the children **multiComponentListener** This method is responsible for listening for a trigger to update the multicomponent. This can be an onChange handler that is defined on an input element in the render function, a global even which is triggered by an action, or an as-yet-unforseen method of updating the app-state. The only rule is the the multiComponentListener method needs to be reliably triggered, somehow, and it needs to set the state.elements array to the an array of valid dashboard components. **componentWillMount** By default, this function will set the initial state.elements array to the value assigned to *initialSelect* in the components settings. This, however, can be overridden to provide custom logic to determine the initial state of the multi component. **NOTE** This sounds more confing than it is. Look at the `/examples/GAMultiSelectComponent.js` and `src/compenents/Multi.js` source code to understand more clearly what is going. **NOTE2** Stay tuned for more out of the box functionality and better documentation!

Creating custom components

Extending components

Components can be extended to provide custom behavior:

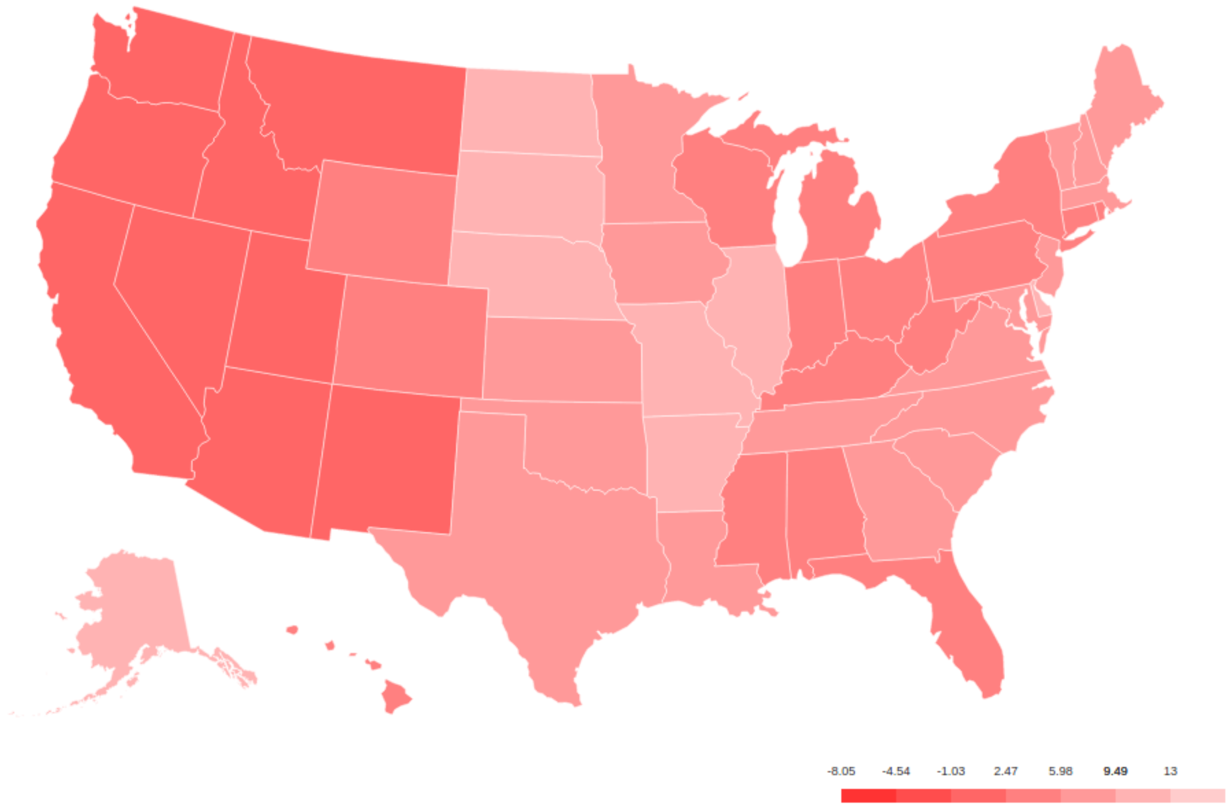
```
import React, { Component } from 'react';
import Registry from '../../src/utils/Registry';
import Chart from '../../src/components/Chart';

export default class CustomChart extends Chart {
  // ... do custom stuff here
}

// make sure to register the component!!
Registry.set('GAChart', CustomChart);
```

Javascript alllows you to override any method of a parent class, but...

** Functions commonly overridden in custom components: **

- **fetchData:** Provide logic for gatherin data
- **onData:** Preprocess the fetched data, when available
- **onResize:** Add a post-hook to a resize event. (this.state.componentWidth should always be available, and is updated after resize, but before onResize is called)

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: 'equirectangular',
  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

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

1.2.10 Theming

Bootstrap grid

React Dash uses bootstrap responsive grid. Full docs are here

Define rows and columns as follows in your settings.js file:

```
{
  type: 'Region',
  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'
    }
  ]
}
```

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!',
  // .... your settings follow
}
```

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'}
}
```

1.3 Implementation Examples

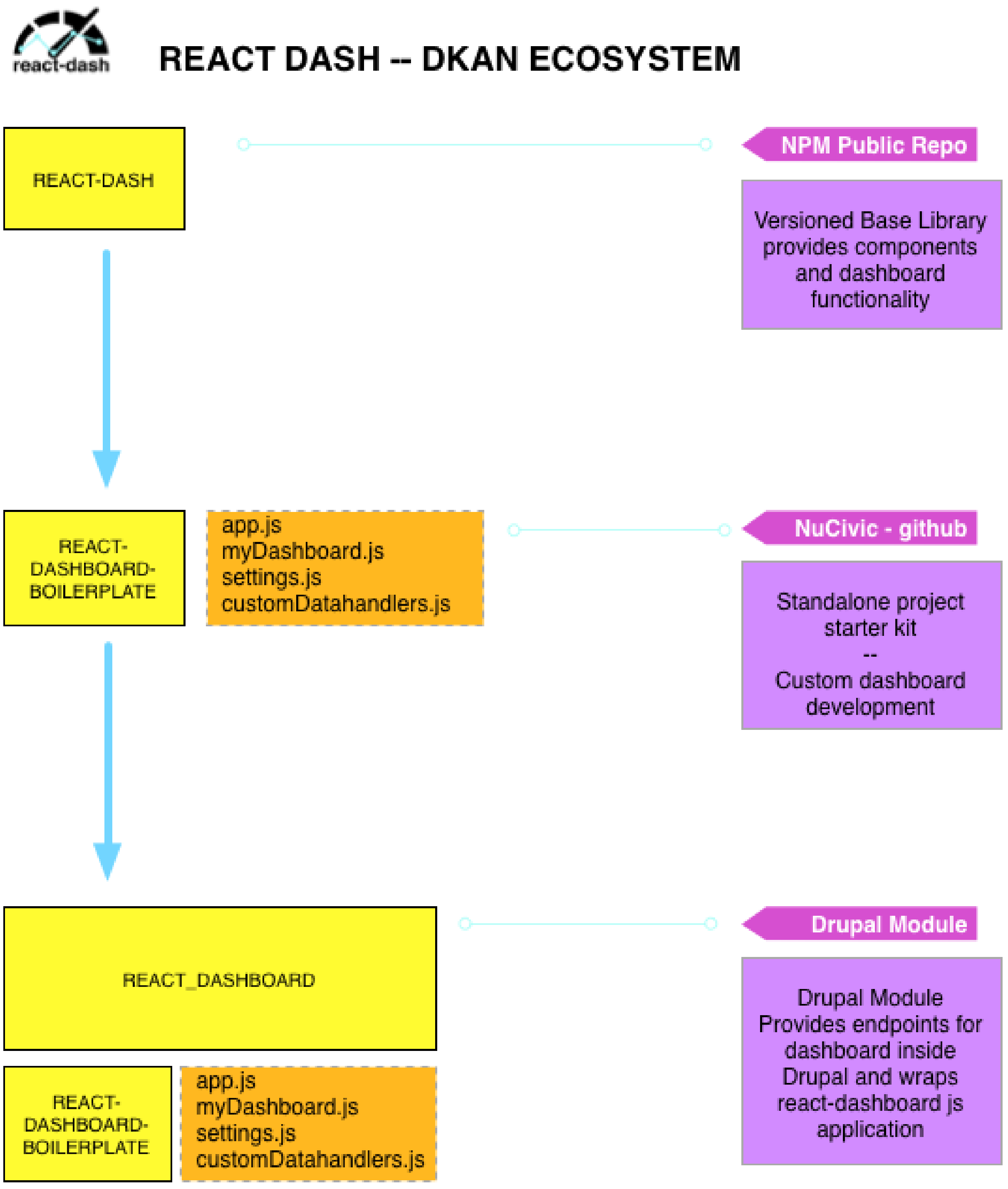
- Hours Worked Performance Dashboard
- Current Library Example
- UCR Arrest Data Dashboard

1.4 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
REACT DASH -- DKAN ECOSYSTEM
REACT-DASH
NPM Public Repo
Versioned Base Library provides components and dashboard functionality
REACT-DASHBOARD-BOILERPLATE
app.js
myDashboard.js
settings.js
customDatahandlers.js
NuCivic - github
Standalone project starter kit -- Custom dashboard development
REACT_DASHBOARD
REACT-DASHBOARD-BOILERPLATE
app.js
myDashboard.js
settings.js
customDatahandlers.js
Drupal Module
Drupal Module Provides endpoints for dashboard inside Drupal and wraps react-dashboard js application

1.5 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!