JView, an reactive and agile visualization tool kit for JData

Introduction

JView is a generic visualization library that utilizes React.js, Mobx.js, Node.js and Socket.io to generate dynamic and synchronized front end view of JData. The main advantage of using this technology stack is that it utilizes the concept of one-way data flow (so-called one-way binding) and component hierarchy of React framework so that the view is synchronized with any JData flow and the user can dynamically change the view structure by reorganizing the components with standard APIs.
The overview of the stack works as follows:

- Node.js, which in this case a JNode service, is the main controller that sets up socket.io and hosts view assets after Webpack-ing React component. It provides a main entry point to the Jview service.
- socket.io is hosted by Node.js when the Node service starts to run. It is a wrap up of Websocket APIs of the TCP socket. It will act as a messaging hub for JData in-flow and JView out-flow.
- Mobx.js is the data layer of all non-stateless React components. It tells when and what the react components should render.
- React.js is the main JView layer that communicates with the client. It includes two important libraries, JView Components and ECharts Adaptor. JView Components include all the default React components written in ES6 that can be reused anytime, including basic Form and Text input to Calendar Selector. The style sheets are included for all the components as well using LESS, so the colour scheme and layout can be changed dynamically based on client preference and demand as well. ECharts Adaptor is a wrap up library of ECharts 2.0, an open source library built by Baidu.inc, to match with JView APIs.

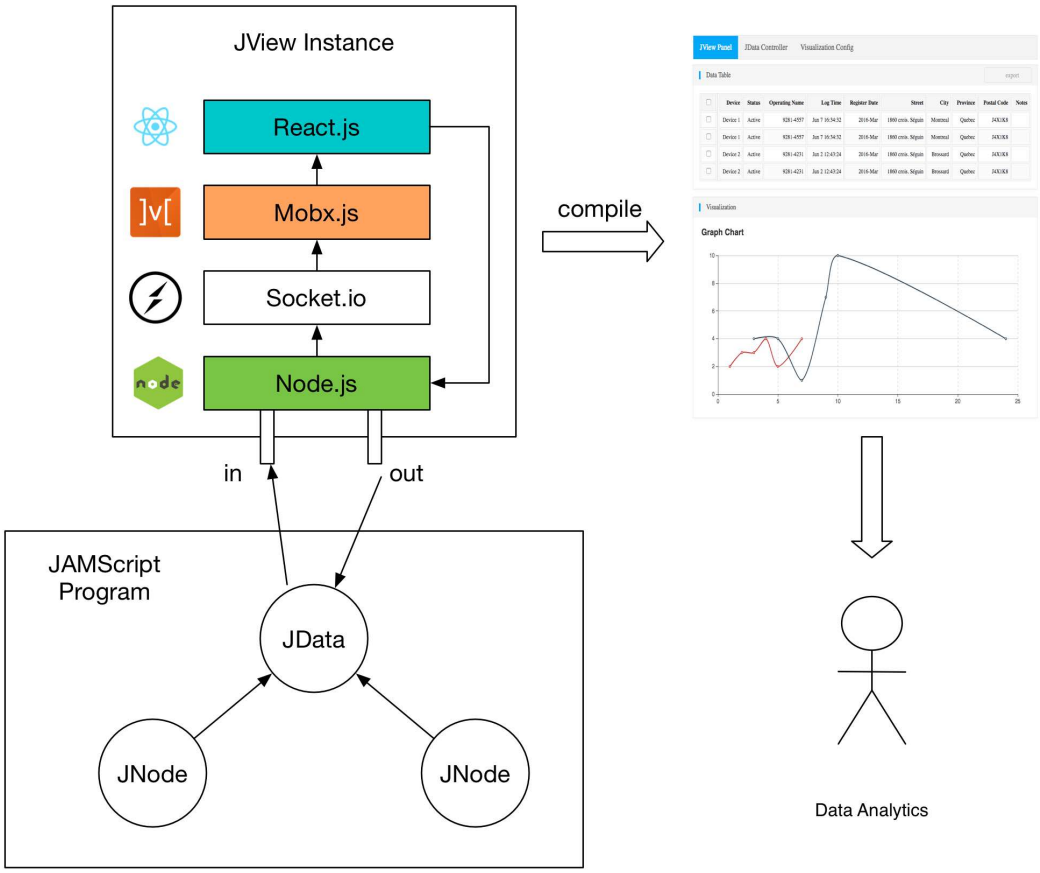
Why JView?

Nowadays, the pace of front end development community has been really fast, for popular JS packages like Webpack or Angular, the average number of monthly releases is around 3. A lot of engineers have the headache choosing the right technology stack from a variety of choices, keeping themselves technologically secure over the newest dependency versions, and maintaining the legacy code. The rationale for having JView integrated with JAMScript is to give JAMScript developer a professional starter kit for visualization and client-side front end that has a certain amount of bootstrap components and wirings to avoid users writing repetitive boilerplate and component code. It is inspired by the old days PHP framework CakePHP whereas the Bake command will interact with the client to get the requirements, and generates code for the whole application just like a micro-Artificial Intelligence. It abstracts and explores in how human engineers think and code, so later on it will save human engineers the time for repetitive work just to have a whole project set up and running, and free their mind for more creative works.

How it works

JData, which is the data store for JAMSCript Programs, is mainly based on pipelining of flows. JView follows with the schema of JData, and it works based on a flow as well. Once the client knows what they want to see from JData, they just need to compile JView with an JSON file defining all the configurations, and a whole visualization application is automatically generated and ready to go.

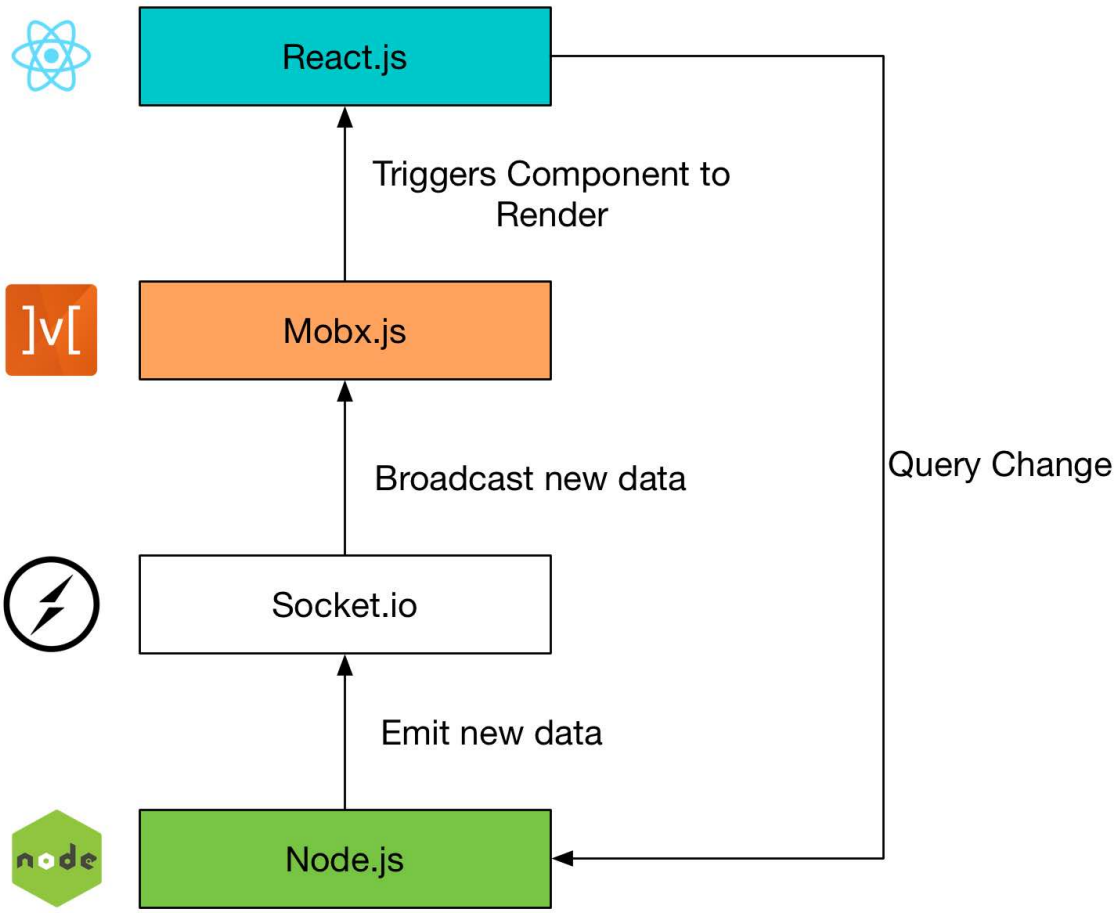
Overall Concept of JView:



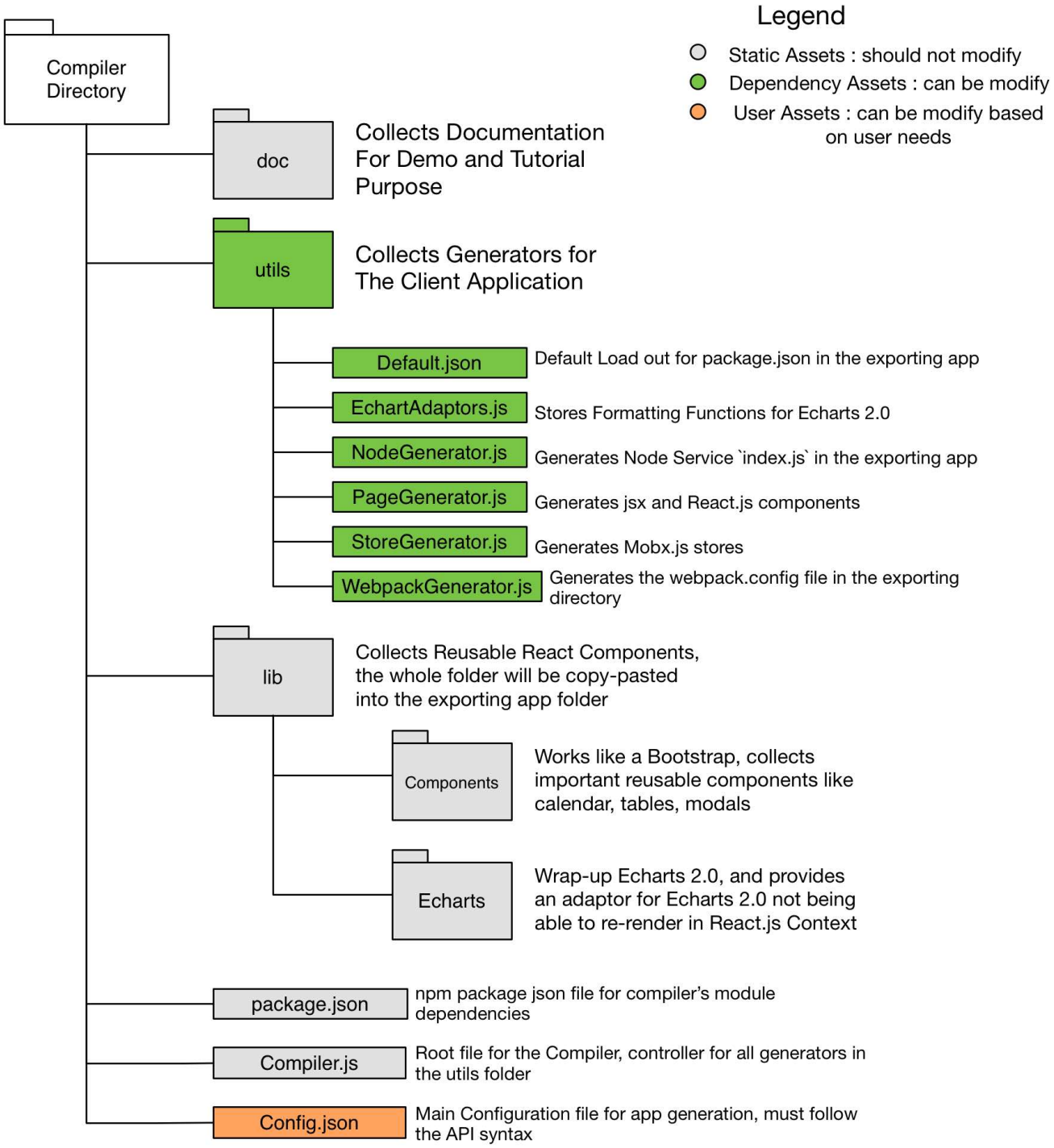
Emergence of React.js has provided a lot of companies, including Baidu and Tableau, a elegant solution to develop huge single page applications. Just like the three main bulletins on React's official documentation, it is Declarative, Component-based, and 'Learn Once Write Anywhere'. A React app basically maintains a hierarchy of components. The encapsulation of each component made it possible to work independently anytime anywhere, and the way how to import a certain component is the same across all containers (a container in React is a parental component that contains a group of child components, just like a Composite pattern in OOP). This enables unification for how a engineer would compose a view or container, and hence it can be abstracted and compressed into a generator. Similar concept can be used to

create generators for other layers of the application as well — if the way how an application is being developed follows closely on a certain coding style, then abstract that coding style into a generator that utilizes encapsulation and all that good stuff.

Technology Stack



Package Directory



API Documentations

The following APIs are for the json file that the Compiler will take as an input.

Root Level:

First depth of the JSON config file.

path (String) - a path you want to generate your JView Application to
pages ('page' JSON Object []) - a array of pages you need to generate, each JSON object inside of the array has to be in Page Config format <See documentation of 'pages' below>
port (Number) - the port a client wants to host the application on
package ('package' JSON Object)- inside of an JS application there will be a package.json maintaining the package information and dependencies. The compiler injects a default package.json file but the user can add additional configs into the file from here <See documentation of 'package' below>

'pages' (JSON Object []):

An array of 'page' JSON object for configurations of all the pages that the client needs to generate.

'page' (JSON Object):

Required Properties for each of these 'pages':
name (String) - name of the page, will be shown of the Nav Menu bar
route (String) - page directory url, will be shown on the browser history
layout (String) - name of the standard layout for this page
panels ('panel' JSON Object []) - on each page, there will be multiple panels, 'panels' is the property that maps to a JSON Object array for panel definition.

'package' (JSON Object):

A JSON object for configurations of all the information that is going to be displayed in 'package.json' file.

Required Properties 'package':
name (String) - name of the project
version (Number) - version number of the project
description (String) - description of the project
author (String) - author name of the project

'panels' (JSON Object[]):

An array of 'panel' JSON object for configurations of all the pages that the client needs to generate.

'panel' (JSON Object):

Required Properties for each of these 'panels':
type (String) - name of the type, in 'FormSet', 'Table', 'Scatter', 'Pie', 'Graph'
name (String) - name/tag of the panel

store (JSON or String [] or 'socket' JSON object) - declare the Mobx.js store for this panel , type of 'store' will vary depends on the type of Panel is selected
**formList (JSON object [])* - declare the form components to show on the form when type of the panel has been chosen to be 'FormSet'
**actionList (JSON object [])* - declare the action buttons for the form when type of the panel has been chosen to be 'FormSet'
**headers (JSON object [])* - declare the table headers (hence, the store must follow the header schema) when type of the panel has been chosen to be 'Table'
**headers (JSON object [])* - declare the table headers (hence, the store must follow the header schema) when type of the panel has been chosen to be 'Table'

Sample and Tutorials:

All the sample and tutorials are included in the doc/Samples directory, to compile the samples, go to compiler root directory, run :

```
node Compiler.js doc/Samples/1.Basic-Formset/Config.json
```

on the command line after installing Node. To run Samples, when you are in the compiler root directory, type :

```
node app/index.js
```

on the command line