JSCatalyst

A UI Developer’s Guide

Written by: Eikos Partners

Version 1.0

Last Update: 3/19/18

Introduction 2

Overview of Application 2

Dependencies 2

Code Structure 3

Common Folder 3

Assets Folder 3

Components Folder 3

Pages Folder 3

initialState.js 3

store.js 3

package.json 3

Layout and Operation 4

State Management 5

Webpack Configuration 5

How to Make an API Call 5

How to Create a Non-D3 Component 7

Tabs 9

Dropdowns 10

Layout Helpers 10

Other Components 10

Charting Libraries 10

ChartJs 10

TauCharts 11

How to Create a D3 Component 11

Details For Each D3 Component 14

How to Create A Data Table 18

Server Table 18

Client Table 20

How to Create A Blotter/Data Grid 20

How to Open JSCatalyst in OpenFin 22

How to Create A New Dashboard 23

<template> 23

<script> 24

Data Properties 24

Lifecycle Hooks 25

Instance Methods 25

<style> 26

Vue-Specific Tips 27

Testing 27

This page is intentionally left blank 29

# Introduction

JSCatalyst is an Open Source Platform to support development and lifecycle management of high impact full-stack JavaScript applications in a CI/CD world. The platform accelerates application development and provides both designers and developers with tools to declaratively build user interfaces that function across all types of devices and browsers.

The purpose of this guide is to explain the layout of the application, provide an overview of dependencies and operations, and walk through how to implement visualizations and dashboards. In particular, we’ll cover:

* An overview of the application and basic architecture
* How to make an API call to get data
* How to create a D3 component
* How to create a non-D3 component (menus, lists, tabs, etc.)
* How to create a data table
* How to create a new dashboard
* Testing

Please refer to the ‘Example’ dashboard which provides a simple structure for a dashboard including layout, data visualizations, and tables.

# Overview of Application

## Dependencies

The UI is built in [Vue.js](https://vuejs.org/), uses [Vuetify](https://vuetifyjs.com/) as a component/CSS framework, as well as [CSS Grid](https://www.w3schools.com/css/css_grid.asp) for layout. Other notable dependencies include:

* [D3 library](https://d3js.org/)
* [Axios](https://github.com/axios/axios) - promise-based HTTP client
* [Ag-Grid](https://www.ag-grid.com/) – HTML5 JavaScript datagrid
* [Vue-resize](https://github.com/Akryum/vue-resize) - detects DOM element resizing and makes charts/graphics responsive
* [Vuex](https://github.com/vuejs/vuex) - centralized state management
* [WindowManager](https://github.com/EikosPartners/windowmanagerjs) - framework developed in-house at Eikos; manages multiple HTML windows

## Code Structure

### Common Folder

Calls to the API are found in the **src » common** folder. Helper files for datatable calls are also found in this folder.

### Assets Folder

CSS stylesheets, static images, and other assets are found in **src » assets**.

### Components Folder

Basic components are contained in **src » components**. Components include items that are reusable (not dashboard-specific) including visualizations, modals, tables, toolbars, and the footer. Universal components are components found on each page - panel headings, the toolbar, etc. Visualizations (charts, graphics, etc.) are also found in the components folder.

### Pages Folder

Each dashboard has its own folder under **src » pages**. Each dashboard’s folder contains its necessary App.vue, app.js, and app.html files as well as any components that are specific to that dashboard. The Index page (listing all accounts and their associated dashboards) is located in **src » pages » index**.

### initialState.js

Defines the initial state of the application.

### store.js

Sets up Vuex, a centralized store for the application.

### package.json

Holds pertinent metadata and dependencies relevant to the project.

Best practices include keeping components as reusable as possible so they can be dropped in to a dashboard or parent component and then passed the relevant parameters as Vue props.

We have structured the application using the [Single File Component](https://vuejs.org/v2/guide/single-file-components.html) pattern. All HTML, CSS, and logic for any .vue file, whether it be a simple footer or a complex dashboard, is contained within the file in the <template>, <script>, and <style> tags.

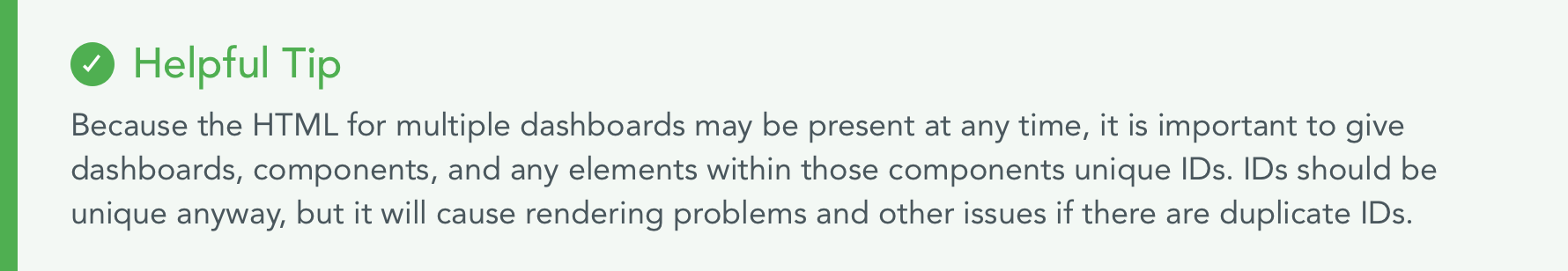
# Layout and Operation

**src » pages » dashboards » dashboards » App.vue** serves as a container for all dashboards. Any dashboard that is opened in the same session will be present in the source HTML, although only the selected dashboard is rendered on the screen (think of each open dashboard as laying on top of the others). This means that each dashboard does not need to be reloaded each time it is opened, making the application fast and performant, and allows the user to easily toggle between dashboards.

This functionality is made possible by WindowManager, which renders each dashboard’s app.html file and places it in the correct location in **dashboards » dashboards » App.vue**. Each app.html file references the unique id of the dashboard, which contains the HTML template, scripts, and styling for that dashboard. Vue takes over from there, executing the scripts that create the virtual DOM.



The app.html page for Alerts dashboard.



The user will always see dashboards/dashboards in the URL bar. However, the account and dashboard are still being passed in as parameters and are stored in sessionStorage.

## State Management

We use Vuex to store the current theming of the application, the current theming of the charts, the colors used for the charts, and all possible color themes.

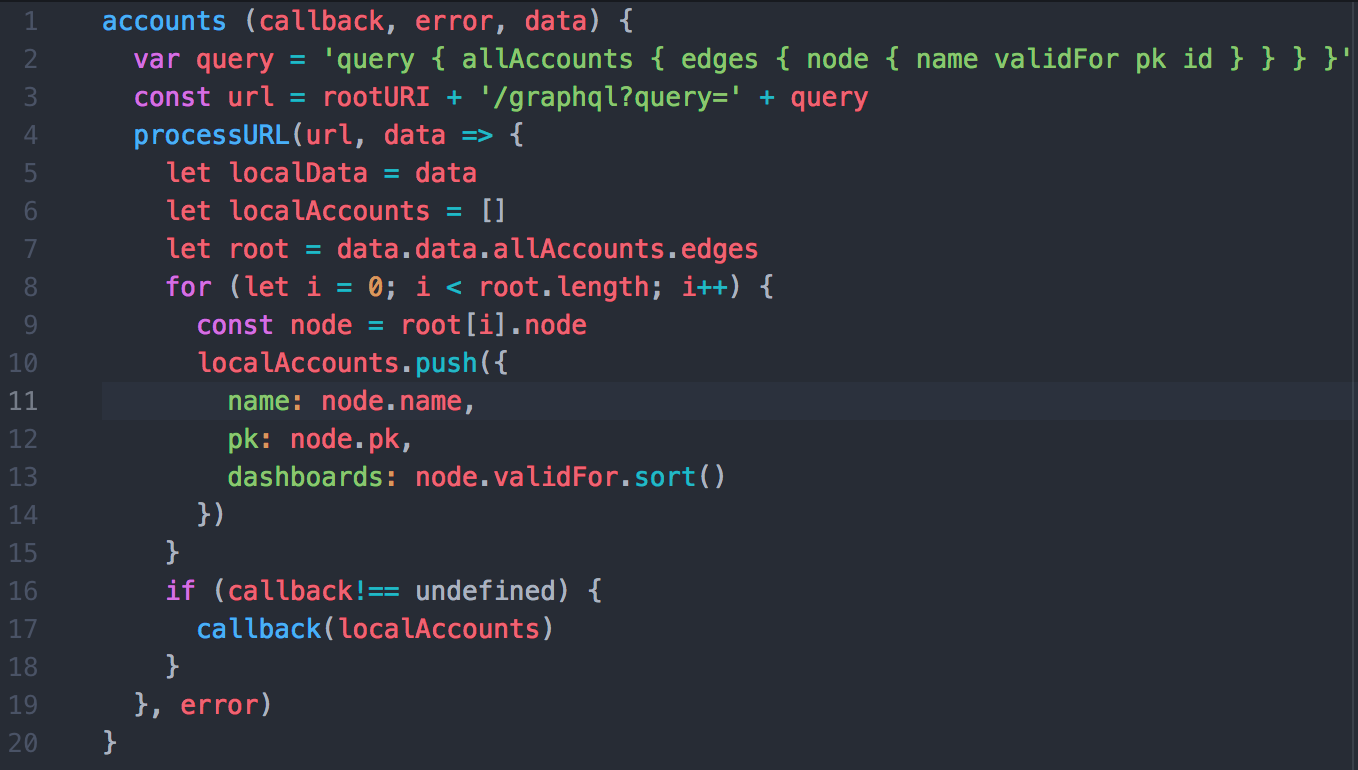
## Webpack Configuration

Webpack splits each dashboard into its own mini-bundle so only the selected/opened dashboards are loaded and rendered.

Webpack also bundles all CSS into venders.css stylesheet. You can see where a particular style is defined using the browser’s dev tools.

# How to Make an API Call

Backend services include both REST and [GraphQL](http://graphql.org/) endpoints. The application currently uses GraphQL endpoints for all accounts; the rest of the API calls are to REST endpoints. Here is the query for accounts:



Calls to the API are made on initial load, when a new dashboard is opened, when a property changes, or when a filter is applied. API request methods are located in the **src » common** folder. API calls can go in any of the [Vue lifecycle methods](https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram), but created is preferred. Pages don’t get reloaded for each API call; Axios makes calls to fetch data on request and the DOM updates to reflect the new information.

To construct a new API call, go to restAPI.js and use the following format:



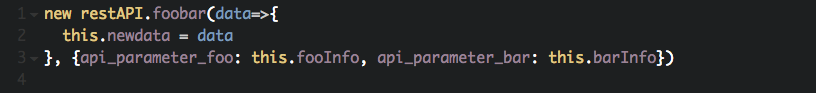
* rootURI is the server URL that will be used for the request (found in baseRest.js).
* Params are the parameters necessary to make the request.
* The processURL function (also located in baseRest.js) takes in the url and a callback function\*, then applies the API response as an argument to the callback function.

Here is the processURL function. You shouldn’t have to change this, but it’s helpful to be aware of how it works:



The third parameter, error, is optional. It is another callback function that will be executed on the response if the API call returns an error. If this parameter is omitted, processURL will just console.log the error message.

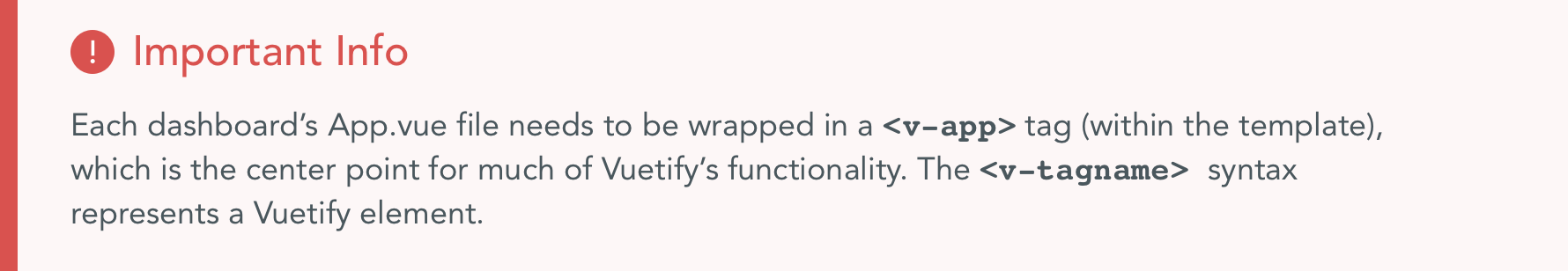
* \* A quick note about callbacks - the callback function is the function that will be executed upon return of the API response. Since JavaScript functions are first class objects, you can pass functions to other functions as variables.
* To utilize an existing API call in a dashboard, use the following format:



* In the callback of the API request, you can assign the response of the API call to a local data property, e.g., this.newdata = data.
* Declare callback functions in their own dashboard/component so as to keep the API calls as reusable as possible, and the callbacks as customizable as possible.
* Data table requests must be customized from table to table. Some helper functions for formatting these requests and responses are found in helpers.js and tableDataParser.js.

# How to Create a Non-D3 Component

For our purposes, non-D3 components consist of dropdowns, tabs, buttons, and other items that are part of the layout but not data visualizations. The UI uses [Vuetify](https://vuetifyjs.com/), a component/CSS framework, to generate these items. Vuetify has extensive documentation, is open-source, and is actively in development. It also works seamlessly with Vue so creating these components is relatively straightforward.



You have to register the component so that it is recognized by the application. The component should be registered in export function of the App.vue of the dashboard where the component resides.

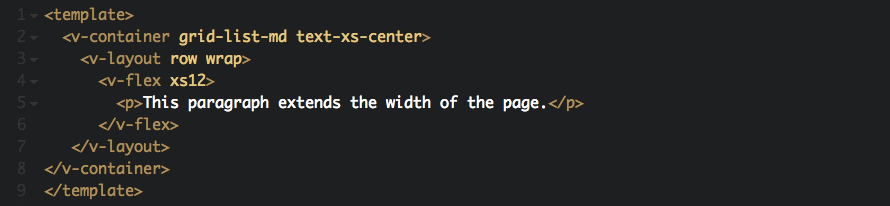


Like bootstrap and other popular CSS frameworks, Vuetify uses a [12-point grid system](https://vuetifyjs.com/layout/grid) for layout. The structure of the layout will be as follows: **v-container » v-layout » v-flex**.

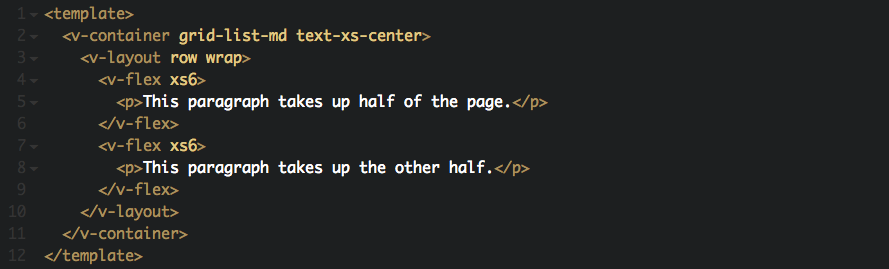
* <v-container> is used for a center focused page, or given the fluid prop to extend its full width.
* <v-layout> is used for separating sections.
* <v-flex> contains your content.

From there, you can specify how many column widths you would like your content to fill.

A full-width column would look like this:



Two columns would look like this:



And so on. Offsets, order, alignments, and nested grids can also control the flow of the page. See the [Vuetify documentation](https://vuetifyjs.com/layout/grid) to read more about these features.

The grid system makes use of the [Material Design Viewport Breakpoints](https://vuetifyjs.com/layout/grid), where you can specify how you want your content to look on different screen sizes. For example, content in <v-flex xs12 s12 md6 lg6 xl6> will take up the full width on small screens and half the width on larger screens.

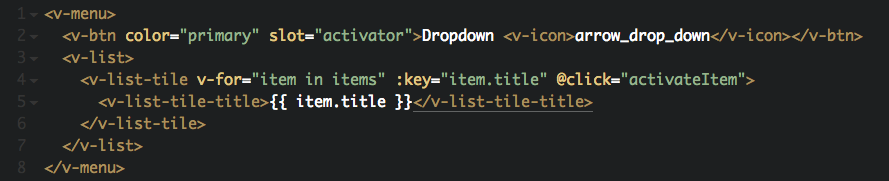
## Tabs

Vuetify offers a customizable [tabs component](https://vuetifyjs.com/components/tabs). You can set the active tab by using the v-model prop. The [v-model directive](https://vuejs.org/v2/guide/forms.html) creates a two-way binding between the HTML and the data property defined in the <script> tag. You can control the content with tags such as <v-tabs-bar> (the tab toolbar), <v-tabs-item> (the individual tab links), and <v-tabs-content> (the tabs themselves).



## Dropdowns

Vuetify uses [dropdown components](https://vuetifyjs.com/components/menus) via the <v-menu> or <v-select> tags. When using <v-menu>, you must specify an activator slot that will trigger the dropdown.



## Layout Helpers

Vuetify provides handy [layout helpers](https://vuetifyjs.com/layout/spacing) to set margin and padding to specified amounts without creating new classes. Use ‘m’ or ‘p’ to set the margin or padding as well as the direction. For example:

* <p class=“mt-1">...</p> applies a margin top of 1 (16px \* .25)
* <li class=“pb-2">...</li> applies a padding bottom of 2 (16px \* .50)
* <div class=“pa-2">...</div> applies a padding of 2 in all directions

## Other Components

Vuetify offers numerous other components you can insert into the application including buttons, alerts, sliders, and icons, among others. We’ve detailed a few for you here, but [see the documentation](https://vuetifyjs.com/vuetify/quick-start) for information on all the available components.

# Charting Libraries

There are many different options for JavaScript charting libraries available to use. Each one has a different approach to rendering visualizations whether it be through the use of Canvas or SVG. This section will provide a quick overview of charting libraries that can be used with JSCatalyst.

## Chart.js

Chart.js is a widely used and well-established library for creating data visualizations based on the HTML5 Canvas. It provides around 8 basic charts that are easily customizable. It also has a Vue wrapper provided by vue-chartjs which makes it easy to create reusable charts. The charts can be updated dynamically to allow for streaming data in two ways.

The first is by using a mixin provided by vue-chartjs which creates a prop chartData and adds a watcher. To update the data, the container the chart is in must take the old data object and combine it with the new data using Object.assign() and reassign the data value to the new object. This will trigger a an update in the chart.

The second option is to create an update method on the chart which calls the chart instance’s update function. Then in the container you must create a ref to the chart. When new data is received by the container the data must be pushed on to the appropriate arrays in the data object and the container must trigger the chart’s update method through the ref.

Despite the easy implementation of Chart.js, the charts that you are able to create are almost exclusively limited to the base charts provided. All of our Chart.js components are contained within wrappers in order to make them easy for developers and designers to implement.

## TauCharts

Taucharts is charting library based on D3 that is focused on providing flexibility while keeping design in mind. It is very straightforward when it comes to creating charts. There is no formatting necessary for the data, just pass it an array of data and tell it which keys should be plotted on the x-axis and y-axis. By providing different options it is possible to customize axis, colors, and design. Another unique option that is included in this library are the facets. Facets allow for multiple frames, each with a chart, to be drawn within one main chart. This allows for easy comparison of information with many variables.

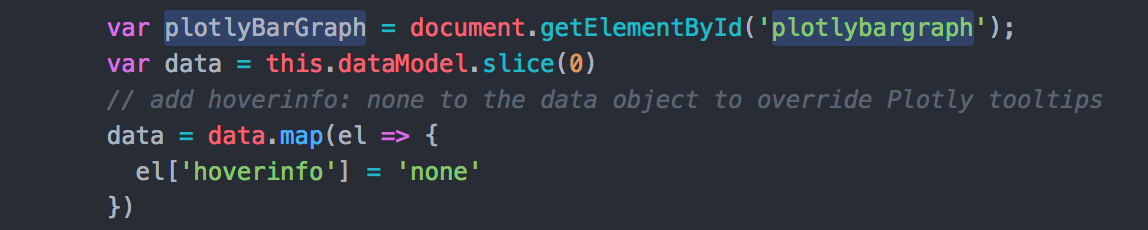
While TauCharts is very easy to use, it does not currently have a version that is built with Vue in mind. Because of this it is hard to update the data of charts that could be used for streaming data. Currently the only way to add data to a chart is by calling the addData() function on the chart instance. This makes it hard to for the container of the chart to pass new data to the chart.

## Plotly

[Plotly.js](https://plot.ly/javascript/) is built on top of d3.js and stack.gl and is a declarative charting library.

Tooltips – to create a custom tooltip in JS Catalyst style, you must completely override Plotly tooltips and create your own.

To override the tooltips, add the following code, which will map over the data model and an additional ‘hoverinfo’ key with a value of ‘none.’

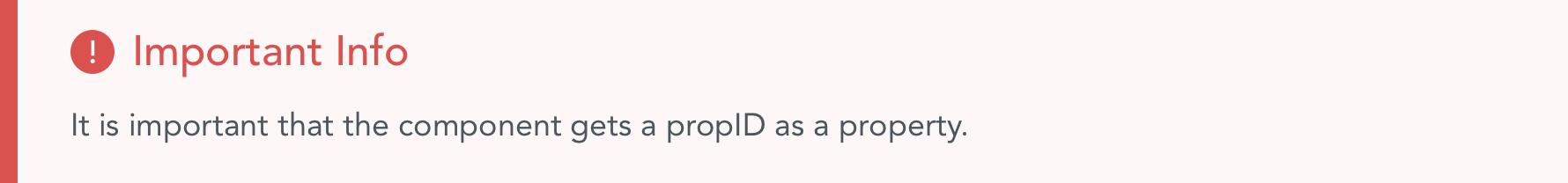


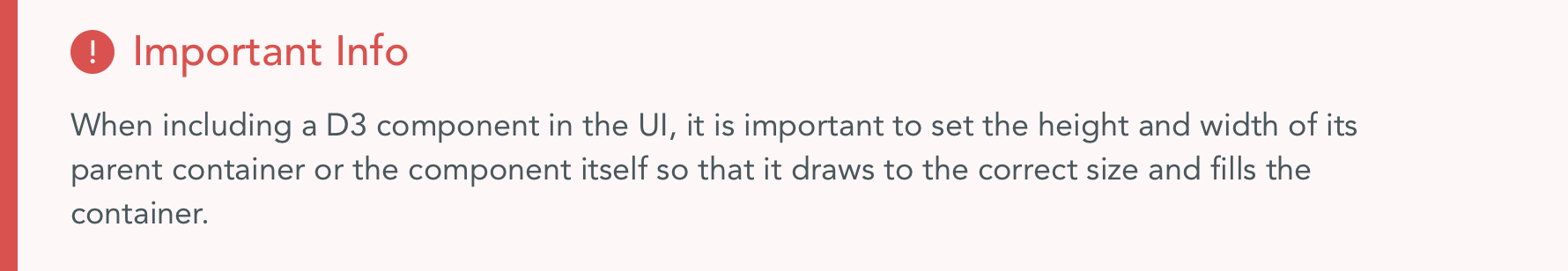
Then, we tap into the plotly\_hover event to add the tooltip and give it the proper placement.



# How to Create a D3 Component

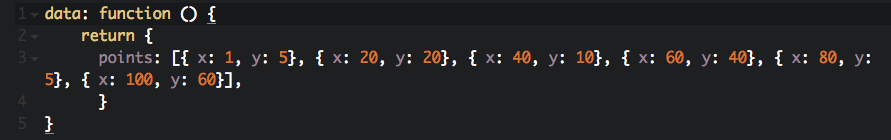
This section will cover how to integrate a D3 component (data visualization) into a dashboard. For a simple example, see SvgComponent.vue in the Example dashboard. Note again that components need to be registered in the App.vue file of the dashboard where they reside.



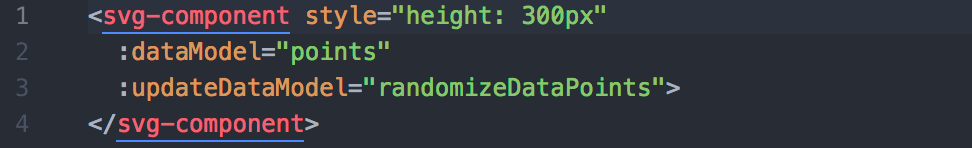


Data that will be used in the visualization should be defined and updated then bound as a prop of the D3 component in the template of the parent (usually App.vue).

Data is defined:



Data is bound in the HTML:



Note that the colon syntax (:dataModel) is a [data-bind in Vue](https://v1.vuejs.org/guide/syntax.html). One of Vue’s niftiest features is its reactive data binding. Vue will reactively update the DOM when a variable in the data is changed without you having to do anything (in the example above, if we make a new element in the points array, the HTML will automatically reflect the addition). Two-way data binding, which binds the HTML back to the JavaScript, is done by providing some kind of input and using the v-model directive.



[This blog post](https://medium.com/js-dojo/exploring-vue-js-reactive-two-way-data-binding-da533d0c4554) provides a simple breakdown of data binding in Vue.

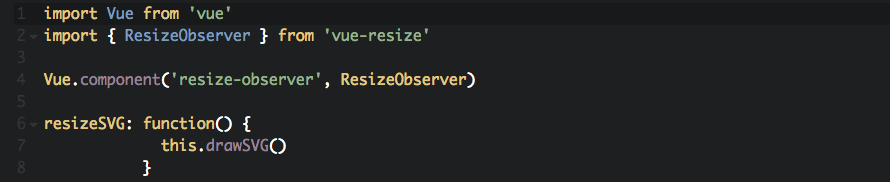
Each D3 component will need a draw function that draws the visualization. The draw function should be called in three places: mounted, for the initial draw; on watch of the dataModel, to draw the SVG when filters are applied or another event changes the data; and in a resize SVG function.

The resize SVG function is used alongside the { Resize Observer } from vue-resize and will redraw the component when the window is resized, making the component responsive.

In <template> of the D3 component:

/Users/sylveelee/Desktop/Screen Shot 2017-12-12 at 2.43.30 PM.png

In <script>:





# Details For Each D3 Component

Bar Chart

Necessary props:

* dataModel (Array)
* propID (String)
* xaxisLabel (String)
* yaxisLabel(String)

Special Styling:

Developer Notes:

Bubble Chart

Bullet Chart

Necessary props:

* dataModel (Array)
  + Necessary keys: title, subtitle, ranges, measures, markers
* propID (String)

Special Styling:

Developer Notes:

* Measures is an array of one or two values. When it contains two values the bar will be two different colors and can be used to show expected and actual values.

Difference Chart

Necessary props:

* dataModel (Array)
  + Necessary keys: date, expected, actual
* propID (String)
* metric (String)
* dateFormat (String) – must be a Momentjs format string that matches the format of the dataModel date values ex.(‘MM-DD-YYYY’ or ‘MM/YY’)

Special Styling:

Developer Notes:

* The dateFormat is used to get the correct format of the D3 axis. Because D3 uses it’s own formatting system the formatTimeMixin is used to map the Momentjs format string to the correct corresponding D3 format string (ex. Momentjs ‘MM-DD-YY’ => D3 ‘%m-%d-%y’)

Gantt Chart

Necessary props:

* dataModel (Array)
  + Necessary keys: startDate, endDate, taskName, status
* propID (String)
* taskStatuses (Array)
* tickFormat(String)

Special Styling:

* The taskStatuses Object is responsible for styling the rectangles in the Gantt chart based on whether the task failed, is running, was killed, or succeeded. By changing the keys and values the styling can be customized

Developer Notes:

Heat Map

Line Chart

Necessary props:

* datamodel (Array)
  + Necessary keys: date, value
* propID (String)
* alertText (String)
* dateFormat (String) – must be a Momentjs format string that matches the format of the dataModel date values ex.(‘MM-DD-YYYY’ or ‘MM/YY’)

Special Styling:

Developer Notes:

* The dateFormat is used to get the correct format of the D3 axis. Because D3 uses it’s own formatting system the formatTimeMixin is used to map the Momentjs format string to the correct corresponding D3 format string (ex. Momentjs ‘MM-DD-YY’ => D3 ‘%m-%d-%y’)

Line Plot

Necessary props:

* datamodel (Array)
  + Necessary keys: date, value
* propID (String)
* metric (String)
* dateFormat (String) – must be a Momentjs format string that matches the format of the dataModel date values ex.(‘MM-DD-YYYY’ or ‘MM/YY’)

Special Styling:

Developer Notes:

* The dateFormat is used to get the correct format of the D3 axis. Because D3 uses it’s own formatting system the formatTimeMixin is used to map the Momentjs format string to the correct corresponding D3 format string (ex. Momentjs ‘MM-DD-YY’ => D3 ‘%m-%d-%y’

Pie Chart

Necessary props:

* dataModel (Array)
  + Necessary keys: label, value
* propID (String)
* pieColors (Array)
* units (String)
* donutWidth (Number)

Special Styling:

* pieColors is an array of hexadecimal color strings that are used to style the pie slices. Default colors are used if this prop is not passed

Developer Notes:

* If donutWidth is passed as prop it will turn the pie chart into a donut chart. The number indicates the pixel radius of the inner white circle

Punch Card

Scatterplot

Necessary props:

* dataModel (Array)
* propID (String)

Developer Notes:

* You must leave room in the right margin for the legend to render

Stacked Bar Chart

Necessary props:

* dataModel (Array)
* propID (String)
* xaxisLabel (String)
* yaxisLabel(String)

Developer Notes:

* You must leave room in the right margin for the legend to render
* The dataIntermediate function transforms the data into one array for each category (so the bar is ‘stacked’).

Sunburst

Two Line Plot

Necessary props:

* dataModel (Array)
* dataModel2 (Array)
* propID (String)
* metric (String)
* metric2 (String)
* startDate (String)
* dateFormat (String) – must be a Momentjs format string that matches the format of the dataModel date values ex.(‘MM-DD-YYYY’ or ‘MM/YY’)

Special Styling:

Developer Notes:

* The dateFormat is used to get the correct format of the D3 axis. Because D3 uses it’s own formatting system the formatTimeMixin is used to map the Momentjs format string to the correct corresponding D3 format string (ex. Momentjs ‘MM-DD-YY’ => D3 ‘%m-%d-%y’)

CSS Grid

JS Catalyst uses CSS Grid for its layout. Some tips:

The wrapper class for each page must be a different name, or the wrapper is applied globally and styles get put on top of each other. We use the following naming conventions: wrapper-componenttest, wrapper-livedashboard, etc.

General notes:

It is important to set both the container div as well as the div to which the svg is to be appended, to a width and height of 100% so that the component renders correctly.



The template should also contain the resize-observer.

# Details for Chart.js Components

The Chart.js components that we provide are rendered within a wrapper component in order to abstract away data formatting and option setting. The base charts are available in the Chartjs/base directory in case you need more customization.

# Chart.js Base Charts

The base charts are where the drawing of the chart actually happens. Each one extends one of the charts provided by the vue-chartjs library. The props that the base charts receive are chartData, propID, and options. The chartData prop is the already formatted data needed to draw the chart. The propID is used to ensure that each chart has a unique id in the DOM. It gets attached to the canvas element created by the base chart. The options prop is used by the chart to determine different properties such as color of grid lines. This is important when using a light and dark theme because different options are needed to render the charts correctly.

Because the charts need to update when the theme has changed, there is a watcher on the options prop. When the component is first mounted it saves the options passed to it in an internalOptions data property. When new options are passed to the chart the watcher is triggered and tests to see if the options are different from the previously saved internalOptions. If they are, then the chart is destroyed and rerendered with the new options. The internal options are then updated to the new options.

All base charts have a mixin that is also provided by the vue-chartjs library in order to make the chart reactive to new data. Whenever a new data object is passed in as a prop the chart automatically rerenders and displays the updated data.

Most users will not need to change or deal with the base charts. The wrappers will be used in most cases and any changes needed can be done within these.

# Chart.js Wrapper Components

Each Chart.js wrapper is a Vue component that neatly contains all the logic needed for formatting chart data and setting new options based on theme changes. The wrapper component imports the corresponding base chart, and the ChartjsHelper class. The component is what is rendered in the wrapper’s HTML template. The ChartjsHelper is used by all wrappers to format data, set theme colors, and set chart options.

The two props the component receives are the propID, which is used on the base chart, and the dataModel, which is formatted and then passed to the base chart as the chartData prop. The options are generated based on whether the application them is light or dark. This is determined using Vuex and checking the displayTheme property in the state. The colors of the chart are also determined using Vuex and the colorTheme property in the state. These colors are then used when formatting the data because Chart.js requires the colors be directly connected in the data passed to the base chart.

There are two watchers, one for the colors and the other for the displayTheme. Whenever a user toggles the light and dark theme the displayTheme watcher is triggered and new options are created and passed to the base chart. Whenever a user changes the chart theme color the watcher for colors is triggered and new data is created using the dataModel and the new colors. It is then passed to the base chart which updates. The ChartjsHelper is responsible for getting the new options based on the chart, and for getting the new colors based on the new color theme.

# Details for Plotly Components

Most of the Plotly charts in JSCatatlyst make it very easy to display multiple datasets on the same graph. For example in our PlotlyAreaChart.vue the data that this chart can accept makes it easy for multiple lines to be displayed on the same axis. Lets look at an example of the data that can be passed in



The x key is used as the scale on the x-axis. Then any key other than the ‘x’ key in the objects will be interpreted as a separate line. In this contrived example we would have two lines on our graph, StockHigh and StockLow. Once our graph is created it is easy to filter the graph and see how each line looks by itself or together on the same axis thanks to Plotly’s built in tools.

Each of the Plotly Graphs utilize the PlotlyHelper class which is located in **src » common » plotlyHelper.js.** This helper is needed in order to parse the data being passed to the component into the correct format needed by Plotly. If you need differently formatted data to be passed in as the dataModel for the Plotly component, you can go into these functions and change them to fit your needs.

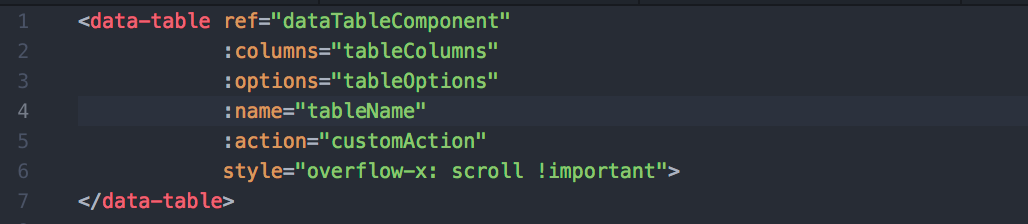
# How to Create A Data Table

The UI uses [vue-tables-2](https://github.com/matfish2/vue-tables-2) to create data tables. To make a table, insert the data table component that’s found in **src » components » Tables » DataTable.vue** into the dashboard.

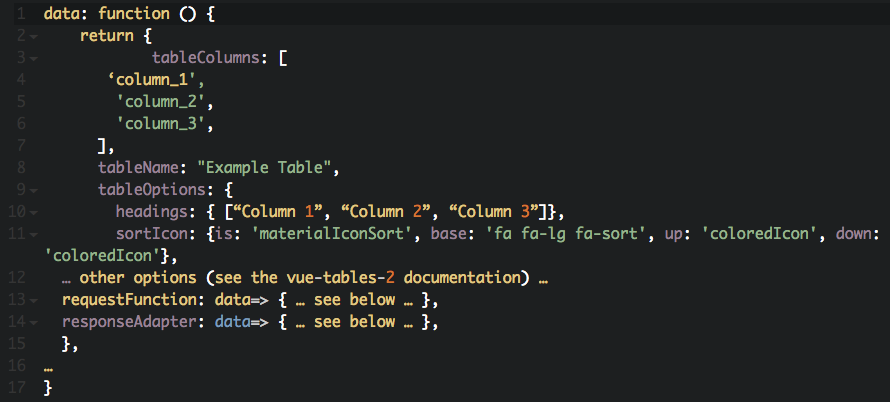
There are two kinds of tables available from vue-tables-2. If the API URL to fetch your data includes the phrase “?draw,” you’ll want a ServerTable, which uses the requestFunction / responseAdapter pattern. Otherwise, you will want a ClientTable.

## Server Table

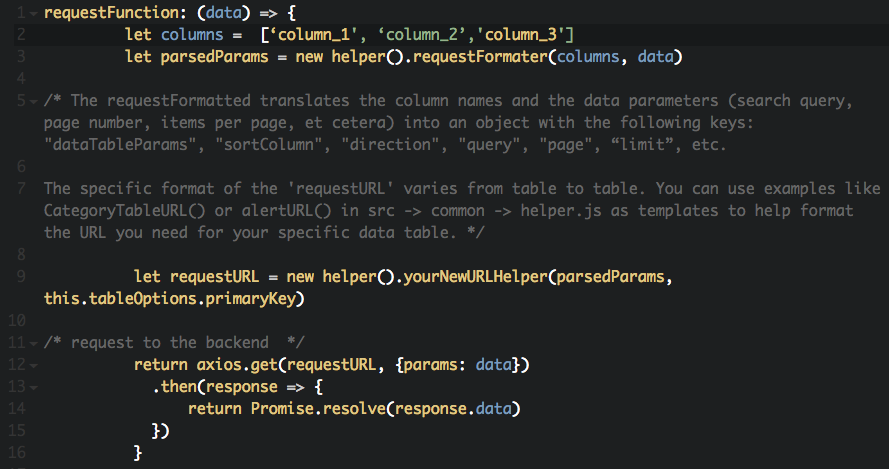
Drop the component into the dashboard’s <template>:



You can then pass it data properties that you define in the data function:



To use vue-tables-2 (front-end display component) and [datatables.net](http://datatables.net) (backend API component) together, you must make some customizations. Here is an example of a request function to fetch data for the table:

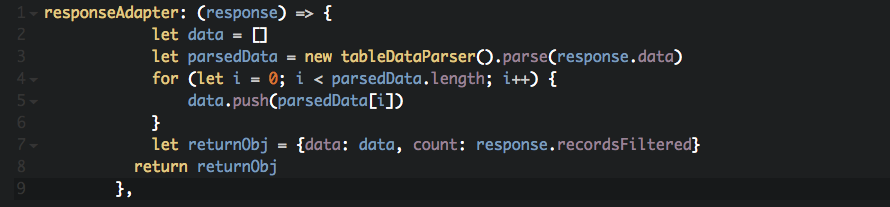


The response from these API calls may need to be parsed to fit the format required by vue-tables-2. Some things to be aware of:

* Sometimes the backend returns data points with inline HTML, i.e. [“data\_point”, 3.5, “<span class=“blue”>et cetera</span>”]
* Vue-tables-2 wants each row formatted as {column\_name: “data\_point”, another\_col: 3.5, one\_more: “et cetera”}

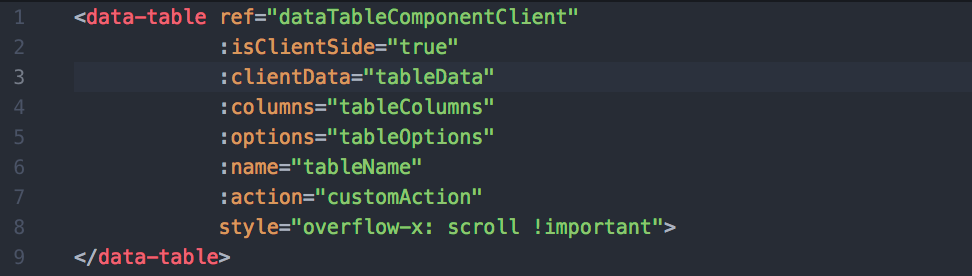
Parsing methods are located in **src»common»tableDataParser.js**. If you need to make a custom parser function, you can use the functions in this file as a template.

Here is an example of a function that reformats the response from datatables.net so that it matches the format required by vue-tables2:



## Client Table

To use a client table, drop the <data-table> component into the dashboard like so:



Then pass it data properties that you define in the data function:

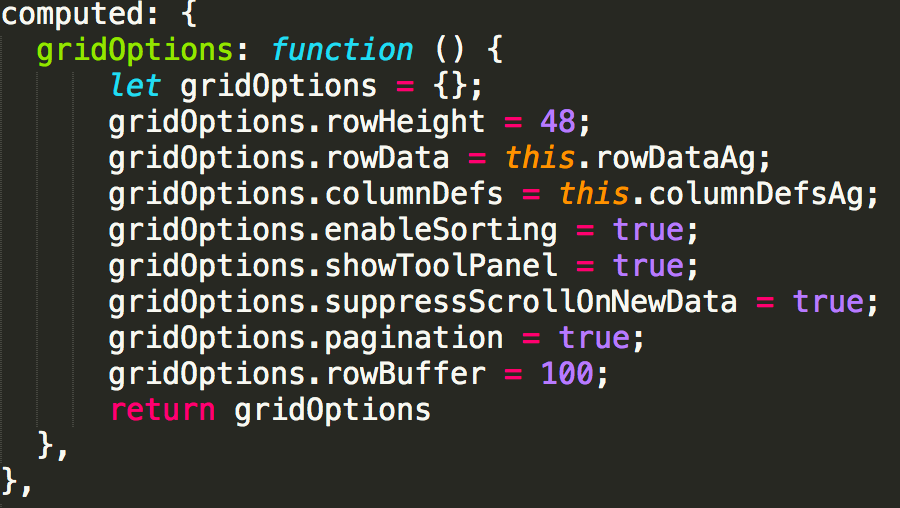
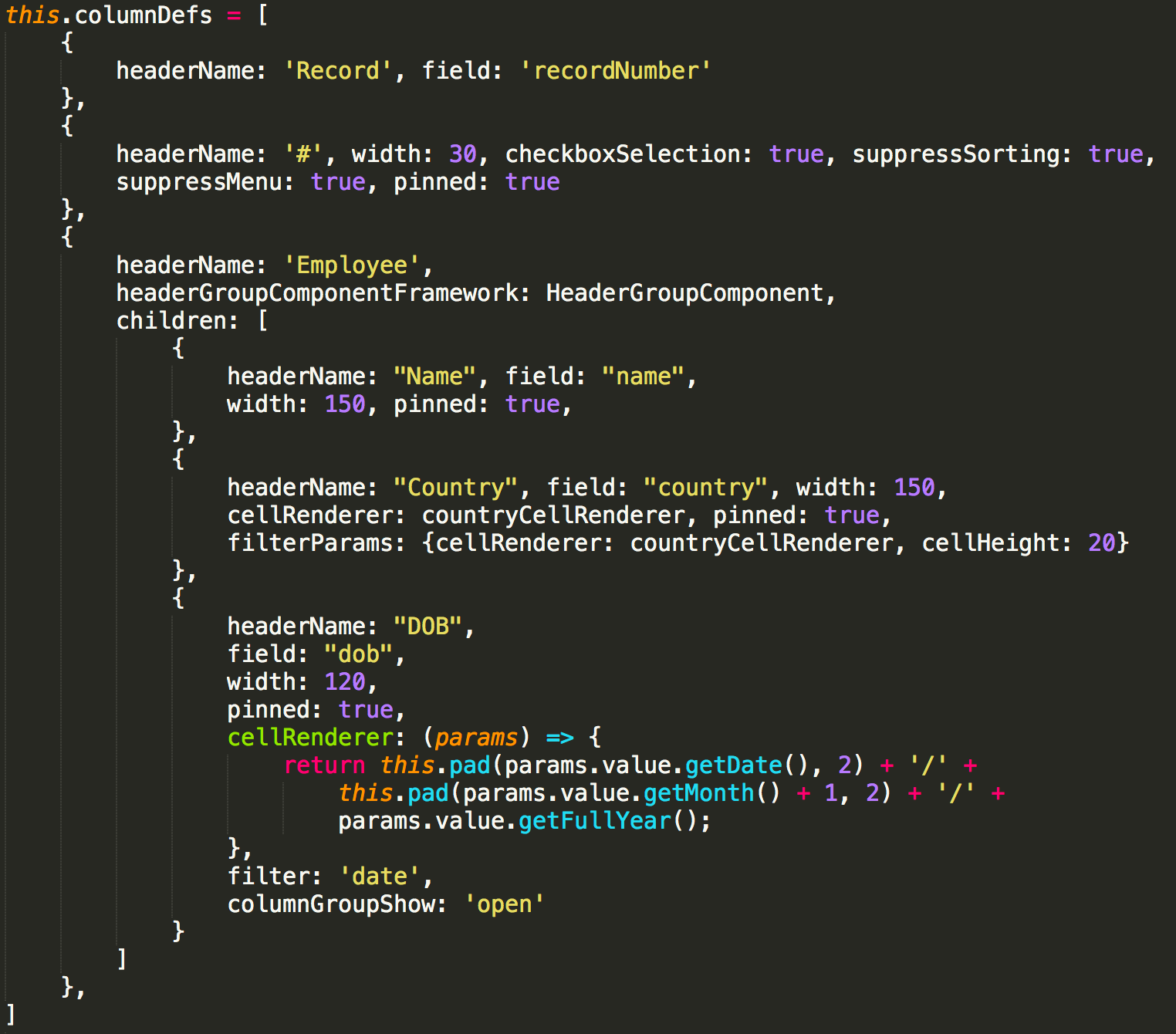


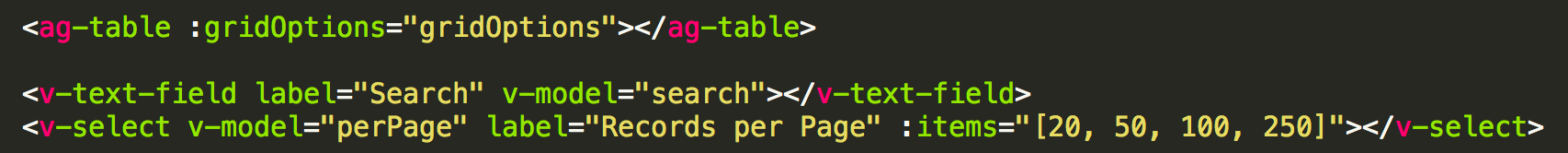
You can then use the regular API pattern to grab the data for tableData. Make sure to use a tableDataParser method on the result in the callback, then assign that parsedData to localThis.tableData.

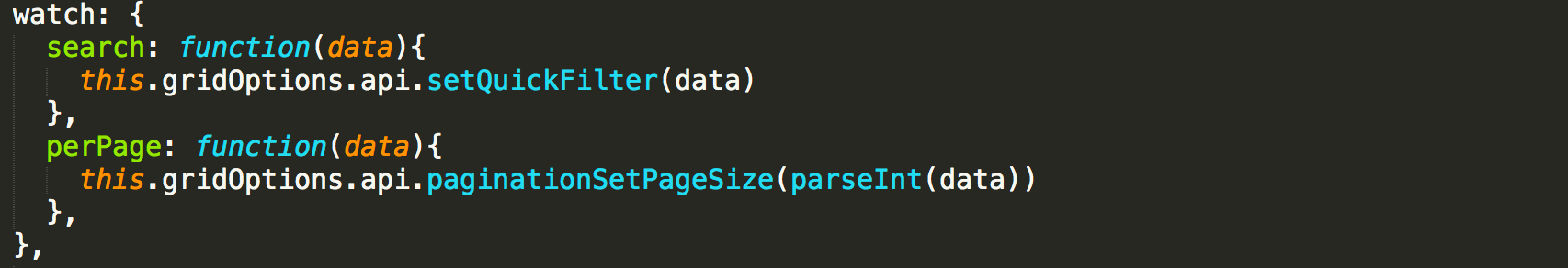
# How to Create A Blotter/Data Grid

For larger datasets (30,000+ rows), [ag-grid](https://github.com/ag-grid/ag-grid-vue) is preferred over vue-tables-2. To make a table, insert the data table component that’s found in **src » components » Tables » AgTable.vue** into the dashboard.

Any customization to ag-grid-vue is done through the gridOptions prop. This includes defining the data itself, the column headers, visual properties such as row height, business logic such as cell click behavior, and other ag-grid specific options. Since this object is complex and subject to changes, you should define it as a computed property of the parent component, and pass it as a prop to the ag-table component.

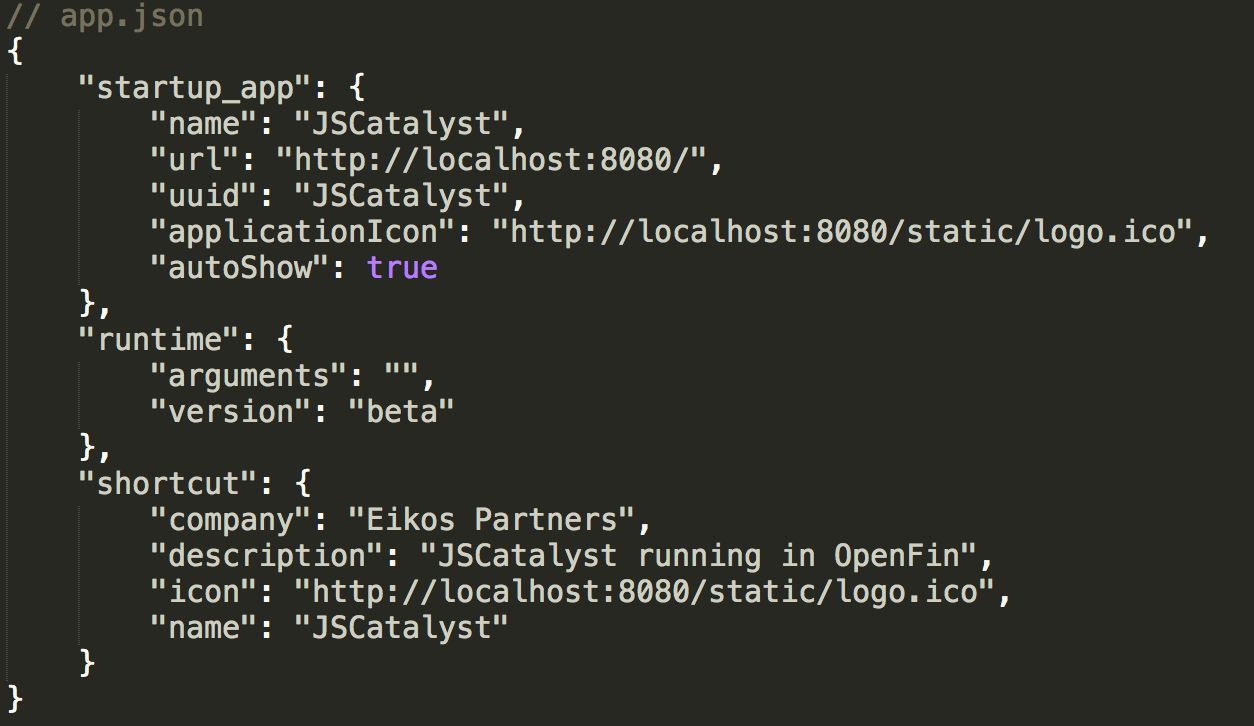
Each object in the columnDefs array must include, at minimum, a headerName and a field. There are many other options available to customize your column and how it renders.

The rowData array must have one object per row. The object’s keys must match the field values from the columnDefs.

The base configuration of ag-grid-vue does not include a search field or pagination options. It is simple, however, to set up separate components that perform these functions by accessing the ag-grid API.

# How to Open JSCatalyst in OpenFin

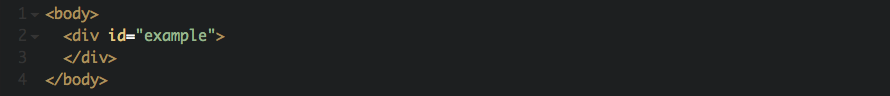
[OpenFin](https://openfin.co/) is a container that allows our application to run on the desktop rather than on the web (through a browser). It runs on Windows, Mac, and Linux.

To run JSCatalyst on OpenFin, you need a) to ensure OpenFin is installed on your computer (npm install -g openfin-cli) and b) a config file (currently ./app.json) that defines OpenFin’s runtime parameters. You can launch the application with openfin --launch --config ./app.json, which we have aliased to npm run fin. Further options for this config file are available [here](https://openfin.co/documentation/application-config/). 

# How to Create A New Dashboard

Each dashboard is kept in its own folder under **src » pages** and consists of three files: app.html, app.js, and App.vue. The folder can also contain any components that are specific to that dashboard.

* app.html simply renders the div with the id that is defined in App.vue:



* app.js is where you import the necessary dependencies and create the Vue instance.



* In this Vue instance, we’re assigning the value of ‘el’ (short for element) to the same id as specified in app.html and App.vue.
* The value of render is telling the UI to render App.vue.
* By adding the store to this object, we’re telling the UI that we’re using Vuex.
* App.vue is where the magic happens. The first two files are mostly boilerplate code, whereas all the HTML, logic, and styling for the dashboard happens in App.vue.

App.vue consists of three areas:

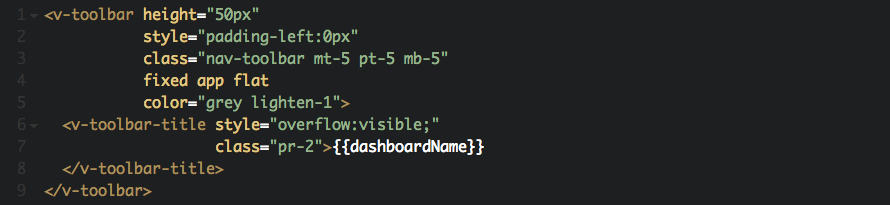
* <template> tag - contains the HTML.
* <script> tag - contains your JavaScript and logic.
* <style> tag - contains styling for the dashboard.

## <template>

Note again that each dashboard’s App.vue file needs to be wrapped in a <v-app> tag in order for any Vuetify elements to render correctly.

The primary toolbar (with the User dropdown and Select a Dashboard dropdown) will appear automatically on each new dashboard; you don’t have to worry about including code for it. Due to its position in dashboards/dashboards, the toolbar is fixed across all screens (each new dashboard is technically rendered beneath the toolbar).

You will, however, need to include code for the secondary toolbar with the dashboard name. Any top-level tabs for the dashboard will be included in this toolbar. Here is the code for the secondary toolbar, note that leaving margins and paddings for the top (class="nav-toolbar mt-5 pt-5 mb-5”) is necessary to make room for the primary toolbar.



## <script>

Import necessary dependencies in the script tag, including helper files and custom components. Anything that you will reference in the component instance needs to be imported here.

## Data Properties

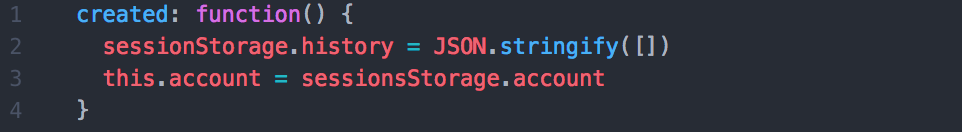
When defining a component, you must declare a data function that returns all the necessary data properties for that component. Vue will recursively convert its properties into getter/setters to make it “reactive”. When the values of those properties change, the view will “react”, updating to match the new values. Every time a new instance is created, the data function is called and returns a fresh copy of the initial data.



## Lifecycle Hooks

Vue offers a series of lifecycle hooks where you can add methods that are called at specific stages. All lifecycle hooks are called with their “this” context pointing to the Vue instance invoking it. Some of the most used hooks include:

* created: called after the instance is created and the following items have been initialized: data observation, computed properties, methods, watch/event callbacks. The mounting phase has not started yet.



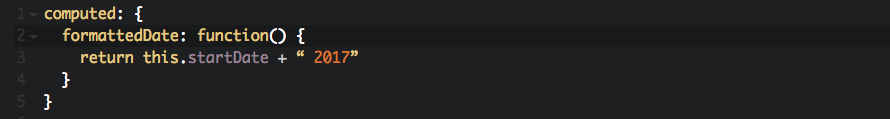
* mounted: called after the instance has been mounted, where el is replaced by the newly created vm.$el. Note that mounted does not guarantee that all child components have also been mounted.
* updated: called after a data change causes the virtual DOM to be re-rendered and patched. While you can perform DOM-dependent operations here, you should generally avoid changing state inside the hook. It’s better to use a computed property or watcher instead.

For more information on lifecycle hooks, see the [Vue documentation](https://vuejs.org/v2/guide/instance.html#Lifecycle-Diagram).

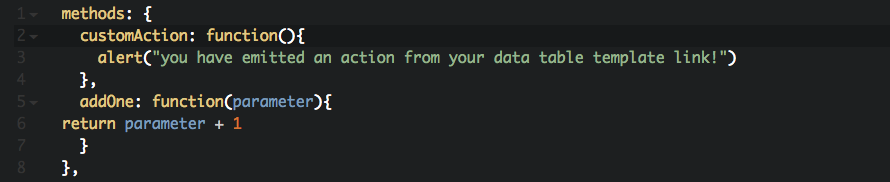
## Instance Methods

Vue offers other instance methods and event handlers that can be called inside the lifecycle hooks or independent of the hooks to handle the logic of the component.

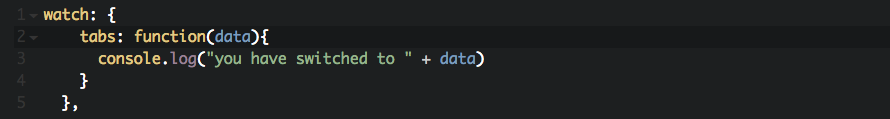
computed: computed properties perform transformations or calculations on properties that are displayed in the view. These changes are cached and only updated when needed.

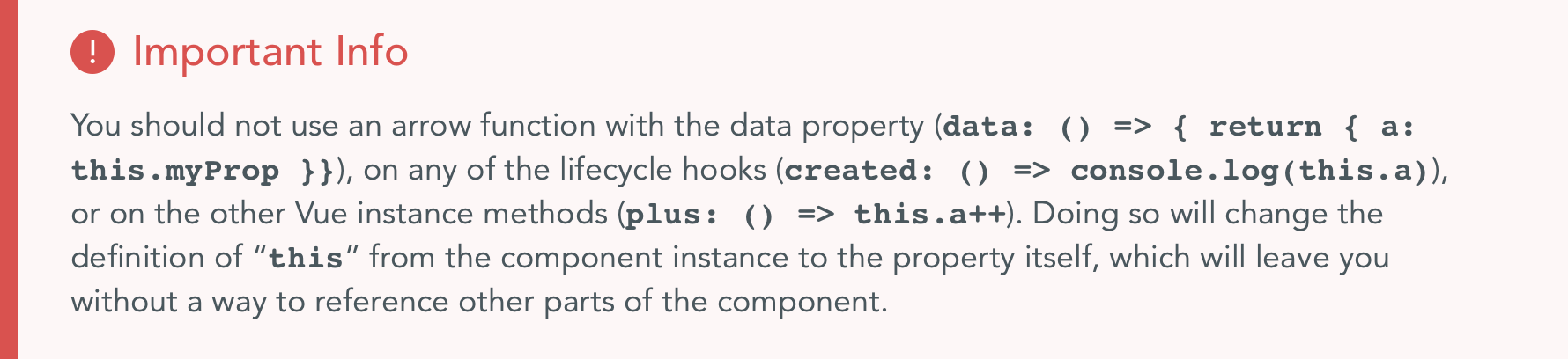


methods: general methods to be mixed into the Vue instance.



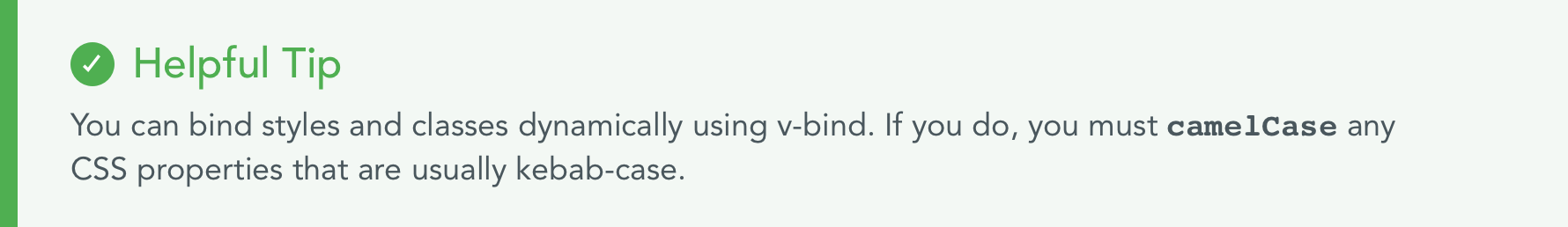
watch: an object where keys are expressions to watch and values are the corresponding callbacks.

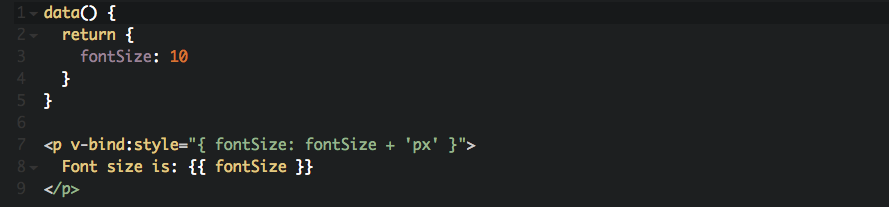




## <style>

Any styling for the component is placed here. Note that styles defined in the component are applied globally. Therefore, it’s preferable to define classes on your elements and scope styles to those classes.





# Vue-Specific Tips

Vue has its own [style guide](https://vuejs.org/v2/style-guide/), which outlines patterns and best practices from the Vue developers. Its suggestions are helpfully divided into Essential, Strongly Recommended, Recommended, and Use With Caution.

GitLab, one of the larger companies that is using Vue, has published a [style guide](https://docs.gitlab.com/ee/development/fe_guide/vue.html) as well.

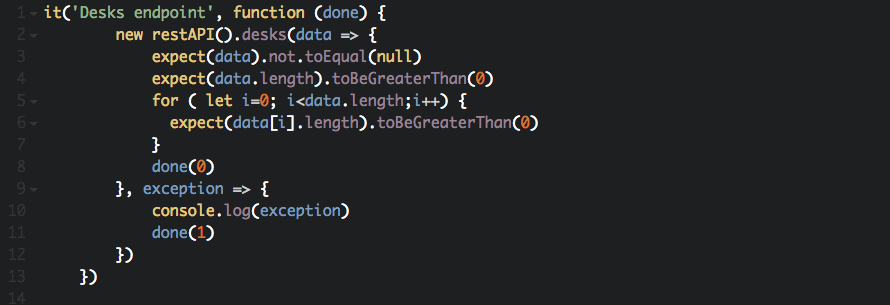
For Vuetify-specific questions, the Vuetify discord channel is very active and engaged. The primary developer is almost always present and willing to help, and you will usually find the problem already mentioned in the #get-help channel.

If you are googling or using stack overflow with questions, remember that the application uses Vue 2.x, which may require different solutions than 1.x.

# Testing

The UI uses [Jasmine](https://jasmine.github.io/) and [Jest](https://facebook.github.io/jest/) for unit testing, and [Cypress](https://www.cypress.io/) for integration tests.

Unit tests are found in the **spec** folder. Tests for checking the API calls are located in **spec » services,** while tests for individual components are found in **spec » vue**. Here is an example of a unit test for an API endpoint:



Integration tests are located in the **cypress** folder, organized by dashboard. The integration tests cover all components in the dashboard. Here is an example of a Cypress integration test for the interval filter:



We have written tests for the Example dashboard, so you can see what tests are helpful on basic components. If you need to write your own tests, you can follow the testing patterns already created, or refer to the testing framework’s documentation.

# Vuex in JSCatalyst

JSCatalyst makes use of Vuex as an application state management tool. Vuex functions similarly to other state management libraries such as Redux, and has four main parts. They are the state, getters, mutations, and actions.

* State – the state is where all the variables are stored that the application’s components need to access
* Getters – the getters are used if some sort of calculation needs to be done on variables in the state and are needed by multiple components
* Mutations – the mutations are used to make synchronous changes directly to variables in the state
* Actions – the actions are used when asynchronous work must be done (such as making API calls) and when the response is received they then commit a mutation

JSCatalyst makes use of most of these parts of Vuex in order to take care of user authentication, theming, and a couple of miscellaneous use cases.

The Vuex store can be found in the **src » store » index.js** file. Because there is not much state to manage in the base JSCatalyst repo, all of the store is located within this file. However, as projects increase in size it is normal to break down the store into related modules. For example the current store could be broken into an Auth and Theme module if needed. To do this first create a Modules folder, then within that folder create an Auth.js and Theme.js file. In these files create move whatever state, getter, mutations, or actions need and set them to const variables. Then at the end of the file export and object with all the variables. Next import the module into the index.js file and create a new attribute on the store called modules. Modules should be an object and all that is needed is to put the imported module into the modules object. Vuex will take care of merging all of the state, getters, mutations, and actions into the global store.

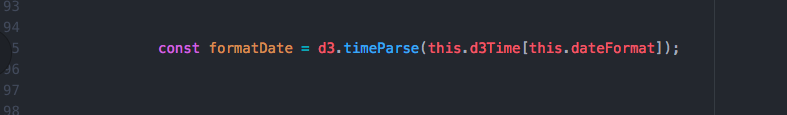
In addition to creating modules needed for specific parts of the application it is also best to split the root store data into separate files. This means taking all base state, getters, mutations, and actions and splitting them into state.js, getters.js, mutations.js, and actions.js files. These should be created in the store folder and should export all the data or functions defined within. Then each should be imported into the main store file and added to the root Vuex store.

# Project Mixins

JSCatalyst has three main mixins that are used throughout the project. In this section we’ll talk about each and their function and go over how to add your own mixin. All mixins can be found in the src/mixins directory if JSCatalyst. Each mixin should be a .js file.

# Format Time Mixin

The format time mixin is essential for D3 graphs that deal with time series data. Information on what charts require this mixin can be found in our in-depth description of each D3 component. The purpose of this mixin is to make it easier for the developer to display their time related data easily by passing in the format of their date data. This mixin maps the d3Time property to the data of the Vue component it is imported into (ex. a component with this mixin could access d3Time by doing this.d3Time).

Not only are there many different libraries used for formatting dates, but D3 has it’s own date formatting syntax which makes it difficult for the developer to figure out how the date will be displayed. And because D3 has it’s own date formatting library if the date passed in is not the same exact format needed by D3 all dates will be returned as null after being parsed. This mixin fixes this issue by mapping the provided dateFormat prop (ex. ‘MM-DD-YY’) to the corresponding D3 date format (ex. ‘%m-%d-%y’). Using it in the Vue component would look like this

The formatDate variable will be used by the D3 when creating the chart to correctly parse the dates in the data. Since this.d3Time is an object we can easily get the necessary D3 date parsing string by getting the value at the dateFormat prop.

The dateFormat prop should be a in a Moment.js format string. Moment.js is used throughout the project and is more reliable and common than using the built in Javascript date constructor.

# Style Toggler Mixin

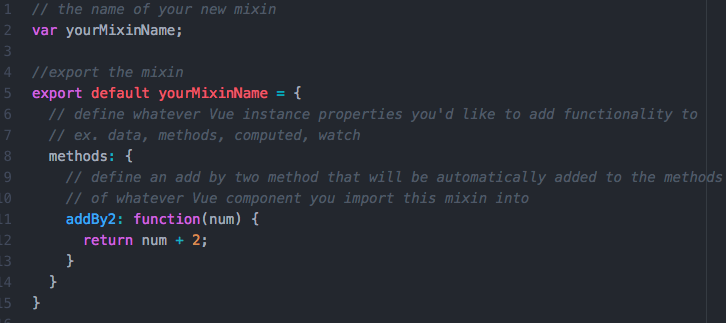
The style toggle mixin is essential for controlling the theming throughout the project. There are three functions within this mixin.

* toggleDark() – this function is responsible for change the theme between light and dark. This is done by getting the current theme class on the root app, finding the opposite, removing the current theme class, and then finally adding the new theme class. It then commits the new theme to the store which causes all visualizations to update for the new theme. Note: the light and dark theme classes of your application must always be named ‘theme--dark’ for dark and ‘theme--light’ for light
* chooseTheme(theme) – This function is responsible for changing the color theme of all of the visualizations in the project. This is done in a similar way to the toggleDark function. It gets all the classes on the root app except the current color theme class. It then takes these classes and adds the new color theme class and sets the root app className to these classes. It then commits the change to the store, and calls our third function to store the array of current colors in the state. Note: all of the color theme classes must be named like ‘blue-theme’, ‘red-theme’, etc so that the correct theme colors can be applied
* themeColorArray(theme) – This function is responsible for getting the current color theme colors and returning them in an array for some visualizations to use. This is done by parsing the style sheets applied to the application and finding the one that contains the current color theme class (ex. ‘.blue-theme’). Then it takes the list of CSS variables and the corresponding colors and transforms that into an array of color strings. This is then returned and is used in the Vuex store to tell the visualizations what colors to use

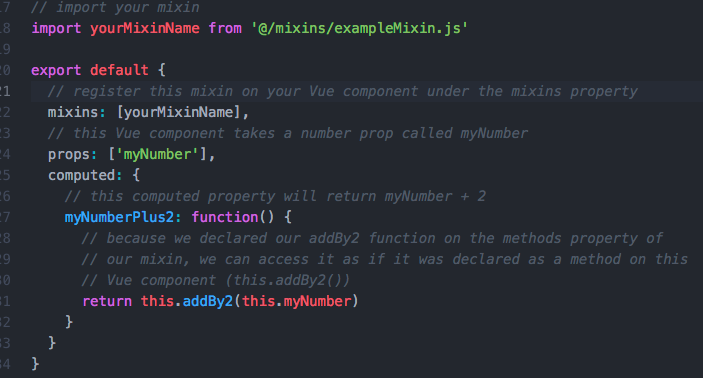
With these three functions it is possible to easily set the current theme of the application, and set the color theme of all the visualizations in the application.

# Adding your own Mixin

In order to add your own mixin to the project create a file in the src/mixins directory and name it your mixin name. At the top of your file declare the variable with the name of your mixin. Then on the next line export default yourMixinName and set it equal to an object. Within that object you can have any keys you would regularly have in a Vue instance such as data, computed, methods, etc. These will get merged with whatever Vue instance you import the mixin into. For example if you wanted a mixin to have a method addBy2, you would declare the methods key in your mixin and then the methods would be an object that contained your addBy2 as a key and the value would be the function. Now whatever Vue instance that imports this mixin will have the addBy2 function available to it like any other method declared on it (ex. You could call the mixin function by doing this.addBy2(someNumber)). To find out more about how Vue merges mixins and deals with conflicting keys on mixins please look at the [Vue documentation here](https://vuejs.org/v2/guide/mixins.html#Option-Merging). Our finished mixin example from above would look like this



Then to import this mixin into your Vue component it would look like this



This example can be found in the src/mixins directory of JSCatalyst for reference if needed.

# Project Plugins

JSCatalyst makes use of many third party plugins such as Vuex for state management and Vue-Router for application routing. In Vue plugins can have varied functionality like adding a custom directive to the Vue instance (ex. adding a new $attribute) or adding a global mixin accessible in all Vue instances. There is one important custom plugin included in JSCatalyst called screenshare.js.

# Screen Share Plugin

The screen share plugin is used to add functionality for sharing charts and dashboards between users. This allows a developer or designer to drop a button on any page with charts and by setting the @click event to shareAll a user can share their page with any other user. All of this is done by broadcasting the chart information through websockets. Once the message is sent the all that is needed on the page receiving it is a Receiver Component. This component should get passed the componentArray which it then iterates through and creates a loader component with that data.

This plugin works by adding a global mixin to the all Vue components in the application. It adds the shareAll and the submit methods to each component which are used for sending charts. As mentioned earlier the shareAll function can be tied to a button and when clicked all charts and components on the current page will be sent out. The submit function needs to be tied to a button with a select, allowing for the user to choose a chart from their page in the select and send just that single chart.

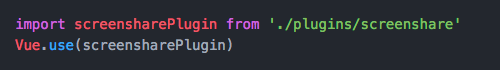
The receiveCharts method is also added by the mixin to allow any page to receive charts. This function parses out all the information needed from the websocket message which can then be passed in to the receiver component. This function also pulls out the styles for how the page should be laid out with the CSS Grid information.

The final attributes the mixin adds are a computed property named charts and a data property named shareableChart. The computed property parses the components on the current page and gets the name of each. These are then stored in array and are available to be used with the select to choose any of the components on the page to send. The data property shareableChart holds on to the currently selected chart. This value gets used if the submit event is called.

Now we’ll walk through a demo of how to add chart sending functionality to a page and how to add receiving functionality to a page.

# Screen Share Walkthrough

The first thing that is needed is if it is not already done in main.js you must import the plugin and tell Vue to use the plugin.



This will install the plugin on the global Vue instance. Next, go to any page that has charts on it and add a Chart Sender component. The Chart Sender Component encapsulates all the logic for sharing dashboards between users. The component’s code contains a select and two buttons. The select allows for a user to select a single chart to chare with the Share Chart button. Or if the whole dashboard needs to be sent a user can press the Share Dashboard button.

All the properties that are used in this component are directly available from the plugin so there is no need to import anything or do anything extra.

Now let’s set up the receiver. Any page or section of a page can be turned into a chart receiver. All that is needed is the Receiver Component. On the page that you want to receive charts import and register the Reciever Component.





Now all you have to do is place the component in the template of your page.



Now your page is able to receive charts from any other user. The Receiver component gets all the functionality for receiving charts from the plugin. The plugin adds a socket listener to listen for shareCharts to be emitted over websockets. When this happens the recieveCharts function is called and the Receiver Component gets all of the components, styles, and classes needed to replicate the dashboard on the receiver’s application.

The Receiver component works by iterating over all the component charts that have been sent and recreating them using a Loader Component. The Loader Component dynamically imports the charts needed to recreate the dashboard and feeds in all the props and styles of the original chart.

# Auth Plugin

The auth plugin is used to encapsulate all the logic needed to add front end auth protection to a Vue project. When using the auth plugin it is important to pass in the Vuex store, the Vue Router instance, and the base URL for your backend api. The way the auth plugin works is it adds an auth module to the store which keeps track of authentication logic and it adds a global beforeEach to the Vue Router in order to check authentication before visiting a page.

It also adds a couple actions to the Vuex store which are responsible for logging in, signing up, logging out, and checking authorization. Let’s look at the login action as an example.



Right now the auth plugin requires that you use Axios to carry out the requests to the api. In the future this plugin will be refactored to allow for other libraries to be used to make requests. This action makes a post request to the backend api. The route that must be available on the api is the ‘/auth/login’ route. This route should take the username and password of the user trying to login and check it with the users in the database. Once there is a response it checks to see if it was a success. If so it commits the is Authenticated mutation allowing the user to navigate through the app. It then commits the authMessage and the expiresSoon mutations to display a message to the user and to tell the application the token issued does not expire soon.

# Theme Plugin

The theme plugin encapsulates all the logic necessary to add custom theming for all the visualizations in JSCatalyst. The plugin works by taking the Vuex store and an array of themes for your application. A theme module is added to the store which is responsible for keeping track of light and dark theming and the color theming of charts. The themes are used to determine all possible themes available in your application. It is important that each color in the theme array corresponds to a theme in your css with the naming convention `(color)-theme`. Please look at the style section of these docs for further instructions on how to create your own theme.

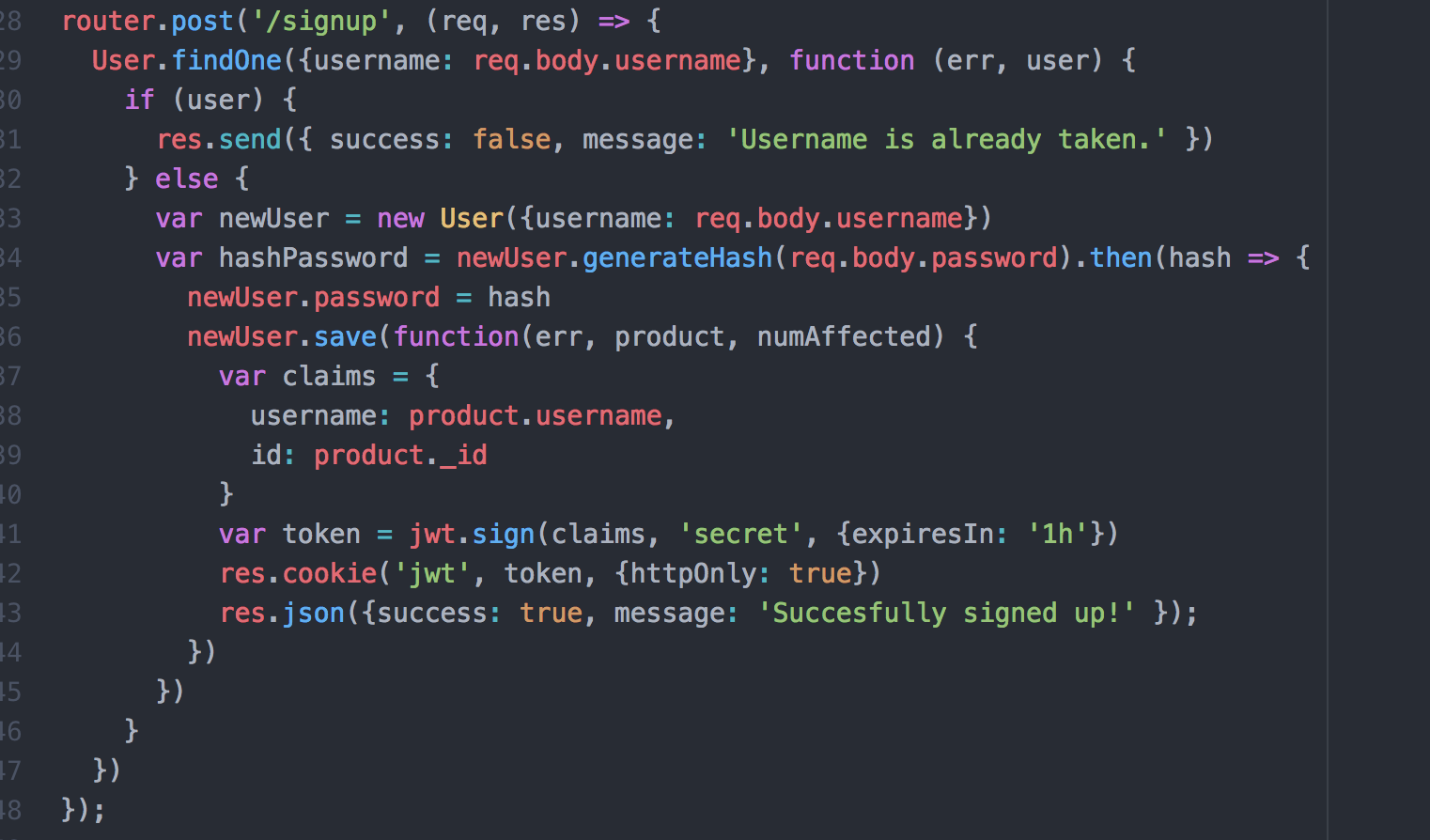
The theme module will also keep track of your application’s light and dark theming. This will ensure that all charts and their axes will update to the correct color when changing from light to dark or dark to light. The light and dark themes should be named with the ‘theme--(light/dark)’. This ensures that when this plugin is used with the Style Toggler Mixin all charts will be updated correctly.

# Authentication in JSCatalyst

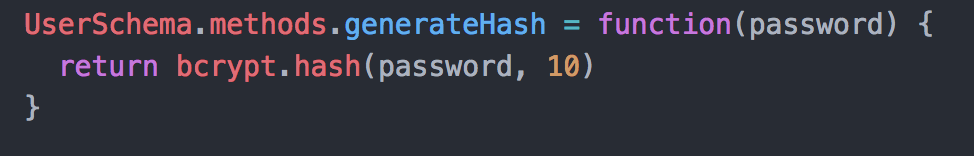
JSCatalyst has authentication built right in to the project and is ready to use. The project uses JSON Web Tokens (JWT) and cookies for security and to ensure that users are authorized to access your servers or data. This section will cover how users are verified on both the server and the frontend throughout the application. All the information for the databases and the routes for the server can be found in the server folder in the root of the project. This folder contains the models and routes for the Mongo and Postgres databases to use.

# Signing up and Logging in

When a user access JSCatalyst they are brought to the home page and can choose to sign up or log in. When a user signs a request is made to the server to ensure that the username is not already taken. If it is not then the user will be created in the database. The code looks like this,



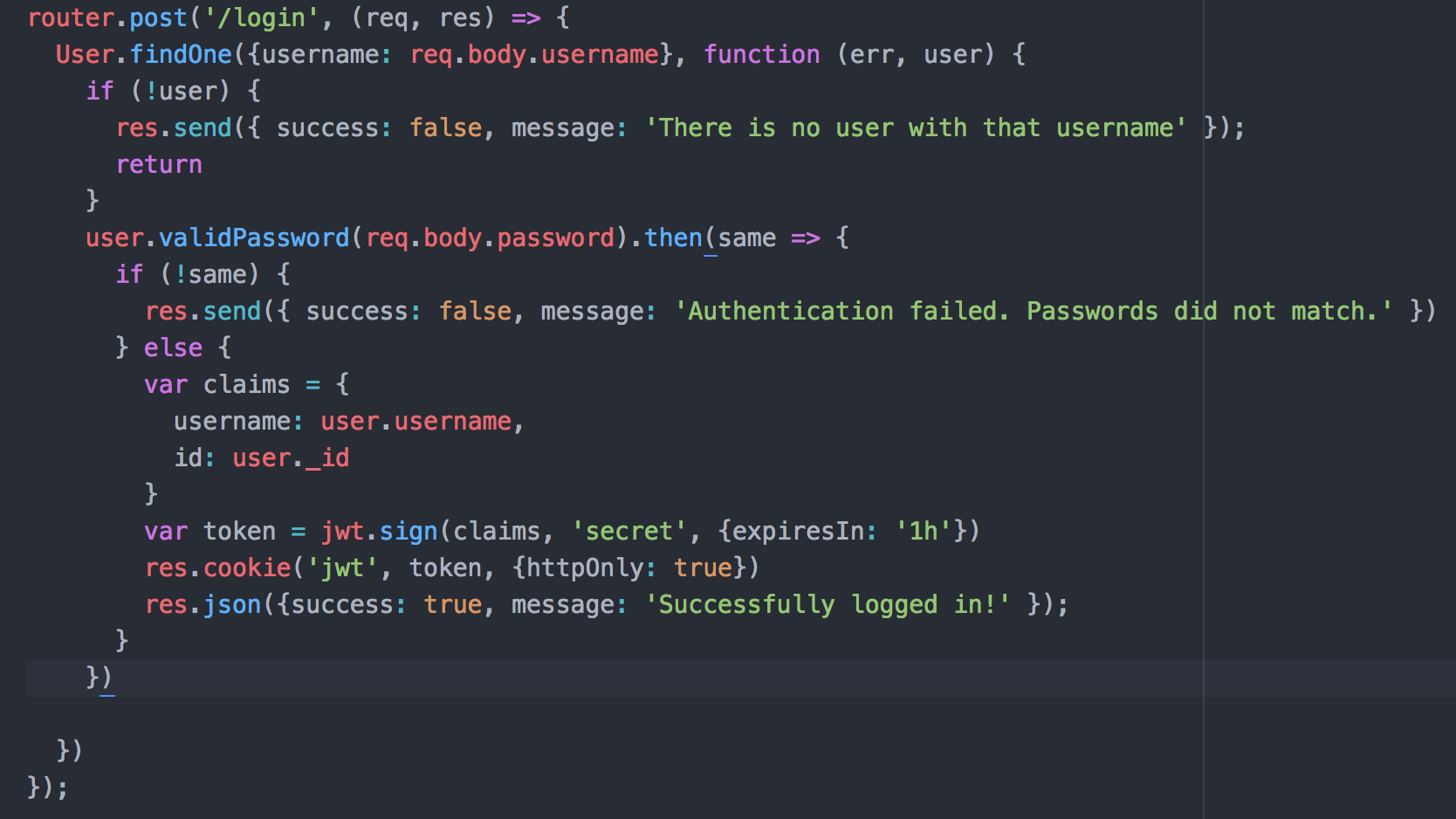
If the user exists then an error message is sent back. If not a new user is created with the username. Then the user’s password is salted and hashed using bcrypt. The function generateHash is a method on the user model and looks like this,



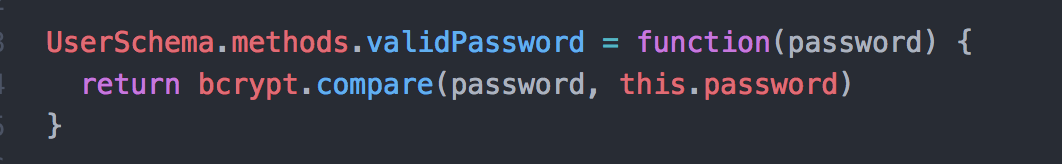
The function takes in the password and returns the bcrypt.hash() function. The second argument is the number of rounds that the password will go through to hash it. The actual number of rounds is 2^rounds so the higher the number the longer it will take. But the password will be more secure. The general time it takes for multiple values can be found on the [bcrypt github page](https://github.com/kelektiv/node.bcrypt.js#a-note-on-rounds). The function returns a promise with the hashed values because the hashing is asynchronous to prevent blocking the server.

After the password is hashed it is added to the new user instance and saved to the databse. The save function accepts a callback that will take the error and user as arguments. This callback is fired after the user is successfully or unsuccessfully saved in the Mongo database. When the user is save we create a claims object to be used when creating the JWT token. The JWT is then created with three arguments. The claims that were created, the secret needed to decode the token, and the expiration date or time it is valid. This cookie is then added to the response object with the key of ‘jwt’ and the option httpOnly set to true so that the cookie is inaccessible by javascript on the user interface. This protects from malicious scripts that could try to access the cookie and get its data. Finally a response is sent back to the frontend with a success message.

When logging in the flow is very similar. The user’s username and password are checked to validate that they are correct and if not they are denied. The code looks like this,



If no user is found then and error message is sent and the request is exited. If a user was found with the username then the password is checked against the hashed password stored in the database. This is done with the validPassword() function on the user instance. It looks like this,

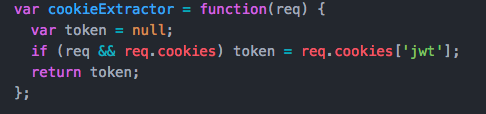


This function takes in the password supplied by the user and returns the bcrypt compare function. This function is passed the password and the password that is stored for that username in the database. It then compares them to see if the hashes match. It returns a promise so when it is done we check to see if they are not the same. If they aren’t an error message is returned. If they are then we create a JWT token and add it to a cookie and send it back with a success message just like in the /signup route.

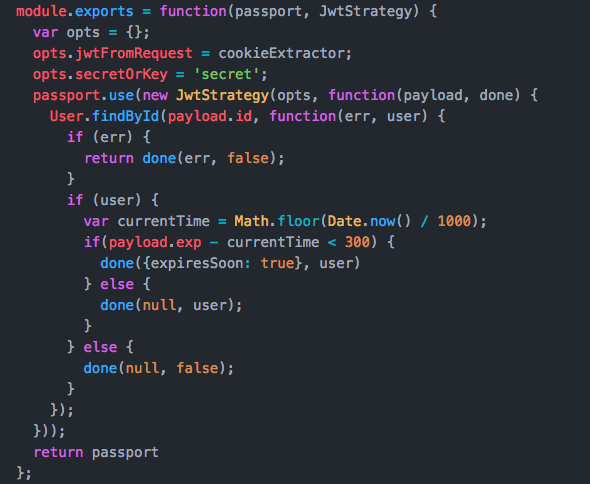
# After being authenticated

After the user signs up or is logged in they have the cookie in their browser. Whenever the user makes a request to the server the cookie is sent with the request and is checked to makes sure it is valid. This is done with a middleware library called Passport. Passport works by intercepting the request to the server and using a strategy to authenticate the request before allowing the user to access that route. There are many strategies available for passport to use but the one that JSCatalyst uses is the passport-jwt library. This strategy is used with the JWT tokens that are issued to authenticated users when they sign up or login. It works by extracting the JWT token from the cookie that is sent with the request. It then takes the token and decrypts it to check that there is a valid user id connected to the user in the database. If there is then the request is allowed to continue. If not then the user is denied access to the api route and is redirected to the home page.

The passport instance that JSCatalyst uses is defined in the passport.config.js file in the root of the project. In this file we customize the passport instance to have the JWT Strategy available to it and we define our custom JWT extraction function. This extraction function is used to get the JWT token out of the cookie. The extraction function looks like this,

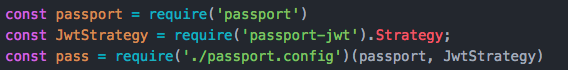


All this function does is check the request for cookies and if it does have cookies extract the one with the key of ‘jwt.’ The function that is then exported by this file looks like this,



It takes the Passport module and the JwtStrategy as arguments. First an options object is created and has two keys defined on it: jwtFromRequest and secretOrKey. jwtFromRequest is the function that will be used for the extracting the JWT token from the cookie. secretOrKey is the same secret that is used encode and decode the JWT token. In this contrived example it is just the string ‘secret.’ We then call the use function on the Passport module to make it use the JwtStrategy for authentication. The JwtStrategy is passed our options we defined earlier and then accepts a call back where a user is searched for by the id that is stored in the JWT. If there is an error or it does not find a user with that id then the error is thrown or the user is denied access to the route. If a user is found then they are allowed to continue to the route they requested. There is a check in place as well to see if the user’s token will expire in less than five minutes. This is done by checking the token’s expiration time against the current time in Unix timeformat. If it will expire soon then we send a custom error along with the user as a flag. This will get used later on in the code.

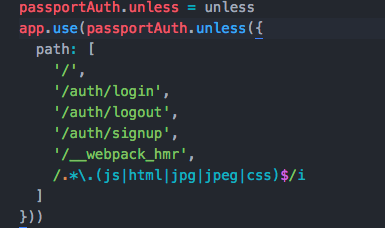
In the app.js file the passport instance the project uses is created by using our passport.config.js function. First all the libraries are imported and then the custom passport instance is created,



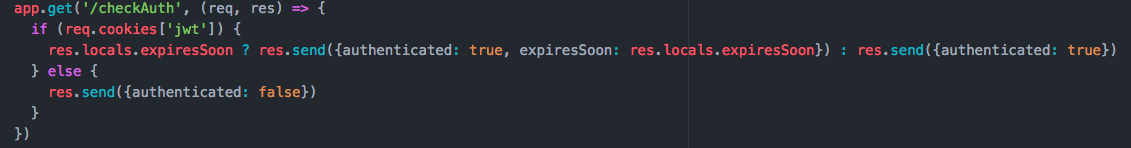
Then we initialize passport to work with the app. Next a wrapper is put around the custom passport instance so that the user can be redirected to the home page and can have their cookie taken away if the JWT token is expired. The wrapper function allows for access to the response so that the cookie can be cleared. The error that is also checked to see if the token will expire soon. If the token will expire soon then a local variable is added to the response so that it can be passed along and sent to the front end with other information. Within the wrapper the authenticate function is called with the ‘jwt’ strategy that we added to the passport instance. If the user is found and the token is not expired then the user is authorized. Additional options are added to the authenticate method declaring we do not want sessions and that if the authentication fails the user should be redirected to the home page.



With all of this set up, Passport will now authenticate every request to the server. However there are some routes that need to be allowed for the application to load and for users to have the ability to login and signup. To do this the express-unless library is used.



In this code we tell the app to use the passport authentication on ever request other than the ones specified in the path array. One final important part of the code is the checkAuth path that is defined in app.js.



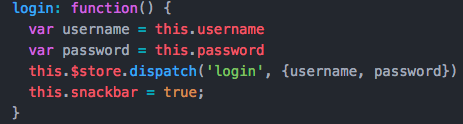
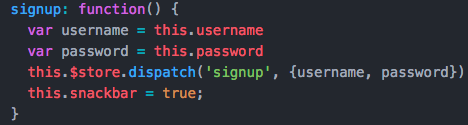
This route that is defined is used to check the user’s authentication before accessing any pages in the application. Because JSCatalyst is a SPA requests are not sent to the server every time a different page is loaded by the Vue Router. This request will check the user’s authentication and will check if their token will expires soon. This ensures that a user cannot access protected routes without being authenticated first.

# Frontend Auth Protections

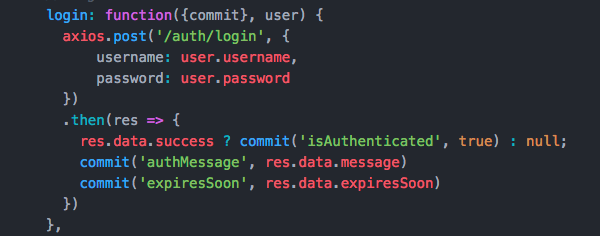
The previous section was mostly on how the user makes request to the server and how they are authenticated and authorized to access data. This section is about the protections that are in place in the front end code to ensure that a user cannot access a page they are not allowed to. These protections are necessary due to the fact that JSCatalyst is a single page application (SPA). As explained in the section above protections have to be in place because Vue Router does not make any requests to the server when visiting a different page.

All the logic needed for the frontend auth protections are encapsulated in the auth.js plugin. This plugin works by adding an auth module to the store and by creating a global beforeEach for the Vue Router to check authentication.

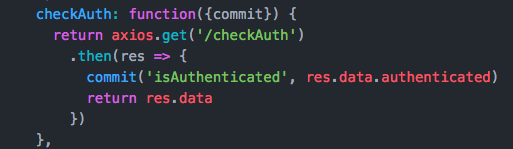
The authentication of the user is stored in the Vuex state under a Boolean value. When a user is logs in or signs up they either dispatch the ‘login’ or ‘signup’ store actions. The code for both looks like this,



In the **src » plugins » auth.js** file the requests get made when these actions are dispatched. These actions make post requests to the server with the username and password entered by the user. When the request goes through the backend flow discussed in the previous section a response is returned with success, message, and expiresSoon properties. These are then used to commit changes to the Vuex store about the user’s authentication.



Another important action added to the Vuex store is checkAuth. This is used to ping the server to check if a user is authenticated. This is also used to check if a user’s token will expire soon. It commits the user’s authentication status to the Vuex state.

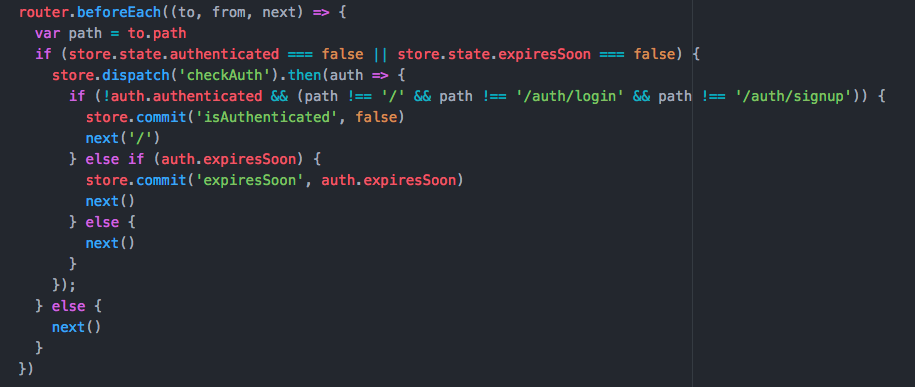


Finally there is a logout action that makes a request to the server to clear the users’s cookie from the browser. It then updates the store to tell the application that the user is no longer authenticated.



# Route Protections

In the **src » plugins » auth.js** file there is a global beforeEach function that is applied to all routes defined on the router. This function is responsible for pinging the server to check if the user is authenticated and allowed to access a certain Vue route. The function looks like this,



This function will get called before ever page change. First the path that the user is trying to access is saved to the path variable. Then the Vuex store is checked to see if the user is not authenticated or to see that if they are authenticated that their token will not expire soon. The first check is necessary because if the application is reloaded or a URL is manually entered and visited then the application’s state is wiped clean. The second check is necessary for checking is a user’s token will expire because if it will then the application will have a modal pop up to allow the user to enter their credentials and extend the session. If one of these conditions are true then the server must be pinged to check the authentication and token expiration. If not then the user is allowed to proceed to their intended route.

After the request is made there is a check of the user’s authentication based on the results of the server. If the user is not authenticated and they are trying to access a Vue route other than the home, login, or signup routes then they will be redirected to the home page. A Vuex commit is called just to ensure that the state reflects that the user is not authenticated. If they are authenticated but their token will expire soon then a Vuex commit is called to tell the application to display the modal to allow the user to extend their session. If none of those conditions are met then the user is allowed to proceed to their intended Vue route.

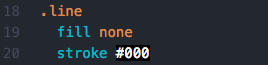
The final detail about the front end authentication is the SessionExtender component. This component is located in the main App.vue file and is rendered only when the user’s session will expire soon. It essentially is a popup login screen to allow the user to supply their username and password again to allow them to continue their session on the application.

# JSCatalyst Styling

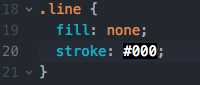
The majority of the styling for JSCatalyst is located in **src » assets » stylus** directory. While each Vue single file component has a <style> section at the bottom they are rarely used. The only components that do utilize these <style> sections are some of the visualizations that require styling for different theming. These were purposefully left in the component so that it would be clear to developers and designers how each component is getting styled.

# Stylus

JSCatalyst uses Stylus as its CSS preprocessor. Stylus is like Sass or Less but is much cleaner, makes writing complex selectors easy, and allows for looping and conditionals right in the Stylus code. Stylus files should be placed in the stylus directories or subdirectories within JSCatalyst. These files should be saved as with a .styl file extension so that the CSS loader will know how to parse the Stylus files. Important to note is that Stylus is reliant on whitespaces or indentations as syntactic sugar to make declaring selectors simple and easy. An example of this is



Here instead of having to use curly braces we are using indentations to declare the font property on the svg element. Also notice how we do not need to use colons ore semicolons on our property declarations. Stylus is smart enough that it knows the first part of the declaration will be the property and after the whitespace is the value. It also knows that each new line is another property being declared. It is still possible to use regular CSS syntax in Stylus files if you want. The above example would look like this



Both declarations would be interpreted in the same exact way.

# Common Parts of JSCatalyst Styles

Common things you might see in the Stylus file are Stylus variables and CSS variables.

# Stylus Variables

Stylus variables are used to reference values throughout your file. They are usually prefaced with a ‘$’ but can also be created without it. Here is an example of declaring a Stylus variable with or without the ‘$’



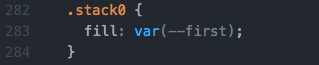
These variables can now be used throughout the Stylus file. These variables can only be referenced within Stylus files, and once the application is running they cannot be referenced within the application because they are processed before launching the application.

# CSS Variables

CSS variables are relatively new and at the time of writing cover 88% of browsers and are supported by all major browsers except for IE. JSCatalyst uses CSS variables in order to reference theme colors throughout the application. While these variables are declared within the Stylus files, once they are processed they can still be referenced within the application. CSS variables are always prefaced with two dashes (‘--‘). This is necessary for CSS to interpret the variable as a custom CSS property. Because CSS variables are custom properties, they can be named whatever you like (ex. ‘--myCustomVar’). Here is an example of CSS variables in our Stylus file



Now in one of our Vue files we can use these CSS variables in the <style> tag. This can be done like so



This is an example taken from the StackedBarChart.vue visualization which can be found at **src » components » visualizations » d3 » StackedBarChart.vue**. This is within the <style> tag of our component and is selecting the first stack of the bar chart. As you can see, in order to use the CSS variable you must wrap the variable in a var(). This is special syntax used by CSS to interpret the variable so it can get its value.

The reason this is useful is because we have multiple themes and each one has the same variables declared on them. Each one has a first variable so when the theme changes, the variable references a new value. When the blue-theme class is applied the --first variable value is #081A4E. When the pink-theme class is applied the --first variable value is #351850. This allows for instant changes in color themes throughout the visualizations using different classes.

Some visualizations cannot be styled using classes and require an array of colors to be passed in with its data. This includes all Chartjs, Chartist, and Plotly charts and a few D3 charts. Vuex is used in order for these charts to get the current theme colors. In the Vuex state there is an array named themeColors which holds all the colors for the current color theme. This array is accessible to all components in the project, and is updated each time the color theme is changed. By having this array, designers and developers have options in how they would like to style new charts. Either through a class system using the CSS variables, or through the themeColor array by directly defining what colors in the array a component should use.

# Adding new themes

In order to add new themes all that is needed is to add new classes in the Stylus \_themes.styl file. All new theme classes should be named in the (color)-theme format (ex. blue-theme, yellow-theme). Each new class should have 8 variables and 8 corresponding hex-code color values. The variables should be named first through eighth and the color spectrum should go from darkest color to lightest color. Once the new theme is added, the name of the theme should be added to the Vuex store so that it is available in the theme picker in the MainToolbar.vue.

# JSCatalyst and CSS Grid

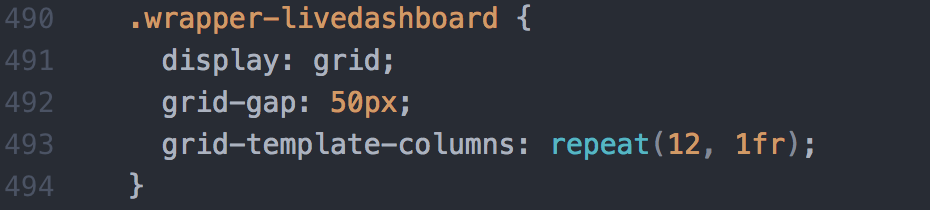
While building out JSCatalyst we realized how difficult and complex it is to effectively layout pages using styling libraries such as Bootstrap or Vuetify. With these styling libraries it’s hard to grasp the necessary nesting of container, layout, and flex components. In addition to this, these libraries create unnecessary complexity in the DOM with all these nested divs. The point of JSCatalyst is to make it accessible to developers and designers alike. This is why we chose CSS Grid for our application layout.

# CSS Grid Basics

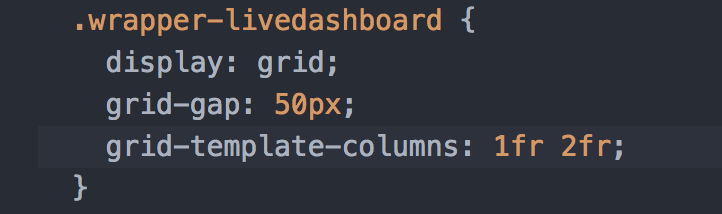
CSS grid is exactly what it sounds like, a grid. This is a step above flexbox, which only works vertically or horizontally. To make a new page with CSS Grid all you have to do is set the display property on your outermost div to grid. The convention that we have adopted is to give the div a class of ‘wrapper-YourPageName’. Then in the <style> tag of your page create a selector for the class and set display: grid; Here’s an example from our LiveDashboard.vue page,



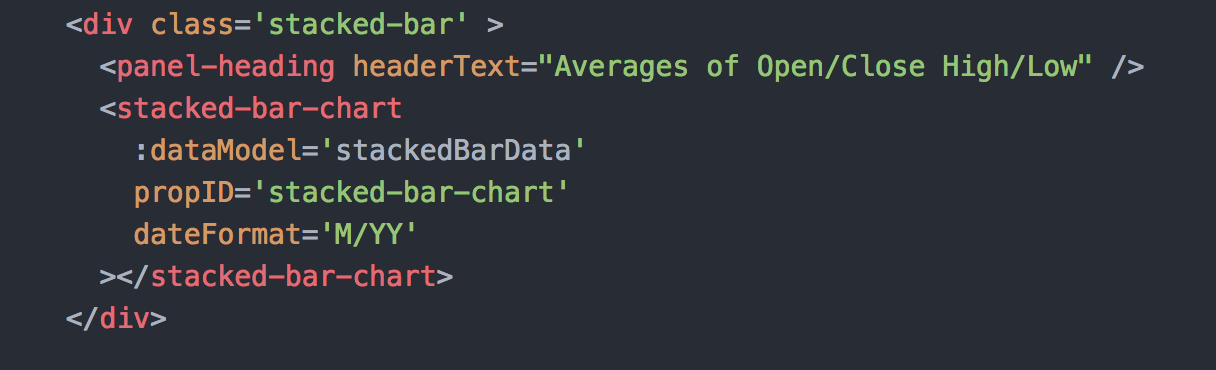
And in the <style> tag of the file,



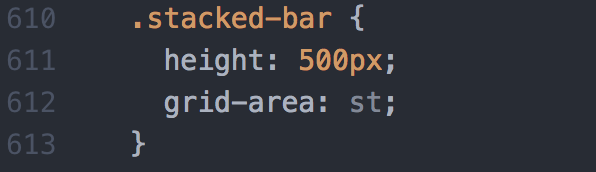
Lets break down the CSS attributes being set on the wrapper. The display: grid; property is necessary to use CSS Grid because this sets up the div to be split into rows and columns. The grid-gap: 50px; is used to create margins between all rows and columns in the grid. The grid-template-columns: repeat(12, 1fr); tells our grid that we want it to be split into 12 equal fractions (fr). The fr unit is new for CSS and just stands for fractions. So for example if we wanted two columns on our page but wanted the second to be twice as big as the first we would do this,



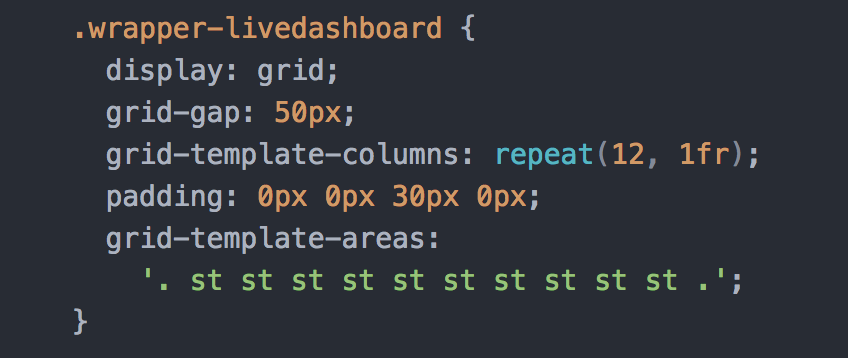
Once we have our grid set up each chart that we add to the page will be placed in its own div which will be given the class corresponding to the name of the chart.



This is the stacked bar chart from the live dashboard. Notice how we only have on div surrounding the component while when using a styling library we would need three or more to get the proper layout. Now that we have the component in its own div we can go to the <style> tag in the file and style this div.



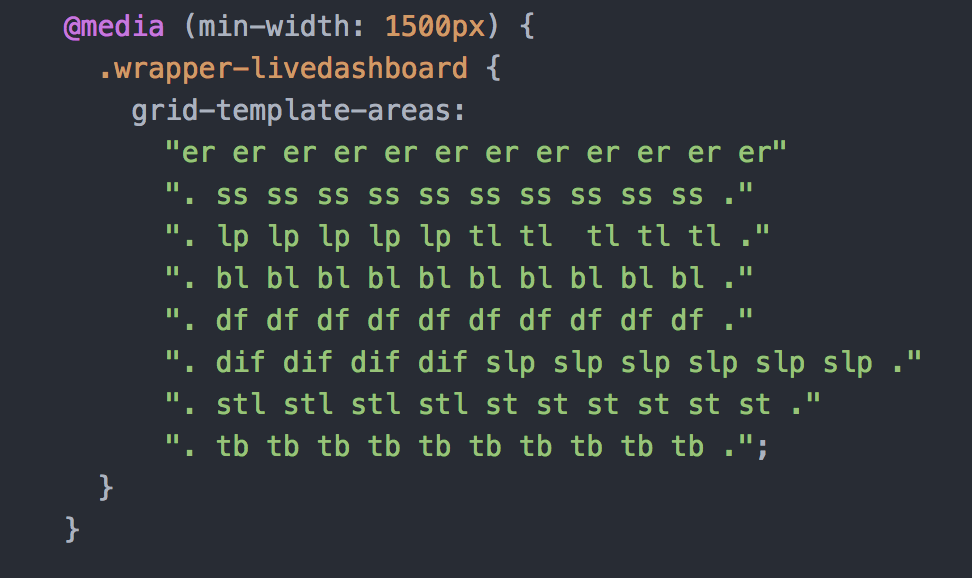
We give the div a height of 500 pixels because all the charts receive their height and width from the container they are in. The grid-area is how we will refer to this div in our CSS Grid wrapper. It is almost like setting this div to a variable so that we can access the div in our wrapper and define where we want this chart to be in our grid. Let’s see how that would be done,



Now in our wrapper we define the grid-template-areas attribute and we use our st grid area from our stacked bar chart. This attribute on the grid allows you to control exactly where the chart is rendered on the page. As you can see there should be one row on our page and it is split into the 12 equal fractions. A period (‘.’) indicates that the space should be left blank allowing you to easily put in white space. The ten st in a row tell the grid that we want our stacked bar chart to be exactly in those columns on our page. Now if we do that for all the other charts on our LiveDashboard.vue page the finished CSS would look like this,

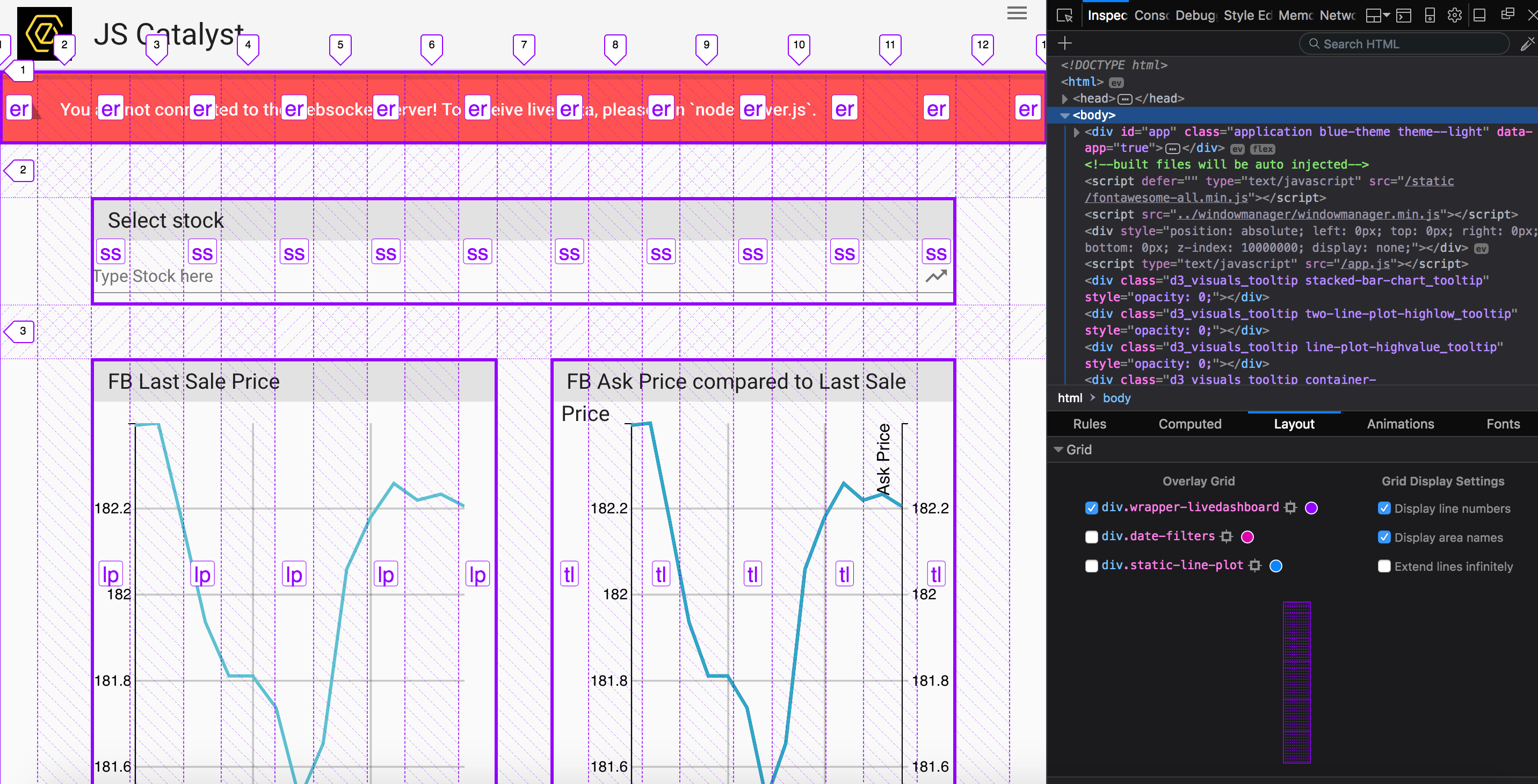


Here we are just defining how the page should look with every chart. This is the base view we want with one chart after another. Howerver we can add media queries to detect the screen size of the user and if it is bigger, such as a monitor, then we can change the layout. To do this all that is needed is to create another set of CSS rules for our wrapper within an media query. It would look something like this,



First we create the media query within the <style> tag of the LiveDashbaord.vue file. And if the window is wider thatn 1500 pixels, the application will use this grid-template-areas to layout all of the charts. Notice that because we have more width to the window we can now put charts next to each other. The whole grid is responsive too. As the window is resized, the charts resize because we used the fraction units. And once the page is resized to smaller than 1500 pixels, the grid will instantly update and change the layout to one chart after the other like we declared above.

When working with CSS Grid we recommend using [Firefox Developer Edition](https://www.mozilla.org/en-US/firefox/developer/). This is because Firefox has the most robust and easy to understand CSS Grid Dev tools. For example if we go to the Live Dashboard page in the running application and open the dev tools we can see all the named grid areas on our page,



The whole page is marked up with the columns and rows, and you can see each grid-area that we defined is clearly labeled.

# Things to keep in mind

There are a few things that you should keep in mind when creating a new CSS Grid layout. For example we recommend sticking to a grid that is split into 12 equal fractions. This enables you to do complex layouts easily, and makes sure that all of the components on the page will be responsive.

Another fact to consider is that OpenFin, at the time of writing, does not support CSS Grid. They use the version of Chrome that was released right before Chrome began supporting CSS Grid. Hopefully in the near future OpenFin will get onboard and upgrade, but for now it is necessary to have fall back styles for your pages.

Further reading on CSS Grid: <https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Grid_Layout>

# This page is intentionally left blank