

# Building chart dashlets and data visualizations using Sucrose



An UnCon 2017 tutorial

Henry Rogers, [hrogers@sugarcrm.com](mailto:hrogers@sugarcrm.com), [@hhrogersii](https://twitter.com/hhrogersii)

Sushma Sheshadri, [ssheshadri@sugarcrm.com](mailto:ssheshadri@sugarcrm.com)

Sugar v7.9.1.0

Get a digital copy of this tutorial at [http://bit.ly/tutorial\\_dashlets](http://bit.ly/tutorial_dashlets)

# Table of contents

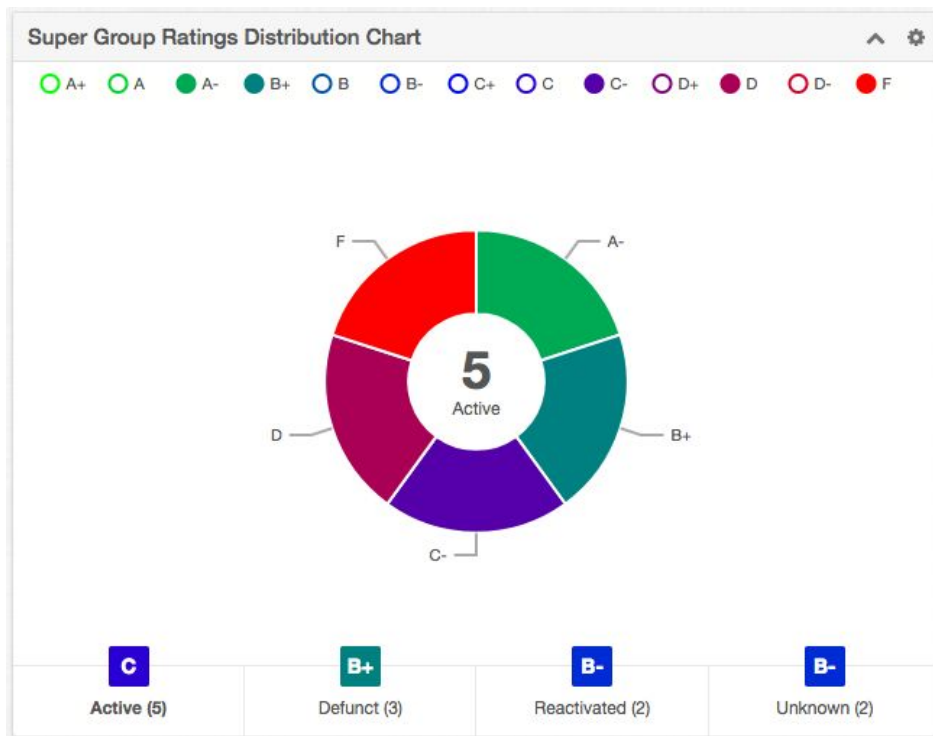
<b>Table of contents</b>	<b>2</b>
<b>Introduction</b>	<b>4</b>
What you'll learn	4
Before you begin	5
<b>Basics / Background</b>	<b>5</b>
Charting Library	5
Building Chart Dashlets-Simple 3-Step Process	5
Define the "data story" you want to tell	5
Identify existing data sources	5
Configure chart display options	5
Bring your visualization to life with user events	5
Step 1: Defining a data story	6
Step 2: Identifying data sources	6
Step 3: Implement chart dashlet	7
<b>Part 1 - Chart Dashlet Powered By Filter API</b>	<b>8</b>
Step 1.1: Data Story	8
Step 1.2: Identifying Data Source	8
Step 1.3: Implement Chart Dashlet	10
Goals for this step:	10
Step 1.3.1: create an HBS file under /my-ratings-chart called my-ratings-chart.hbs	11
Step 1.3.2: create a PHP file under /my-ratings-chart called my-ratings-chart.php	11
Step 1.3.3: create a JS file under /my-ratings-chart called my-ratings-chart.js	12
Step 1.3.4: define the chart model in method initialize	13
Step 1.3.5: get Super Group records from the Filter API in method loadData	14
Step 1.3.6: transform the server data in the function evaluateResult	14
Step 1.3.7: render the chart in method renderChart	17
Step 1.3.8: add My Ratings Distribution Chart dashlet to home dashboard	17
<b>Part 2: Switch to using Report API</b>	<b>18</b>
Step 2.1: Data Story	18
Step 2.2: Identify Data Source	18
Step 2.3: Implement chart dashlet	19
Goals for this step:	19

Step 2.3.1: Change the PHP config file to enable dashlet on Super Group module pages	19
Step 2.3.2: Add dashlet configuration panel options to the PHP config file	20
Step 2.3.3: Set state variables in new JS controller method initDashlet	21
Step 2.3.4: Update state variables when options change in new method bindDataChange	21
Step 2.3.5: Modify the loadData method to get data from the Report API	22
Step 2.3.6: Modify the evaluateResult function to process Report data	23
<b>Part 3: Add interactive aggregation tabs</b>	<b>25</b>
Step 3.1: Data Story	25
Step 3.2: Identify Data Source	26
Step 3.3: Implement Chart Dashlet	26
Goals for this step:	26
Step 3.3.1: Modify the dashlet layout template with new tab elements	26
Step 3.3.2: Create a new event assignment for tab clicks in the JS controller	27
Step 3.3.3: Add the function changeStatus to the JS controller	27
Step 3.3.4: Add the function buildStatusCollection to the JS controller	27
Step 3.3.5: Add the method rollupStatusAverages to the JS controller	30
<b>Part 4: Final formatting and finishing</b>	<b>32</b>
Step 4.1: Data Story	32
Step 4.2: Identify Data Source	32
Step 4.3: Implement Chart Dashlet	32
Goals for this step:	32
Step 4.3.1: Add tooltips to status average tabs	32
Step 4.3.2: Update CSS and HBS template to support accessibility	33
Step 4.3.3: Provide a custom formatter for the donut hole text	34
<b>Summary</b>	<b>36</b>
<b>Extra credit</b>	<b>36</b>
<b>Share your success</b>	<b>36</b>
<b>Additional resources</b>	<b>36</b>
<b>Need help?</b>	<b>36</b>

# Introduction

Building a quality data visualization dashlet involves more than just code. This tutorial will guide you through the process of developing a chart dashlet from planning to execution.

The final result of this tutorial will be an interactive chart dashlet:



## What you'll learn

After completing this tutorial, you will know how to...

- Build a custom basic chart dashlet from scratch
- Extend the basic chart dashlet to include aggregate data
- Configure dashlet options
- Transition existing 7.9 chart library (NVD3) based dashlets to Sucrose

At the end of this tutorial you will have the tools to unlock the meaning behind the data stored in your Sugar instance.

## Before you begin

Before we kick things off, make sure you have done each of the following:

- Have downloaded and installed a code editor and [Postman](https://www.getpostman.com/)<sup>1</sup>.
- Have a locally running version of Sugar 7.9.1.0 with no demo data and with the superhero sample data and modules installed (see [instructions here](#)<sup>2</sup>).
- Have downloaded and installed the Loadable Package `sugarcrm-uncon2017-chart-dashlets-2.x.zip` from the Uncon 2017 github repo [here](#)<sup>3</sup>. Instructions for how to install the package are in the [readme](#)<sup>4</sup>.
- Set up your browser to disable the cache so your changes will show up on reload

Ready, set, code! Actually, by doing a little planning at the beginning, we will reduce the amount of time spent reworking code.

## Basics / Background

### Charting Library

Sugar 7.9.x uses NVD3 as the charting library. Starting in Sugar 7.10, we are introducing a new open source charting library called [Sucrose](#)<sup>5</sup>, which is our customized and enhanced version of NVD3. The basic structure and configuration is the same, however new capabilities have been introduced with plans for further enhancements in future Sugar releases. In this tutorial we will provide instructions for transitioning to Sucrose.

### Building Chart Dashlets-Simple 3-Step Process

- Define the “data story” you want to tell
- Identify existing data sources
- Implement chart dashlet
  - Build a basic chart dashlet
  - Configure chart display options
  - Bring your visualization to life with user events

---

<sup>1</sup> <https://www.getpostman.com/>

<sup>2</sup> <https://github.com/sugarcrm/uncon/tree/2017/ProfessorM>

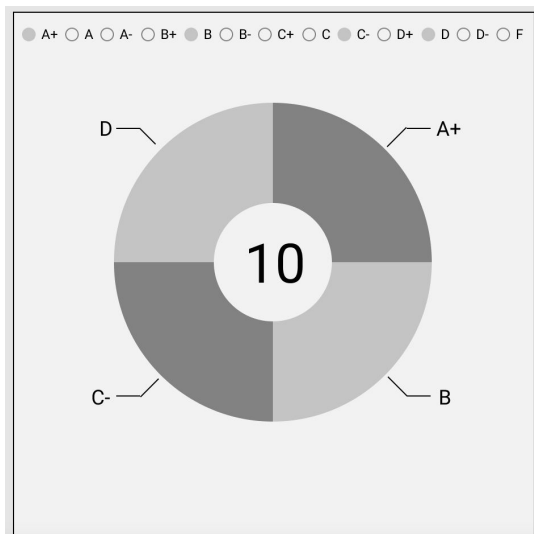
<sup>3</sup> <https://github.com/sugarcrm/uncon/tree/2017/chart-dashlets/releases>

<sup>4</sup> <https://github.com/sugarcrm/uncon/tree/2017/chart-dashlets>

<sup>5</sup> <http://sucrose.io>

## Step 1: Defining a data story

First and foremost, it is essential to have a reasonably firm data story in mind. Once you have your data story firmed up along with a rough picture of how you would want to visualize your data on the chart, you can move on to the next step to choose the data source for your story. Sometimes it is helpful to have a napkin sketch or low resolution composition to work from but it isn't always necessary. For this tutorial I've created a comp using [Figma](#)<sup>6</sup>:



## Step 2: Identifying data sources

The next step in our development process is to identify all available data sources that might help us construct the data needed by the chart. Out-of-the-box (OOTB) Sugar provides a number of useful data sources including REST-based resources such as the

- Filter API - Part 1 in the next section of this tutorial will take you through the steps for creating a chart dashlet using Filter API
- Report API - Part 2 of this tutorial will you through the steps for building a chart dashlet using Reports API.
- Module specific APIs - Not covered by the tutorial. You will get extra credit if you can build your custom dashlet using Opportunity or Forecast APIs :-)

If you don't think the existing APIs provide you with what you are looking for, you can write your own endpoint and use it as a data source for your dashlet.

---

<sup>6</sup> <http://www.figma.com>

## Step 3: Implement chart dashlet

By the time you reach this step, you have a clear idea of what you want to build (data story) and how you want to build (data source). Now that you are familiar with the basic steps for building a chart dashlet, let's jump in and start building one.

Lets pause here for a minute and get an overview of what you are going to learn in the remaining sections of this tutorial.

Part 1 will take you through the steps for creating a chart dashlet using filter API

Part 2 will walk you through the steps for creating a chart dashlet using reports API

Part 3 will teach you how to go beyond the basic chart dashlet and also make it more interactive

Part 4 will show you how to improve the usability and visual appeal of your dashlet

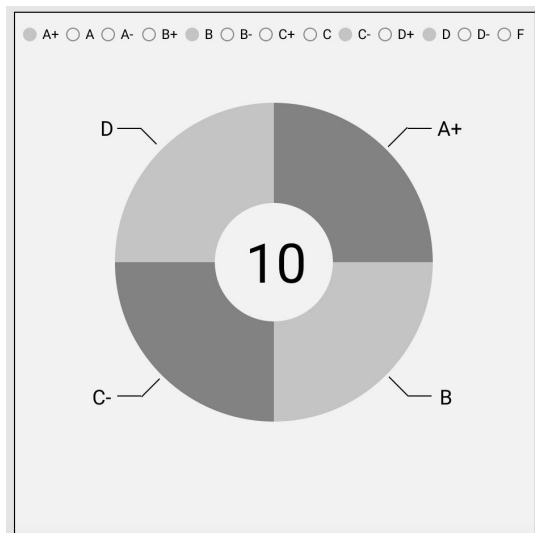
**Note:** The screenshots in this tutorial may not look like yours. Depending on how you may have modified the data in previous tutorials, the data in your instance may be different. The main thing to verify is that the elements are all there.

**Note:** Because the code changes are being made in different places within the files, please refer to the example code as needed for each of the following steps to confirm your changes are correct.

# Part 1 - Chart Dashlet Powered By Filter API

## Step 1.1: Data Story

We'll be building a visualization of the Super Group module. Each Super Group has a rating (A+ through F) and a status. For our tutorial we will be visually answering the question, *"What is the distribution of ratings for the top 10 Super Groups sorted by their rating?"* For each step in this tutorial we should be asking ourselves if the change helps achieve the data story. If we come across an improvement during development that adds value, it is fine to update or revise the data story. Just keep it in mind that clarity and ease of understanding is an important feature. Sometimes it is helpful to have a napkin sketch or low resolution composition to work from but it isn't always necessary. For this tutorial I've created a comp using [Figma](https://www.figma.com)<sup>7</sup>:



Let's use a pie chart to display the relative count for each rating group represented in the ten highest rated Super Groups. Each slice is labeled with the rating and the legend has the entire list of ratings. The legend keys that have at least one Super Group are filled in.

## Step 1.2: Identifying Data Source

*So what data do we need?* Starting with what we want to end up with, here is a typical data structure expected by the chart library pie chart model:

```
{
  "properties": {
    "title": "Rating distribution for Top 10 Super Groups sorted by rating"
  },
  "data": [
    {"key": "A+", "value": 0, "color": "green"},
  ]
}
```

<sup>7</sup> <http://www.figma.com>



```

    {"key": "A", "value": 4, "color": "aquamarine"},
    ...
    {"key": "D-", "value": 2, "color": "purple"}
    {"key": "F", "value": 1, "color": "red"}
  ]
}

```

Common data structure examples for our chart engine are available in

</sugarcrm-root/styleguide/content/charts/data>

For a distribution chart like a pie chart, it is important to display all the available options in the legend, which is why some of the data array ratings have an initial value of 0. We are also going to set the color of the pie chart slices in the data array.

So to start, we need an array of ratings, which we can get from the collection of Sugar app string arrays (see the [dropdown editor in Studio](#)<sup>8</sup> for a list of all available key value arrays, or create your own):

```

'grading_list': {
  "A Plus": "A+",
  "A": "A",
  "A Minus": "A-",
  "B Plus": "B+",
  "B": "B",
  "B Minus": "B-",
  "C Plus": "C+",
  "C": "C",
  "C Minus": "C-",
  "D Plus": "D+",
  "D": "D",
  "D Minus": "D-",
  "F": "F"
}

```

Next, we need a value for each rating. For part 1 of this tutorial, we'll get a collection of Super Group records from our Sugar instance using the Filter API. This is the type of data structure we will get back from the Filter API (simplified for clarity):

```

records: [
  {
    "id": "eb50ab54-89bf-11e7-82d8-a45e60e68e7b",
    "date_modified": "2017-08-25T11:05:03-07:00",
    "ratesg_c": "A Minus",
    "status_c": "Active"
    "_module": "Accounts"
  },

```

---

<sup>8</sup>[http://support.sugarcrm.com/Documentation/Sugar\\_Developer/Sugar\\_Developer\\_Guide\\_7.9/Architecture/Languages/Managing\\_Lists/index.html](http://support.sugarcrm.com/Documentation/Sugar_Developer/Sugar_Developer_Guide_7.9/Architecture/Languages/Managing_Lists/index.html)

```

...
{
  "id": "ed579c78-89bf-11e7-a1cf-a45e60e68e7b",
  "date_modified": "2017-08-25T11:05:06-07:00",
  "ratesg_c": "B",
  "status_c": "Reactivated",
  "_module": "Accounts"
}
]

```

We now have all the data necessary to construct our chart and a clear picture of what we want to end up with.

Let's code!

## Step 1.3: Implement Chart Dashlet

Goals for this step:

- Create an .hbs Handlebars template with chart container
- Create a .php config file to assemble components
- Create a .js view controller with methods:
  - initialize: chart model
  - renderChart: boilerplate
  - loadData: Filter API Top 10 SuperGroups

After you install the tutorial Loadable Package, navigate to the folder `/sugarcrm-root/custom/clients/base/views` in your IDE. Inside this folder you will see a prebuilt dashlet for each of the four parts of the tutorial under: `/ratings-chart-1`, `/ratings-chart-2`, `/ratings-chart-3`, and `/ratings-chart-4`. Feel free to refer to them as we go along or copy and paste the code if you wish.

First, lets create a new folder under `/sugarcrm-root/custom/clients/base/views` called `/my-ratings-chart`

Each custom chart dashlet typically has three components:

1. .hbs Handlebars template: defines the HTML structure of our dashlet
2. .php configuration file: defines our dashlet components
3. .js view controller: extracts our data from the server, transforms it into a format expected by our chart engine, and then loads it into the Handlebars template using the chart rendering functions.

### Step 1.3.1: create an HBS file under `/my-ratings-chart` called `my-ratings-chart.hbs`

```
<div data-content="chart">
  <div class="nv-chart nv-chart-donut">
    <svg id="{{cid}}"></svg>
  </div>
</div>
<div class="block-footer hide" data-content="nodata">{{str "LBL_NO_DATA_AVAILABLE"}}</div>
```

You can change the chart class (in this case the class is `nv-chart-donut`) depending on the chart type you want. The `{{cid}}` id is a unique id created for each view by the Sugar app. We will be targeting this SVG element as the container for our chart. The `div.block-footer` is hidden initially but will be displayed if the data returning from the server is empty.

**Sucrose Changes:** the chart CSS classes have changed in Sucrose. Now the namespace starts with “sc-” instead of “nv-” so the div above would have `class="sc-chart sc-chart-donut"`.

### Step 1.3.2: create a PHP file under `/my-ratings-chart` called `my-ratings-chart.php`

```
<?php
$viewdefs['base']['view']['my-ratings-chart'] = array(
  'dashlets' => array(
    array(
      'label' => 'My Ratings Distribution Chart',
      'description' => 'Uses the Filter Api as datasource',
      'filter' => array(
        'module' => array(
          'Home',
        ),
        'view' => array(
          'record',
        )
      ),
      'config' => array(),
      'preview' => array(),
    ),
  ),
);
```

The dashlets array describes where the dashlet can be viewed. For this chart tutorial, we'll start with the home dashboard only. See the Sugar [documentation](http://support.sugarcrm.com/Documentation/Sugar_Developer/Sugar_Developer_Guide_7.9/User_Interface/Dashlets/index.html)<sup>9</sup> for details on setting up the dashlet config and preview options.

---

9

[http://support.sugarcrm.com/Documentation/Sugar\\_Developer/Sugar\\_Developer\\_Guide\\_7.9/User\\_Interface/Dashlets/index.html](http://support.sugarcrm.com/Documentation/Sugar_Developer/Sugar_Developer_Guide_7.9/User_Interface/Dashlets/index.html)

Step 1.3.3: create a JS file under `/my-ratings-chart` called `my-ratings-chart.js`

```
/**
 * @class View.Views.Base.MyRatingsChartView
 * @alias SUGAR.App.view.views.BaseMyRatingsChartView
 * @extends View.View
 */
({
  plugins: ['Dashlet', 'Chart'],
  className: 'my-ratings-chart-wrapper',
  initialize: function(options) {
    ...
  },
  loadData: function(options) {
    ...
  },
  evaluateResult: function(serverData) {
    ...
  },
  renderChart: function() {
    ...
  }
})
```

We need to define two plugins: the Dashlet plugin and the Chart plugin. The Dashlet plugin sets up all of the common dashlet boilerplate like the config menu in the header bar. The Chart plugin adds a number of default variables and functions like responsive window sizing and event listeners. The `className` property can be any name but is generally unique for each chart dashlet type in case any common css properties need to be adjusted. The four functions in the code snippet above are typically all that are needed for a chart dashlet. We'll code each of the functions separately below. See the cases summary, bubble chart, and opportunity metrics under `/sugarcrm-root/clients/base/views` for similar examples of basic chart dashlet controllers.

### Step 1.3.4: define the chart model in method `initialize`

```
initialize: function(options) {
    this._super('initialize', [options]);

    this.chartData = [];

    this.chart = nv.models.pieChart()
        .margin({top: 0, right: 0, bottom: 0, left: 0})
        .donut(true)
        .maxRadius(120)
        .colorData('data')
        .direction(app.lang.direction)
        .tooltipContent(function(key, x, y, e, graph) {
            return '<p><b>Grade: ' + key +
                '</b><br>Count: ' + parseInt(y, 10) + '</b></p>';
        })
        .strings({
            noData: app.lang.get('LBL_CHART_NO_DATA')
        });
},
```

**Sucrose Changes:** the chart instance names are the same in Sucrose but the location has changed. For a pie chart you would call `sc.chart.pieChart()` to get an instance. The rest of the methods are the same.

In the initialize function we are defining two `this` scope variables: the `chartData` that will be populated later, and the `chart`. Each of our chart models is exposed under `nv.models` as `pieChart()`, `multiBarChart()`, `lineChart()`, etc (see the note above for Sucrose chart models). In the code snippet above I removed some of the options with default values for clarity. The following is a list of the required options:

- `margin`: the default is 10px for each side (as an integer), but we will want to override this for our dashlet
- `donut`: this options sets up the `pieChart()` instance as a donut chart
- `maxRadius`: we are setting a max radius in pixels (as an integer) for the tutorial so our charts all have the same radius on a widescreen monitor which typically looks better when multiple pie charts are on a dashboard (on a small screen device the pie radius is allowed to get smaller so that it will fit a smaller dashlet size)
- `colorData`: for most charts we can set this to 'default,' but we'll be defining the pie chart colors mapped to each rating in the distribution
- `direction`: Sugar charts are bidirectional (i.e., respects Right-To-Left language layout) so we want to pass in the current app direction
- `tooltipContent`: this options accepts a type of function called a custom formatter that you can use to return any HTML DOM string using the provided parameters
- `strings`: use this option to pass in translated strings expected by the chart type

### Step 1.3.5: get Super Group records from the Filter API in method

#### loadData

```
loadData: function(options) {
  if (this.meta.config) {
    return;
  }
  var url = app.api.buildURL('Accounts', 'filter');
  var api_args = {
    'fields': 'id,status_c,ratesg_c',
    'order_by': 'ratesg_c',
    'max_num': 10
  };
  var options = {
    success: _.bind(function(serverData) {
      this.evaluateResult(serverData);
    }, this),
    complete: options ? options.complete : null
  };
  app.api.call('create', url, api_args, options, {context: this});
},
```

In the loadData function we will be making a call to Filter API to get a collection of Super Group records. The Super Group (Accounts) module has been modified to include the `ratesg_c` field that can be updated by editing a Super Group record.

Before making the REST call, first check to see if the dashlet config panel is being viewed (the existence of `meta.config`) and, if so, stop execution of the dashlet load data/render view methods. Next, we make a POST request to the Filter API endpoint with a URL constructed using an API utility `buildURL` and a set of API arguments to pass in the POST body. The three fields we need are `id`, `status_c`, and `ratesg_c`, sorted by `ratesg_c`. We only want the top 10 Super Groups (sorted by their rating), so we set `max_num` to 10. When the server returns data on success, we'll process it with the `evaluateResult()` callback function. The complete callback is generally used to cancel the spinner when the Refresh option is picked in the dashlet config dropdown menu.

### Step 1.3.6: transform the server data in the function `evaluateResult`

Because all our charts are data driven documents (i.e., D3), it is critical that our data is properly formatted. This is where things get interesting and the code can get a bit messy, so I'll step through this function a block at a time. See note above about using the example code as a reference.

Starting with the function signature:

```
evaluateResult: function(serverData) {
```

Looking back at our data plan, we know we will need the grade list so let's go ahead and add that. We'll add some defensive code around the data returned from the Filter API:

```
var gradeList = app.lang.getAppListStrings('grading_list');
var records = serverData && serverData.records ? serverData.records : [];
```

The `ratingsData` variable will store the data that will be rendered by the pie chart:

```
var ratingsData = [];
```

The `total` variable is important because it will collect the sum of all rating values:

```
var total = 0;
```

Users respond to color in unique and unexpected ways, so choose your color palette carefully. For our chart we want to go from 'green' for the A range to 'blue' for the C range to 'red' for the F range. To create this color spectrum, we turn to D3's most powerful feature: scales. This linear scale is similar to how gradients work in CSS. First we set the expected numeric domain (for our use we'll be going from 0 to the size of the `gradeList` minus one since this is zero based index). Then we set a hex values color stop for of our domain values.

The `d3.scale.linear()` scale is now able to interpolate any input numeric value between 0 and 12 into a color value:

```
var gradeSize = _.size(gradeList) - 1;
var colorRange = d3.scale.linear()
  .domain([0, gradeSize / 2, gradeSize])
  .range([d3.rgb('#0f0'), d3.rgb('#00f'), d3.rgb('#f00')]);
```

We know from our data plan that the `gradeList` is a set of key/value pairs in the format '`A plus`': '`A+`' and we want to turn it into an associative array of rating objects. Let's loop through it and set the initial `value` to 0 (we'll update this value when we loop through the records) and set a `color` property using our `colorRange` scale from above based on the current index:

```
var colorIndex = 0;
_.each(gradeList, function(grade, key, values) {
  gradeList[key] = {
    index: colorIndex,
    value: 0,
    key: grade,
    color: colorRange(colorIndex)
  };
  colorIndex += 1;
});
```

After processing the gradeList we end up with something like (abbreviated):

```
gradeList: {
  "A Plus": {
    "index": 0, "value": 0, "key": "A+", "color": "#00ff00"
  },
  ...
  "F": {
    "index": 12, "value": 0, "key": "F", "color": "#ff0000"
  }
}
```

Now that our gradeList data collector is set up, we can now loop through the Super Group records using the `ratesg_c` value (which contains a key from gradeList) and increment the associated rating option value by 1:

```
_.each(records, function(record) {
  gradeList[record.ratesg_c].value += 1;
});
```

Our data collector is now fully populated so let's extract the objects into an array:

```
_.each(gradeList, function(grade) {
  ratingsData.push(grade);
});
```

Finally, we'll calculate the total value that we defined as a `this` scope variable in the `initialize` method. The `total` variable is important because the Chart plugin relies on this variable to know if it should display the chart or show the "Data not available" message:

```
this.total = d3.sum(ratingsData, function(g) { return g.value; });
```

We now have everything we need to construct the data object expected by our chart library:

```
this.chartData = {
  data: ratingsData,
  properties: {
    title: app.lang.get('LBL_DASHLET_GRADES_CHART_NAME'),
    total: total
  }
};
```

And finally we can make a call to render the chart:

```
this.renderChart();
},
```



### Step 1.3.7: render the chart in method `renderChart`

```
renderChart: function() {
  if (!this.isChartReady()) {
    return;
  }

  // Set value of label inside donut chart
  this.chart.hole(this.total);

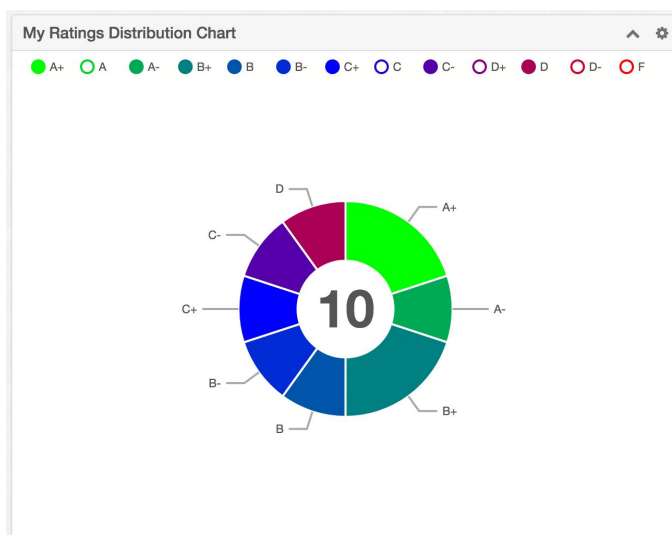
  d3.select(this.el).select('svg#' + this.cid)
    .datum(this.chartData)
    .call(this.chart);

  this.chart_loaded = _.isFunction(this.chart.update);
  this.displayNoData(!this.chart_loaded);
}
```

The `renderChart` function is pretty much the same across all dashlets. We first check if the chart model was instantiated properly using a utility function from the Chart plugin, then set any chart model data driven properties. After making a call for D3 to first select the current dashlet view DOM element and then sub-selecting the SVG container, we are able to bind the `chartData` to the selection and then call the chart instance which receives the data. If the chart is instantiated properly the instance will have a new `update` function property. If it does not, we display the “No data available” message in the dashlet using a Chart plugin utility function.

### Step 1.3.8: add My Ratings Distribution Chart dashlet to home dashboard

Now that all the code is written it's time to test your new dashlet! Create a new Home dashboard and add two new dashlets: yours (My Ratings Distribution Chart) and ours (Ratings Distribution Chart 1). If all goes well, you should see two dashlets matching this screenshot:



## Part 2: Switch to using Report API

### Step 2.1: Data Story

Looking back at the data story for this tutorial, *“What is the distribution of rating for the top 10 Super Groups sorted by ratings?”* and reviewing our dashlet, maybe we can identify some improvements. First question we might ask is why are we limiting the number of Super Group Records to just the Top 10? That doesn’t seem like a fair distribution since there are only 12 records. Maybe we would want to set the distribution across all Super Groups.

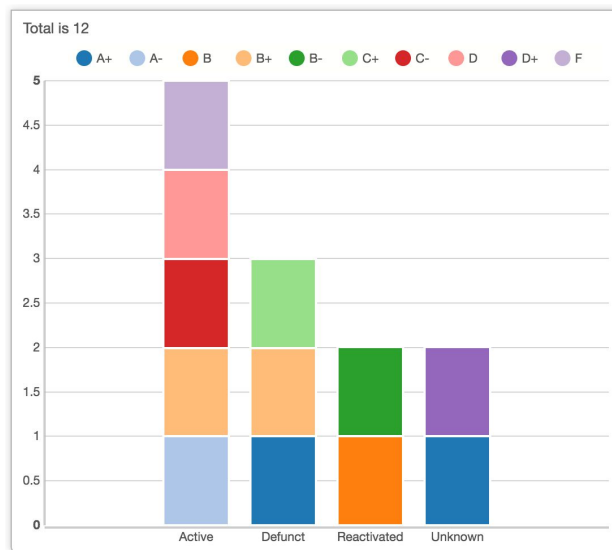
The next question that would be good to ask is since Super Groups have an additional property of status, do we want to see the grade distribution for just the active Super Groups? Let’s see how our data story could be rewritten to accommodate these changes. *“What is the rating distribution for all active Super Groups sorted by their rating?”* Now, let’s see what changes are needed to our dashlet to implement the new data story.

### Step 2.2: Identify Data Source

The Filter API is very good (and fast) for retrieving a specific set of records, but is designed for getting “chunks” of records one at a time. For calculating an aggregation of data like counts or sums, there is a better API that we can leverage using the Reports module. So first we’ll need a new Report. Follow the these steps to construct your source report:

1. Go to the Report module and create a new report
2. Select Summation Report in the Report Wizard
3. Select Accounts module (which is the module renamed as Super Groups)
4. Skip the “Define Filters” definition panel by clicking the “Next” button
5. In the “Define Group By” panel click the Account module and then type status in the “Search for field.” Pick the “Status” field in the lower list to move it to the panel
6. Also search and select the “Rating” field and add it to the panel
7. Select the “Next” button to go to the “Display Summaries” panel
8. Pick to add “Count” from the field list and then click “Next”
9. In the “Chart Options” panel choose the “Vertical Bar” chart type and then click “Next”
10. In the “Report Details” panel, name the report “Super Groups by Status by Rating” and then click the “Save and Run” button

If your report is created correctly you should see a chart like this:



If our visualization is a pie chart, why are we creating a bar chart? We could have added a filter to our report for status equal to “Active” and configure the report with a pie chart. However, because we’ll be reusing this report in later steps where we need all the status groups, the report should have the breakdown of rating distributions for all status groups. The multiple grouping capability of a multibar chart provides the data in the needed format. Before you move on, copy the Report id from the end of the URL to a scratch file. We’ll need it later.

## Step 2.3: Implement chart dashlet

Goals for this step:

- Add default status selector and input field for Report Id in dashlet config panel
- Change PHP config
  - Enable display of dashlet on Accounts list and record view
- Change JS controller:
  - Modify loadData & evaluateResult methods

### Step 2.3.1: Change the PHP config file to enable dashlet on Super Group module pages

Let’s make some other improvements to our dashlet by updating the dashlet title:

```
'description' => 'Uses the Report Api as datasource'
```

And by making the dashlet available on the Super Groups list and record detail pages in the right-hand side intelligence pane:

```

        'filter' => array(
            'module' => array(
                'Home',
                'Accounts',
            ),
            'view' => array(
                'record',
                'records',
            )
        ),
    ),

```

### Step 2.3.2: Add dashlet configuration panel options to the PHP config file

Now that we have the report built, we need a way to tell the dashlet to use the new report as our data source and what Super Group status to filter the report data on. We can do this by setting up two options in the dashlet configuration panel. Make the following modification to your PHP config file to set a default `status` and an empty `report_id` option in the `config` array:

```

'config' => array(
    'status' => 'Active',
    'report_id' => '',
),

```

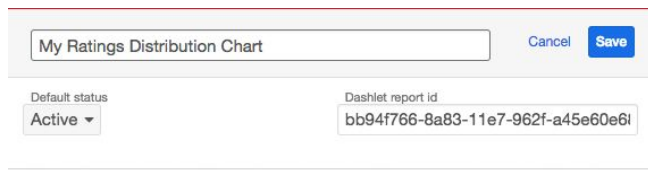
Next we need to link those settings to fields on the config panel itself so directly under the dashlets array in your `my-ratings-chart.php` file, add the following:

```

'panels' => array(
    array(
        'name' => 'panel_body',
        'columns' => 2,
        'labelsOnTop' => true,
        'placeholders' => true,
        'fields' => array(
            array(
                'name' => 'status',
                'label' => 'Default status',
                'default' => 'Active',
                'type' => 'enum',
                'options' => 'account_status_list',
                'enum_width' => 'auto',
            ),
            array(
                'name' => 'report_id',
                'label' => 'Dashlet report id',
                'default' => '',
                'type' => 'base',
            ),
        ),
    ),
),
),

```

Save your changes and reload your dashboard. If you select the “Edit” option under the dashlet config menu, you should now see two edit fields:



If you have the report id handy from Step 2.2 above, go ahead and paste it in now and click “Save.”

### Step 2.3.3: Set state variables in new JS controller method `initDashlet`

Once the dashlet config panel is updated, we now need to make the config options easier to use in the dashlet. Refer to the `ratings-chart-2.js` example code as needed for each of the following steps to confirm your changes are correct. For this step we will create a new method in the JS controller:

```
initDashlet: function() {
  this._super('initDashlet');
  if (this.meta.config) {
    return;
  }
  var status = this.settings.get('status');
  var reportId = this.settings.get('report_id');
  this.statusState = _.isEmpty(status) ? 'Active' : status;
  this.reportId = _.isEmpty(reportId) ? '' : reportId;
},
```

In this method we need to check if the dashlet config panel is being viewed and, if so, we exit the method. For the two settings, we first get them from the dashlet settings model and save them to the `this` scope for later use.

### Step 2.3.4: Update state variables when options change in new method `bindDataChange`

One additional change we need to make in the JS controller is to listen for changes to the settings made in the config panel and update the `this` scope variables with those changes. To do this we’ll create a new `bindDataChange` function to handle these events.

```
bindDataChange: function() {
  if (!this.meta.config) {
    return;
  }
  this.settings.on('change:status', function(model) {
    var status = this.model.get('status');
    this.statusState = _.isEmpty(status) ? 'Active' : status;
  });
}
```

```

    }, this);
    this.settings.on('change:report_id', function(model) {
        var reportId = this.model.get('report_id');
        this.reportId = _.isEmpty(reportId) ? '' : reportId;
    }, this);
},

```

### Step 2.3.5: Modify the `loadData` method to get data from the Report API

Now it is time to change our dashlet from getting data from the Filter API to getting data from the Report API. First add some defensive code to make sure the `reportId` has been set.

```

if (this.meta.config || _.isEmpty(this.reportId)) {
    return;
}

```

Using the API utility `buildURL` like before, we are passing the `this.reportId` as a url parameter in the REST call instead of in the body of the POST.

```

var url = app.api.buildURL('Reports/chart/' + this.reportId);

```

The `api_args` object should be also be modified.

```

var api_args = {
    'ignore_datacheck': true
};

```

The data that we get back from the Report API (simplified for clarity) is significantly different than the Filter API:

```

chartData: {
    "properties": [
        {
            "title": "Total is 12",
            "type": "stacked group by chart"
        }
    ],
    "label": ["A-", "B+", "C-", "D", "F", "A+", "C+", "B", "B-", "D+"],
    "values": [
        {
            "label": "Active",
            "values": [1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
        },
        ...
        {
            "label": "Unknown",
            "values": [0, 0, 0, 0, 0, 1, 0, 0, 0, 1]
        }
    ]
}

```

Referring back to our new Report, the Super Groups are grouped by Status and then Rating, so each object in the `values` array represents a data group and each rating in the `label` array represents a data series. This will be important when we process the data in the next step.

### Step 2.3.6: Modify the `evaluateResult` function to process Report data

See note above about using the example code as a reference while you work through this step. The first modification we need to make is to get `chartData` out of the `serverData` parameter:

```
var chartData = serverData && serverData.chartData ? serverData.chartData : [];
```

In the view controller for Part 1, we received and processed Super Group records with the `ratesg_c` field that stores the rating using the list key (e.g., “A Plus”, “A”, “A Minus”). When we change to the Report API, the data that we get back has the rating with the `ratesg_c` value (e.g., “A+”, “A”, “A-”). So to properly process the `serverData`, we will need to set up a new accumulator object.

```
var reportStatusValues = [];  
var gradeScale = {};
```

Now looping through the `gradeList` as before, we’ll instead be inserting objects into our `gradeScale` accumulator using the grade (label) as the key:

```
_.each(gradeList, function(grade, key, values) {  
  gradeScale[grade] = {  
    index: colorIndex,  
    value: 0,  
    key: grade,  
    color: colorRange(colorIndex)  
  };  
  colorIndex += 1;  
});
```

In the next block we’ll first check that the `chartData` object has a non-empty values array, then filter the `chartData.values` looking for only the data group (see above) that matches our current `statusState` “Active”. Now looping through the values of the “Active” data group we can update the `gradeScale` accumulator and total based on the group value. Finally, we build our `ratingsData` array from `gradeScale`.

```
if (chartData.values && chartData.values.length) {  
  reportStatusValues = chartData.values  
    .filter(_.bind(function(value) {  
      return value.label === this.statusState;  
    }, this));  
  
  if (reportStatusValues.length) {  
    reportStatusValues[0].values
```

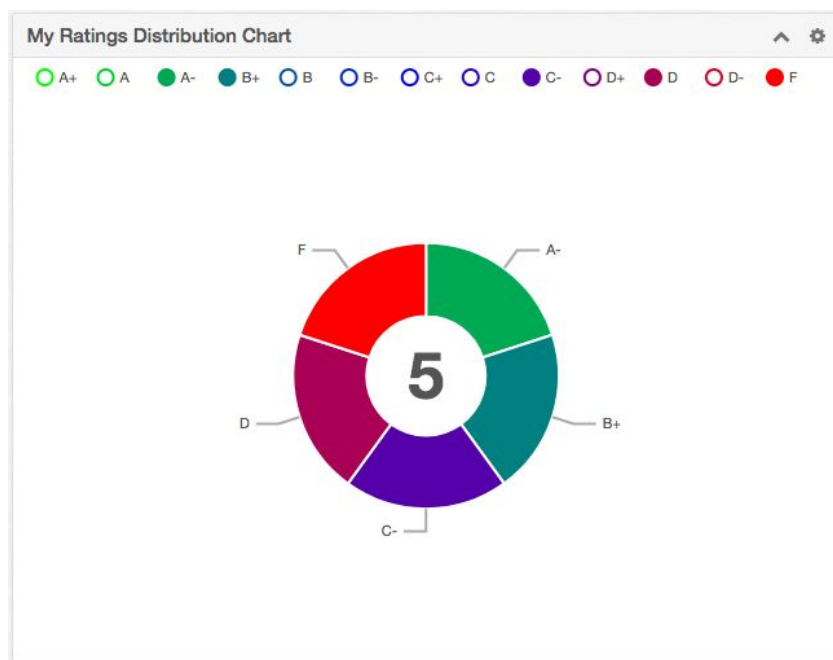
```

        .forEach(function(value, i) {
            var key = chartData.label[i];
            gradeScale[key].value = value;
            total += value;
        });

    _each(gradeScale, function(grade) {
        ratingsData.push(grade);
    });
}
}

```

Once you've made these changes, reload your browser. If all the changes are correct you should see the following dashlet:



The changes are not remarkably different from the first dashlet, but now we have a way of using the Report API to group and aggregate our data for us. Our dashlet is also updated so only the “Active” Super Groups are represented. So far, so good...but we can do more.

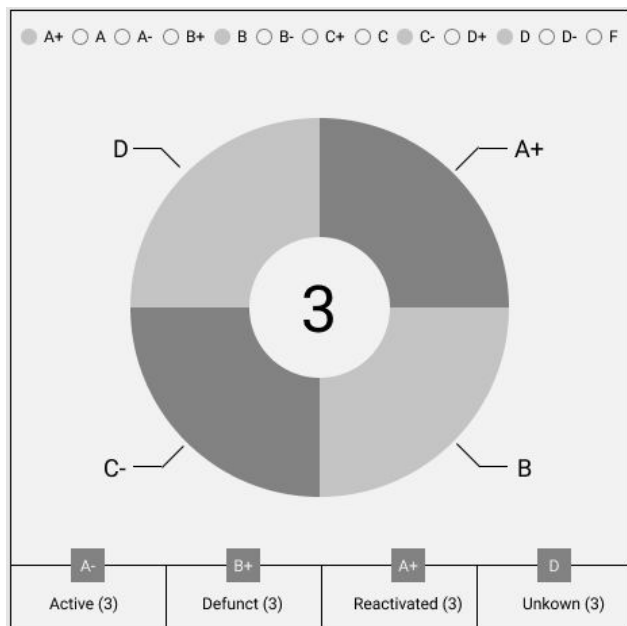


## Part 3: Add interactive aggregation tabs

### Step 3.1: Data Story

Like we did at the start of Part 2, let's review our data story and working dashlet looking for possible improvements. Currently we have *"What is the rating distribution for all active Super Groups sorted by their rating?"* What are the deficits with the current story? Well for one, we are limiting the user to only seeing the distribution for one Super Group status at a time. Also, if the user wants to see the distribution for another status, they have to go into the dashlet config panel and manually change the status.

Instead, let's try adding a set of tabs at the bottom of the dashlet for each Super Group status. The tab will show the average rating and the number of Super Groups in each status. We'll also make each of the tabs interactive so when a user clicks the status tab, the report data is filtered on the selected tab. So with these changes in mind our data story should be updated: *"As Professor M, I want to monitor the ratings of all my Super Groups by viewing the average rating for each status and the rating distribution and count for a selected Super Group sorted by rating."* Since we are making some changes to the layout, we should also update the mockup.



## Step 3.2: Identify Data Source

As in Part 2, let's continue to use Report API as our data source.

## Step 3.3: Implement Chart Dashlet

Goals for this step:

- Update HBS template to with tabs and styling
- Change Js controller:
  - `changeStatus`: listener for the tab click event
  - `rollupStatusAverages`: rollup of report data to create status averages
  - `buildStatusCollection`: create a status collection based on "account\_status\_list"

### Step 3.3.1: Modify the dashlet layout template with new tab elements

So, the chart layout changed. Where should we start? By modifying the HBS template to include our new dashlet elements, we can start to get a handle on what our new data structures should be. After the closing tag of the `<div class="nv-chart nv-chart-donut">...</div>` element, insert the following:

```
{{#eachOptions statusCollection}}
  <div class="opportunity-metric" style="width:{{../statusWidth}}">
    <span class="label" data-status="{{value.key}}"
style="background-color:{{value.color}}">{{value.grade}}</span>
    <div class="opportunity-metric-description">{{value.label}} ({{value.count}})</div>
  </div>
{{/eachOptions}}
```

All great designers steal so let's borrow the tab structure at the bottom of the Opportunity Metrics dashlet (available on an Account record view). This will give us some CSS styling OOTB to speed our development. The `eachOptions` Handlebars helper supplied by Sugar has the ability to loop through a status collection. The tabs will need a dynamically calculated CSS `statusWidth` attribute and each tab data object should have an HTML data attribute for status based on the `value.key`. We should set a `value.color` for the tab indicator background to match the average grade as well as the indicator label based on the `value.grade` property. Below the indicator we'll show the Super Group status `value.label` and `value.count`.

Because we are adding new HTML content to the bottom of a dashlet, we should reduce the height of the pieChart container so the content does not scroll. Because we enabled the dashlet on the home dashboard as well as the intelligence pane dashboard, we need both `.dashboard` and `.sidebar-content` CSS declarations below since sidebar dashlets are typically shorter:

```

<style>
.dashboard .thumbnail.dashlet:not(.collapsed) .my-ratings-chart-wrapper-3 .nv-chart {
    height: 401px;
}
.sidebar-content .thumbnail.dashlet:not(.collapsed) .my-ratings-chart-wrapper-4 .nv-chart {
    height: 301px;
}
</style>

```

**Sucrose Changes:** the chart CSS classes have changed in Sucrose. Now the namespace starts with “sc-” instead of “nv-” so the CSS above would use the .sc-chart selector.

### Step 3.3.2: Create a new event assignment for tab clicks in the JS controller

Creating new event assignments in Sugar is very easy. Define a top level events variable in the view controller as an object with event listeners as the key and event handlers as the value.

```

events: {
    'click [data-status]': 'changeStatus'
},

```

The event handler will fire each time a user clicks an element with a `data-status` attribute.

### Step 3.3.3: Add the function `changeStatus` to the JS controller

We'll also need to add the event handler method to our view controller:

```

changeStatus: function(evt) {
    var state = $(evt.currentTarget).data('status');
    if (state) {
        this.statusState = state;
        this.loadData();
    }
},

```

Using jQuery we get the `status` data attribute from the current event target. If it's defined we then set the `this` scope variable and reload the data.

### Step 3.3.4: Add the function `buildStatusCollection` to the JS controller

As before, since there is a lot going on here, we'll show the final function and then step through the code:

```

buildStatusCollection: function(chartData, gradeScale) {
    var statusList = app.lang.getAppListStrings('account_status_list');
    var rollup = this.rollupStatusAverages(chartData, gradeScale);

    // build the status collection from status list
    // with values from the chart data rollup
    var statusCollection = [];
    _.each(statusList, _.bind(function(status, key) {
        var r = rollup[key];
        var s = {
            key: key,
            label: status,
            count: r ? r.count : 0,
            grade: r ? r.grade : '',
            color: r ? r.color : 'white',
            active: key === this.statusState ? true : false
        };
        statusCollection.push(s);
    }, this));
    this.statusCollection = statusCollection;

    this.statusWidth = 100 / statusCollection.length + '%';

    this._renderHtml();
},

```

Given the data structure needs defined in the template code in Step 3.3.1, we know that we need an array that looks like:

```

statusCollection: [
    {
        "key": "Active",
        "label": "Active",
        "count": 5,
        "grade": "C",
        "color": "#2b00d5",
        "active": true
    },
    ...
    {
        "key": "Unknown",
        "label": "Unknown",
        "count": 2,
        "grade": "B-",
        "color": "#002bd5",
        "active": false
    }
]

```

Starting with the function signature:

```
buildStatusCollection: function(chartData, gradeScale) {
```

We know we need an array of Super Group (account) statuses, so we can use the `getAppListStrings` language utility again.

```
var statusList = app.lang.getAppListStrings('account_status_list');
```

This give us:

```
account_status_list: {
  "Active": "Active",
  "Defunct": "Defunct",
  "Reactivated": "Reactivated",
  "Unknown": "Unknown"
}
```

Then roll up the `chartData` to construct an average for each status (see Step 3.5 below):

```
var rollup = this.rollupStatusAverages(chartData, gradeScale);
```

Which give us the following data:

```
rollup: {
  "Active": {
    "key": "Active", "count": 5, "grade": "C", "color": "#2b00d5"
  },
  ...
  "Unknown": {
    "key": "Unknown", "count": 2, "grade": "B-", "color": "#002bd5"
  }
}
```

We now can loop through the `statusList` we defined above and populate the `statusCollection` using values from the `chartData` rollup (if they exist or default values if not):

```
// build the status collection from status list
// with values from the chart data rollup
var statusCollection = [];
_.each(statusList, _.bind(function(status, key) {
  var r = rollup[key];
  var s = {
    key: key,
    label: status,
    count: r ? r.count : 0,
    grade: r ? r.grade : '',
    color: r ? r.color : 'white',
    active: key === this.statusState ? true : false
  };
});
```

```

        statusCollection.push(s);
    }, this));
    this.statusCollection = statusCollection;

```

Next we need to set the `statusWidth` variable in the `this` scope for the template tab width:

```

    this.statusWidth = 100 / statusCollection.length + '%';

```

And finally rerender the template so that the status tabs are properly displayed:

```

    this._renderHtml();
},

```

The call to this function is made inside the `evaluateResult` function just before the closing brace of `chartData.values` check:

```

if (chartData.values && chartData.values.length) {
    ...
    this.buildStatusCollection(chartData, gradeScale);
}

```

### Step 3.3.5: Add the method `rollupStatusAverages` to the JS controller

Since we made a call in `buildStatusCollection` to the function `rollupStatusAverages`, we now need to construct that function. For each Super Group status, the algorithm calculates the average rating for all Super Groups with that status.

Refer to the following source data structures while inspecting the code block below that calculates the average.

```

chartData: {
  "label": ["A-", "B+", "C-", "D", "F", "A+", "C+", "B", "B-", "D+"],
  "values": [
    {
      "label": "Active",
      "values": [1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
    }
  ]
}

gradeScale: {
  "A+": {
    "index": 0, "value": 0, "key": "A+", "color": "#00ff00"
  },
  ...
  "F": {
    "index": 12, "value": 1, "key": "F", "color": "#ff0000"
  }
}

```

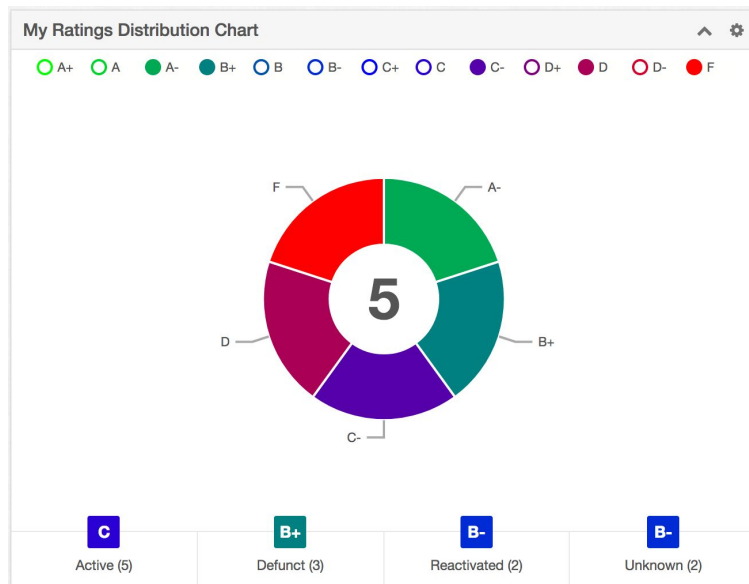
```

rollupStatusAverages: function(chartData, gradeScale) {
  var rollup = {};
  chartData.values.map(function(value) {
    var count = d3.sum(value.values);
    var average = Math.round(d3.sum(value.values, function(v, i) {
      var key = chartData.label[i]; // A-
      var gradeIndex = gradeScale[key].index; // 2
      return v * gradeIndex / count;
    }));
    var grade = _.find(gradeScale, function(grade) {
      return grade.index === average;
    });
    var status = {
      key: value.label,
      count: count,
      grade: grade.key,
      color: grade.color
    };

    rollup[value.label] = status;
  });
  return rollup;
},

```

If you are confident that your code is correct it's time to save and see what you end up with:



Hey, that's looking much more informative. Good job! If you click on a status tab the pie chart data should rerender with the new data for that status.

## Part 4: Final formatting and finishing

### Step 4.1: Data Story

Like we did in Parts 2 and 3, now it's time to reassess our data story and compare with the current dashlet to see if we've met all our objectives. Our data story is: *"As Professor M, I want to monitor the ratings of all my Super Groups by viewing the average rating for each status and the rating distribution and count for a selected Super Groups sorted by rating."* Maybe it's me but I think our "persona" Professor M can easily find the information he is looking for. What do you think? Is there more we can do to make the dashlet more user friendly?

How about we:

1. Add a tooltip to the status tab label to make it clear that the rating is an average
2. Add some visual user feedback for tab click events
3. Make it easier to use the keyboard to tab between the tabs
4. Format the donut hole text to make it clear what the number represents

There is often the temptation to overload a visualization with a lot of cool features, but keep in mind that sometimes those improvements increase complexity and make it harder for the user to easily and quickly get the information they need.

### Step 4.2: Identify Data Source

As in Part 2 and Part 3, let's continue to use Report API as our data source.

### Step 4.3: Implement Chart Dashlet

Goals for this step:

- Improve tooltips for tabs
- Enhance accessibility with CSS styling and DOM attributes
- Custom hole text formatter

#### Step 4.3.1: Add tooltips to status average tabs

Sugar will automatically add tooltips to DOM elements if they have the correct attributes. Inside our HBS template let's make the following change to the `span.label` element:

```
<span class="label" style="background-color:{{value.color}}" rel="tooltip"
title="{{value.label}} Groups: {{value.count}}&#10;Average Rating:
{{value.grade}}">{{value.grade}}</span>
```



The key attributes here are `rel` and `title`. Note that we can't use HTML code in the title but we can use unicode characters. Here we are using `&#10;` to create a line break. The tooltip should look like:



### Step 4.3.2: Update CSS and HBS template to support accessibility

Next lets add some CSS so that the user will have a visual clue that the tabs are “clickable”. In the `<style>` block at the top of our HBS template lets add the following:

```
.my-ratings-chart-wrapper .opportunity-metric {
  cursor: pointer;
}
.my-ratings-chart-wrapper .opportunity-metric .label {
  cursor: pointer;
}
```

If you now try hovering over the tab or average indicator the cursor will now be a “hand.” This is a usability convention that indicates that the element is clickable.

The next step in improving the usability of our dashlet is to include some ARIA (Accessible Rich Internet Applications) attributes. These attributes will make the dashlet easier for everyone to use which is the ultimate goal of accessibility. In our HBS template, add the `tabindex`, `role`, `aria-label`, `aria-pressed`, and `aria-current` attributes to the `div.opportunity-metric` element:

```
<div class="opportunity-metric" data-status="{{value.key}}" style="width:{{../statusWidth}}"
  tabindex="0" role="button"
  aria-label="{{value.count}} {{value.label}} groups with {{value.grade}} average rating"
  aria-pressed="{{#if value.active}}true{{else}}false{{/if}}"
  aria-current="{{#if value.active}}true{{else}}false{{/if}}">
```

The `tabindex` and `role` attributes will make our tabs behave like buttons and will include them in the tab flow. The `aria-label` attribute will be read by accessibility screen readers. The `aria-pressed` and `aria-current` attributes will indicate the current state of the tab.

Then add the following CSS to the HBS style block:

```
.my-ratings-chart-wrapper .opportunity-metric[aria-pressed=true]
.opportunity-metric-description {
  font-weight: bold;
}
```

```

.my-ratings-chart-wrapper .opportunity-metric:focus .label {
  outline: 3px auto #589bda;
  outline: 3px auto -moz-mac-focusring;
  outline: 3px auto -webkit-focus-ring-color;
}

```

These declarations will add bold styling to the tab label and a “focus ring” around the currently focused tab which will provide the user some visual feedback when the tab is clicked.

Then add a new event to the JS controller:

```
'keyup [data-status]': 'changeStatus'
```

And finally modify the `changeStatus` function like so:

```

changeStatus: function(evt) {
  var state = $(evt.currentTarget).data('status');
  if (state && (evt.type === 'click' || evt.keyCode === 13 || evt.keyCode === 32)) {
    this.statusState = state;
    this.loadData();
  }
},

```

These event handlers will enable the user to tab through the tabs and activate them with the spacebar or enter key. Try this by clicking on the “Active” tab, then pressing the tab key to move between the tabs. Pressing the spacebar or enter key will activate the currently focused tab.

### Step 4.3.3: Provide a custom formatter for the donut hole text

The last step in our dashlet development journey is to change the content of the donut hole text. Currently there is just a large number in the middle without any context so the user has to spend thought cycles decoding the meaning. That’s not good. Fortunately the Sugar chart library has a way to customize the donut hole content.

In the JS controller, modify the chart model instance code in the initialize method to include the following custom formatter:

```

this.chart = nv.models.pieChart()
...
.holeFormat(_.bind(function(wrap, data) {
  var wrapEnter = wrap.selectAll('text').data([null]).enter().append('g')
    .attr('transform', 'translate(0,5)');
  wrapEnter.append('text')
    .text(this.chart.hole())
    .attr('class', 'nv-pie-hole-value')
    .attr('style', 'font-size: 3em');
  wrapEnter.append('text')
    .text(this.statusState)
    .attr('class', 'nv-pie-hole-label')

```

```

        .attr('dy', '1.5em')
        .attr('style', 'font-size: 1em');
    }, this))

```

**Sucrose Changes:** the chart CSS classes have changed in Sucrose. Now the namespace starts with “sc-” instead of “nv-” so the class attributes above would use the `.sc-pie-hole-value` and `.sc-pie-hole-label` selectors.

The `holeFormat` method of the `pieChart` class expects a function that will return an SVG code snippet that will be appended to the SVG chart DOM in place of the hole text. Now that we have some custom code in place we can target those elements for other uses.

Because the call to the Report API to retrieve the requested records can take some time, it would be nice for us to provide some feedback to the user that their click actually is working and that the data is currently being loaded. To accomplish this we’ll modify the `changeStatus` function in the JS controller to replace the `.nv-pie-hole-label` with a “Loading...” message:

```

changeStatus: function(evt) {
    var state = $(evt.currentTarget).data('status');
    if (state && (evt.type === 'click' || evt.keyCode === 13 || evt.keyCode === 32)) {
        d3.select(this.el)
            .select('svg#' + this.cid + ' .nv-pie-hole-label')
            .text('Loading...');
        this.statusState = state;
        this.loadData();
    }
},

```

Reload the dashlet and click the tab. You should see the donut hole label change until new data is returned.

**Sucrose Changes:** the chart CSS classes have changed in Sucrose. Now the namespace starts with “sc-” instead of “nv-” so the class attributes above would use the `.sc-pie-hole-label` selector.

## Summary

Now that we have our dashlet complete, it's time for some user testing. Without providing any instruction, watch someone view and interact with the dashlet. Did they ask questions? Did they know to click on the various elements? As you test with multiple users you will get many different responses. Its up to you to decide how best to balance the feedback with the goals for your dashlet. Good luck!

## Extra credit

Want to keep going? Here are some improvements that you can try to make as you work along with the steps above:

- Is it possible to not make a call to the Report API to get new data for each tab click?
- What would you need to change to include a count in the pie chart slice labels?
- Once 7.10 is released, try modifying your dashlet to work with the Sucrose chart library.

## Share your success

We'd love to see your progress! Tweet a screenshot of your custom chart dashlet or a selfie of yourself with your favorite Sugar tutorial expert. Here are some ideas to get you started:

- "Just built my first Sugar custom dashlet using the Sugar REST API at an #UnCon #tutorial! #LifelsSweet"
- "Really enjoyed the Sugar custom dashlet tutorial developed by @hhrogersii #tutorial at #UnCon!"

If you continue on with the extra credit, take this tutorial in a new direction, or have feedback, we'd love to hear about it! Email us at [developers@sugarcrm.com](mailto:developers@sugarcrm.com).

## Additional resources

For more information about the new Sucrose library you can find information at:

- Sucrose website: [sucrose.io](http://sucrose.io)

## Need help?

Post your questions in the UnCon Community:

<https://community.sugarcrm.com/community/uncon>.