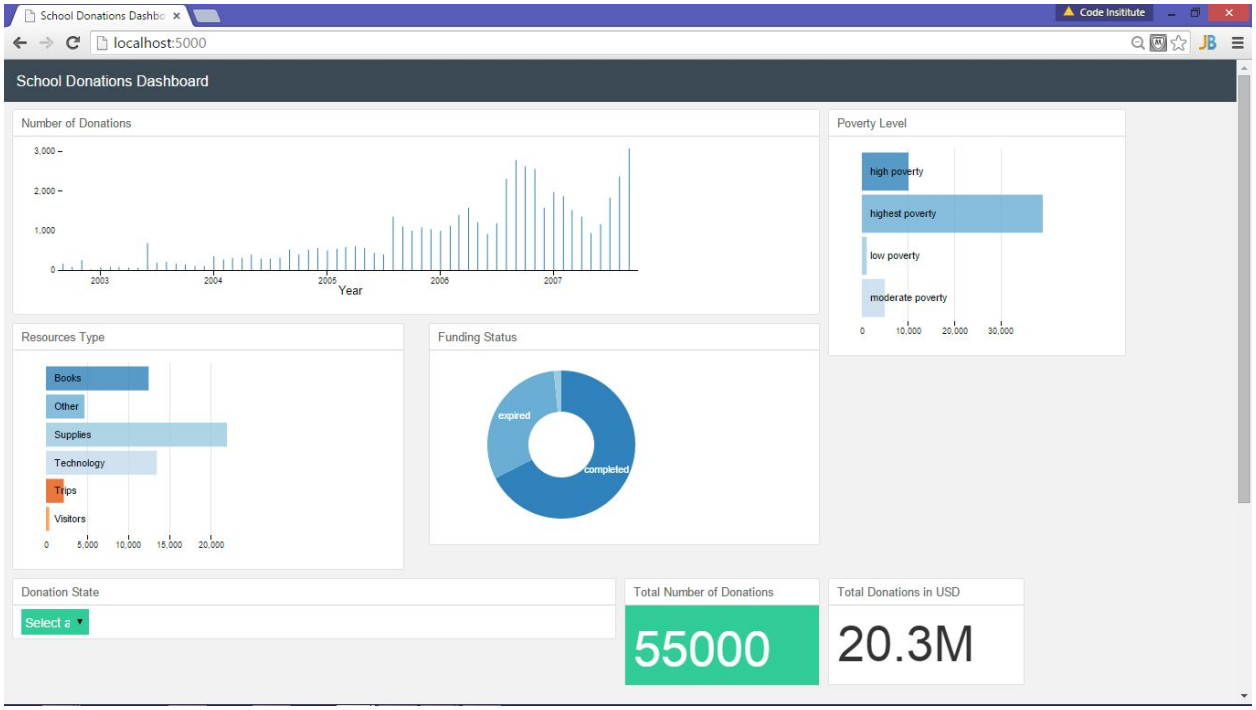
Stream 2 Project



The goal of this project is to expand on the skillsets and tools for creating a meaningful interactive data visualization learned throughout Streams 1 & 2 . To do this, we will use a dataset from DonorsChoose.org to build a data visualization that represents school donations broken down by different attributes over a timeline. We will be covering a wide range of technologies: MongoDB for storing and querying the data, Python for building a web server that interacts with MongoDB and serves html pages, Javascript libraries: d3.js, dc.js queue.js and crossfilter.js for building interactive charts.

Background

DonorsChoose.org is a US based nonprofit organization that allows individuals to donate money directly to public school classroom projects. Public school teachers post classroom project requests on the platform, and individuals have the option to donate money directly to fund these projects. The classroom projects range from pencils and books to computers and other expensive equipments for classrooms. In more than 10 years of existence, this platform helped teachers in all US states to post more than 7700,000 classroom project requests and raise more than $280,000,000. DonorsChoose.org has made the platform data open and available for making discoveries and building applications. In this project we will be using one of the available datasets for building an interactive data visualization that represents school donations broken down by different attributes.

The components of our project and their function:

1. D3.js: A javascript based visualization engine which will render interactive charts and graphs based on the data.
2. Dc.js: A javascript based wrapper library for D3.js which makes plotting the charts a lot easier.
3. Crossfilter.js: A javascript based data manipulation library. Enables two way data binding.
4. Queue.js: An asynchronous helper library for JavaScript.
5. Mongo DB: NoSQL Database used to convert and present our data in JSON format.

6. Flask: A Python based micro - framework used to serve our data from the server to our web based interface

What we'll do:

Together we will build the core components and structure of a working Dashboard. This will include:

1. Creating the Python app required to server the database content to the web interface
2. Writing the HTML required to display the dashboard
3. Importing the JavaScript libraries and writing the code required to render the data to our dashboard elements
4. Creating core CSS used to style dashboard elements

What you will do:

1. You will organise the presentation layer layout to display the data a visually effective manner.
2. You will add at least one additional data dimension and associated visualization to the data already presented on the dashboard
3. You will include additional functionality that provides a Dashboard tutorial which targets each Dashboard element with an explanation of its purpose.

4. You will embed the dashboard into a site - either one you have already created or a new simple site instance.
5. You will make a great and wonderful thing of beauty.

Let's Build the Project!

Uploading the data to MongoDB

You are provided with a file called opendata_projects_clean.csv . The data is represented in csv format. You will upload the file to an instance of MongoDb running on your machine. In doing so, the content will be converted to JSON format.

1. Run mongoDB by running the command *mongod* in your Terminal/Command Prompt .
2. Leave the prompt running as it is and open another Terminal/Command Prompt window.
3. Copy the csv file to the same location as the directory opened in the second terminal window
4. Enter the following command:

```
mongoimport -d donorsUSA -c projects --type csv --file
opendata_projects_clean.csv --headerline
```

- The database created: `donorsUSA`
- The collection name: `projects`
- The date type to be uploaded: `csv`
- The filename: `opendata_projects_clean.csv`
- Treat the first record imported as the field names: `--headerline`

There are over 87,000 records in the file, so it will take a few minutes to upload the data. You will see a progress indicator in the terminal letting you know how much data has approximately been uploaded.

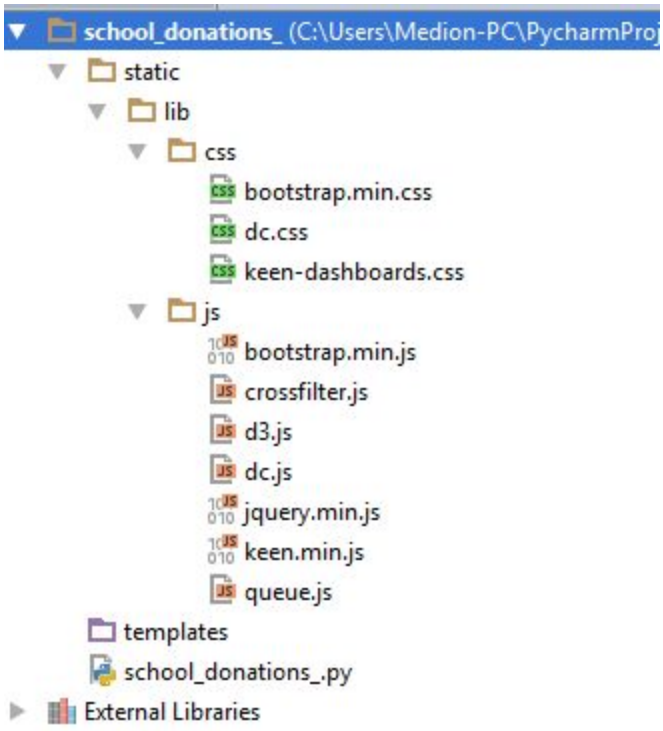
5. Open Mongo Management Studio to see the uploaded data. It's now in JSON format

   We will use the following collection attributes/fields as the basis of our project:

   1. school_state
   2. resource_type
   3. poverty_level
   4. date_posted
   5. total_donations
   6. funding_status

Set up our project structure

Create a new Flask project in Pycharm called school_donations_
In addition to the default static and templated directories, Modify the project to look like below:



We've create a lib directory under the default static directory. Under lib we have added a css directory and a js directory. These directories this will contain our 3rd party JavaScript and CSS content. Go ahead and add the files you have been supplied with to the respective directories.

Access MongoDB using Python

We will be use Python Flask for building a server that interacts with MongoDB and renders the html page that contains our charts.

1. Create a new Flask project in Pycharm called school_donations
2. Add the code below to school_donations.py set up our database connection and collection call.
3. You'll need to install the bson package for the code to work

4. We'll also set up a route to a file called index.html file that we'll create next.

```
from flask import Flask
from flask import render_template
from pymongo import MongoClient
import json
from bson import json_util
from bson.json_util import dumps

app = Flask(__name__)

MONGODB_HOST = 'localhost'
MONGODB_PORT = 27017
DBS_NAME = 'donorsUSA'
COLLECTION_NAME = 'projects'
FIELDS = {'funding_status': True, 'school_state': True, 'resource_type':
True, 'poverty_level': True,
          'date_posted': True, 'total_donations': True, '_id': False}




@app.route("/")
def index():
   return render_template("index.html")


@app.route("/donorsUS/projects")
def donor_projects():
   connection = MongoClient(MONGODB_HOST, MONGODB_PORT)
   collection = connection[DBS_NAME][COLLECTION_NAME]
```

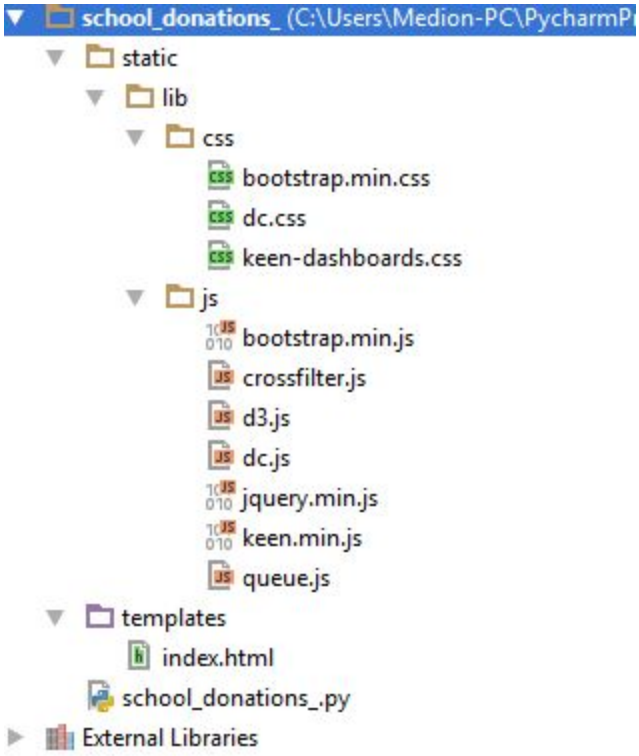
```
    projects = collection.find(projection=FIELDS, limit=55000)
    json_projects = []
    for project in projects:
        json_projects.append(project)
    json_projects = json.dumps(json_projects, default=json_util.default)
    connection.close()
    return json_projects


if __name__ == "__main__":
    app.run(debug=True)
```

Create our HTML page to display the dashboard

Add a new file named index.html to the templates directory

Our project structure should now look like below:



Add the code below. Notice that the code also refers to our 3rd party css and javascript, but also to a file called graph.js that we will create ourselves. graph.js will contain the logic and data binding for our dashboard elements.

```
<!DOCTYPE html>
<html>
<head>
    <title>School Donations Dashboard</title>
    <link rel="stylesheet" href="./static/lib/css//bootstrap.min.css">
    <link rel="stylesheet" href="./static/lib/css/keen-dashboards.css">
    <link rel="stylesheet" href="./static/lib/css/dc.css">
    <link rel="stylesheet" href="./static/css/custom.css">
```

```
 </head>
<body class="application">

<div class="navbar navbar-inverse navbar-fixed-top" role="navigation">
   <div class="container-fluid">
       <div class="navbar-header">
           <a class="navbar-brand" href="./">School Donations
Dashboard</a>
       </div>
   </div>
</div>









<div  id="outer" class="container-fluid">

   <div class="row">

       <div class="col-sm-8">
           <div class="row">

               <!-- Time Chart -->
```

```
<div class="col-sm-12">
    <div class="chart-wrapper">
        <div class="chart-title">
            Number of Donations
        </div>
        <div class="chart-stage">
            <div id="time-chart"></div>
        </div>
    </div>
</div>
<!-- Time Chart -->

<!-- Resources -->
<div class="col-sm-6">
    <div class="chart-wrapper">
        <div class="chart-title">
            Resources Type
        </div>
        <div class="chart-stage">
            <div id="resource-type-row-chart"></div>
        </div>
    </div>
</div>
<!-- Resources -->


<!-- funding Pie -->
<div class="col-sm-6">
    <div class="chart-wrapper">
        <div class="chart-title">
            Funding Status
        </div>
        <div class="chart-stage">
```

```
                    <div id="funding-chart"></div>
                </div>
            </div>
        </div>
        <!-- funding Pie -->

    </div>
</div>

<!-- Poverty -->
<div class="col-sm-3">
    <div class="chart-wrapper">
        <div class="chart-title">
            Poverty Level
        </div>
        <div class="chart-stage">
            <div id="poverty-level-row-chart"></div>
        </div>
    </div>
</div>
<!-- Poverty -->




<!-- menu select -->
<div class="col-sm-6">
    <div class="chart-wrapper">
        <div class="chart-title">
            Donation State
        </div>
        <div class="chart-stage">
            <div id="menu-select"></div>
```

```
        </div>
    </div>
</div>
<!-- menu select -->

<!-- Metric 1 -->
<div class="col-sm-2">
    <div class="chart-wrapper">
        <div class="chart-title">
            Total Number of Donations
        </div>
        <div class="chart-stage" style=" background: #33CC99;
            position: relative;
            font-size: 29px;
            color: white;
            vertical-align: middle;
            text-align: center;
            padding-top: 12px;
            padding-bottom: 0px;">
            <div id="number-projects-nd"></div>
        </div>
    </div>
</div>
<!-- Metric 1 -->

<!-- Metric 2 -->
<div class="col-sm-2">
    <div class="chart-wrapper">
        <div class="chart-title">
            Total Donations in USD
        </div>
        <div class="chart-stage">
            <div id="total-donations-nd"></div>
```

```
                </div>
            </div>
        </div>
        <!-- Metric 2 -->

    </div>
</div>

<hr>
<p class="small text-muted"> Data viz &#9829; from Code
Institute</a></p>

</div>
<script src="./static/lib/js/jquery.min.js"></script>
<script src="./static/lib/js/bootstrap.min.js"></script>
<script src="./static/lib/js/crossfilter.js"></script>
<script src="./static/lib/js/d3.js"></script>
<script src="./static/lib/js/dc.js"></script>
<script src="./static/lib/js/queue.js"></script>


<script src="./static/lib/js/keen.min.js"></script>
<script src='./static/js/graphs.js' type='text/javascript'></script>
</body>
</html>
```

Enabling data binding to our frontend using D3.js,Dc.js,Crossfilter.js and Queue.js

We will be utilizing the following libraries for visualization:

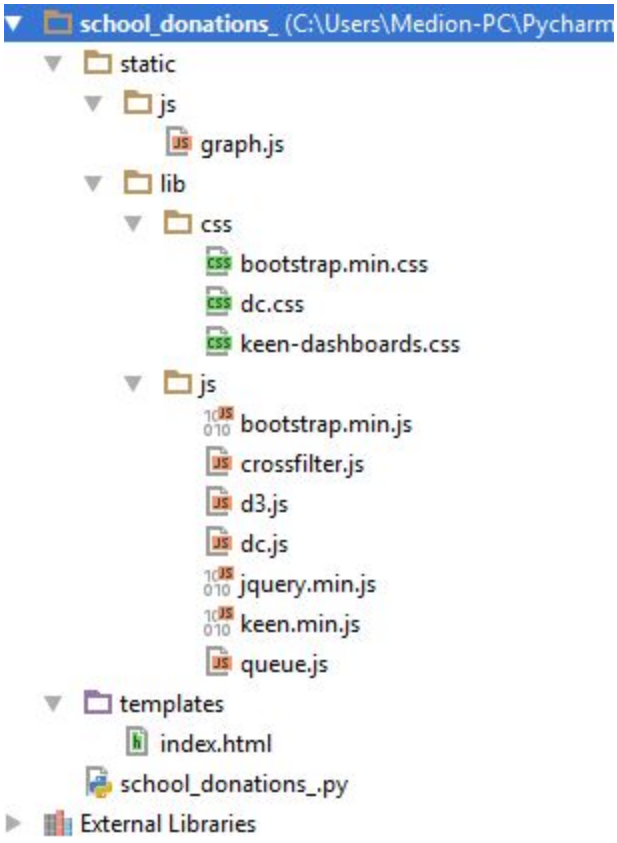
1. D3.js: Which will render our charts. D3 creates svg based charts which are passed into html div blocks
2. Dc.js: which we will use as a wrapper for D3.js. This allows use to only have to add the basic chart parameters to our code to get them up and running
3. Crossfilter.js: which is used for exploring large multivariate datasets in the browser. Really great for slicing and dicing data. Enables drill down based analysis
4. queue.js: An asynchronous helper library for data ingestion involving multiple api's. Its job is to wait until the data is available from each api before passing on the combined data for processing
5. Dc.css : Contains the styling directives for our dc charts
6. keen.js : A dashboard template libary
7. bootstrap: used in conjunction with keen.js to layout our dashboard elements

In addition to the libraries listed above we will create our own JavaScript file called graph.js

We will inject our data from our python class into this file. The data will then be filtered using crossfilter.js before being bound to the charts using a combination of D3.js and DC.js. We also use Queue.js in case we need to read from additional datasources/api's at a later date.

Create graph.js and add it to its own directory below the static directory in its own directory called js . We add it here because this is is where our custom logic will live so we separate it from the 3rd party libraries. The 3rd party JavaScript libraries in the lib folder will generally remain unchanged. This is seen as good project organisational structure.

The project structure should now look like below:



Add the code below to graph.py

```
queue()
    .defer(d3.json, "/donorsUS/projects")
    .await(makeGraphs);

function makeGraphs(error, projectsJson) {

    //Clean projectsJson data
    var donorsUSProjects = projectsJson;
    var dateFormat = d3.time.format("%Y-%m-%d %H:%M:%S");
    donorsUSProjects.forEach(function (d) {
        d["date_posted"] = dateFormat.parse(d["date_posted"]);
        d["date_posted"].setDate(1);
        d["total_donations"] = +d["total_donations"];
```

```
});


//Create a Crossfilter instance
var ndx = crossfilter(donorsUSProjects);

//Define Dimensions
var dateDim = ndx.dimension(function (d) {
    return d["date_posted"];
});
var resourceTypeDim = ndx.dimension(function (d) {
    return d["resource_type"];
});
var povertyLevelDim = ndx.dimension(function (d) {
    return d["poverty_level"];
});
var stateDim = ndx.dimension(function (d) {
    return d["school_state"];
});
var totalDonationsDim = ndx.dimension(function (d) {
    return d["total_donations"];
});

var fundingStatus = ndx.dimension(function (d) {
    return d["funding_status"];
});


//Calculate metrics
var numProjectsByDate = dateDim.group();
var numProjectsByResourceType = resourceTypeDim.group();
var numProjectsByPovertyLevel = povertyLevelDim.group();
var numProjectsByFundingStatus = fundingStatus.group();
```

```
    var totalDonationsByState = stateDim.group().reduceSum(function
(d) {
        return d["total_donations"];
    });
    var stateGroup = stateDim.group();


    var all = ndx.groupAll();
    var totalDonations = ndx.groupAll().reduceSum(function (d) {
        return d["total_donations"];
    });

    var max_state = totalDonationsByState.top(1)[0].value;

    //Define values (to be used in charts)
    var minDate = dateDim.bottom(1)[0]["date_posted"];
    var maxDate = dateDim.top(1)[0]["date_posted"];

    //Charts
    var timeChart = dc.barChart("#time-chart");
    var resourceTypeChart = dc.rowChart("#resource-type-row-chart");
    var povertyLevelChart = dc.rowChart("#poverty-level-row-chart");
    var numberProjectsND = dc.numberDisplay("#number-projects-nd");
    var totalDonationsND = dc.numberDisplay("#total-donations-nd");
    var fundingStatusChart = dc.pieChart("#funding-chart");


    selectField = dc.selectMenu('#menu-select')
        .dimension(stateDim)
        .group(stateGroup);


numberProjectsND
```

```
        .formatNumber(d3.format("d"))
        .valueAccessor(function (d) {
            return d;
        })
        .group(all);

    totalDonationsND
        .formatNumber(d3.format("d"))
        .valueAccessor(function (d) {
            return d;
        })
        .group(totalDonations)
        .formatNumber(d3.format(".3s"));




timeChart
        .width(800)
        .height(200)
        .margins({top: 10, right: 50, bottom: 30, left: 50})
        .dimension(dateDim)
        .group(numProjectsByDate)
        .transitionDuration(500)
        .x(d3.time.scale().domain([minDate, maxDate]))
        .elasticY(true)
        .xAxisLabel("Year")
        .yAxis().ticks(4);

    resourceTypeChart
        .width(300)
        .height(250)
        .dimension(resourceTypeDim)
```

```
        .group(numProjectsByResourceType)
        .xAxis().ticks(4);

    povertyLevelChart
        .width(300)
        .height(250)
        .dimension(povertyLevelDim)
        .group(numProjectsByPovertyLevel)
        .xAxis().ticks(4);

    fundingStatusChart
        .height(220)
        .radius(90)
        .innerRadius(40)
        .transitionDuration(1500)
        .dimension(fundingStatus)
        .group(numProjectsByFundingStatus);


    dc.renderAll();
}
```

So what's involved in the code we just wrote above?

In our graph.js file we have the following :

A queue() function which utilizes the queue library for asynchronous loading. It is helpful when you are trying to get data from multiple API's for a single analysis. In our current project we don't need the queue functionality, but it's good to have a

code than can be reused as per the need. The queue function process that data hosted at the API and inserts it into the apiData Variable.

```
queue()
    .defer(d3.json, "/donorsUS/projects")
    .await(makeGraphs);

function makeGraphs(error, projectsJson) {
```

…

Then we do some transformations on our data using d3 functions. We pass the data inside the `projectsJson` variable into our dataSet variable. We then parse the date data type to suit our charting needs and set the data type of total_donations as a number using the + operator.

We also change the date type from string to datetime objects, and we set all projects date days to 1. All projects from the same month will have the same datetime value.

```
//Clean projectsJson data
  var donorsUSProjects = projectsJson;
  var dateFormat = d3.time.format("%Y-%m-%d %H:%M:%S");
  donorsUSProjects.forEach(function (d) {
      d["date_posted"] = dateFormat.parse(d["date_posted"]);
      d["date_posted"].setDate(1);
      d["total_donations"] = +d["total_donations"];
  });
```

Next Steps are ingesting the data into a crossfilter instance and creating dimensions based on the crossfilter instance. Crossfilter acts as a two way data binding pipeline. Whenever you make a selection on the data, it is automatically applied to other charts as well enabling our drill down functionality.

```
//Create a Crossfilter instance
```

```
var ndx = crossfilter(donorsUSProjects);

//Define Dimensions
var dateDim = ndx.dimension(function (d) {
    return d["date_posted"];
});
var resourceTypeDim = ndx.dimension(function (d) {
    return d["resource_type"];
});
var povertyLevelDim = ndx.dimension(function (d) {
    return d["poverty_level"];
});
var stateDim = ndx.dimension(function (d) {
    return d["school_state"];
});
var totalDonationsDim = ndx.dimension(function (d) {
    return d["total_donations"];
});

var fundingStatus = ndx.dimension(function (d) {
    return d["funding_status"];
});
```

Next we calculate metrics and groups for grouping and counting our data.

```
//Calculate metrics
   var numProjectsByDate = dateDim.group();
   var numProjectsByResourceType = resourceTypeDim.group();
   var numProjectsByPovertyLevel = povertyLevelDim.group();
   var numProjectsByFundingStatus = fundingStatus.group();
   var totalDonationsByState = stateDim.group().reduceSum(function
(d) {
       return d["total_donations"];
   });
   var stateGroup = stateDim.group();


   var all = ndx.groupAll();
   var totalDonations = ndx.groupAll().reduceSum(function (d) {
       return d["total_donations"];
   });

   var max_state = totalDonationsByState.top(1)[0].value;

   //Define values (to be used in charts)
   var minDate = dateDim.bottom(1)[0]["date_posted"];
   var maxDate = dateDim.top(1)[0]["date_posted"];
```

Now we define the chart types objects using DC.js library. We also bind the charts to the div ID's in index.html

```
 //Charts
   var timeChart = dc.barChart("#time-chart");
   var resourceTypeChart = dc.rowChart("#resource-type-row-chart");
   var povertyLevelChart = dc.rowChart("#poverty-level-row-chart");
   var numberProjectsND = dc.numberDisplay("#number-projects-nd");
```

```
    var totalDonationsND = dc.numberDisplay("#total-donations-nd");
    var fundingStatusChart = dc.pieChart("#funding-chart");
```

.

Nearly there! We assign properties and values to our charts. We also include a select menu to choose between any or all US states that have given donations for a particular date range

```
selectField = dc.selectMenu('#menu-select')
        .dimension(stateDim)
        .group(stateGroup);


numberProjectsND
        .formatNumber(d3.format("d"))
        .valueAccessor(function (d) {
            return d;
        })
        .group(all);

    totalDonationsND
        .formatNumber(d3.format("d"))
        .valueAccessor(function (d) {
            return d;
        })
        .group(totalDonations)
        .formatNumber(d3.format(".3s"));
```

```
timeChart
      .width(800)
      .height(200)
      .margins({top: 10, right: 50, bottom: 30, left: 50})
      .dimension(dateDim)
      .group(numProjectsByDate)
      .transitionDuration(500)
      .x(d3.time.scale().domain([minDate, maxDate]))
      .elasticY(true)
      .xAxisLabel("Year")
      .yAxis().ticks(4);

  resourceTypeChart
      .width(300)
      .height(250)
      .dimension(resourceTypeDim)
      .group(numProjectsByResourceType)
      .xAxis().ticks(4);

  povertyLevelChart
      .width(300)
      .height(250)
      .dimension(povertyLevelDim)
      .group(numProjectsByPovertyLevel)
      .xAxis().ticks(4);




fundingStatusChart
```

```
        .height(220)
        .radius(90)
        .innerRadius(40)
        .transitionDuration(1500)
        .dimension(fundingStatus)
        .group(numProjectsByFundingStatus);
```

And finally we call the dc renderAll() function which renders our charts

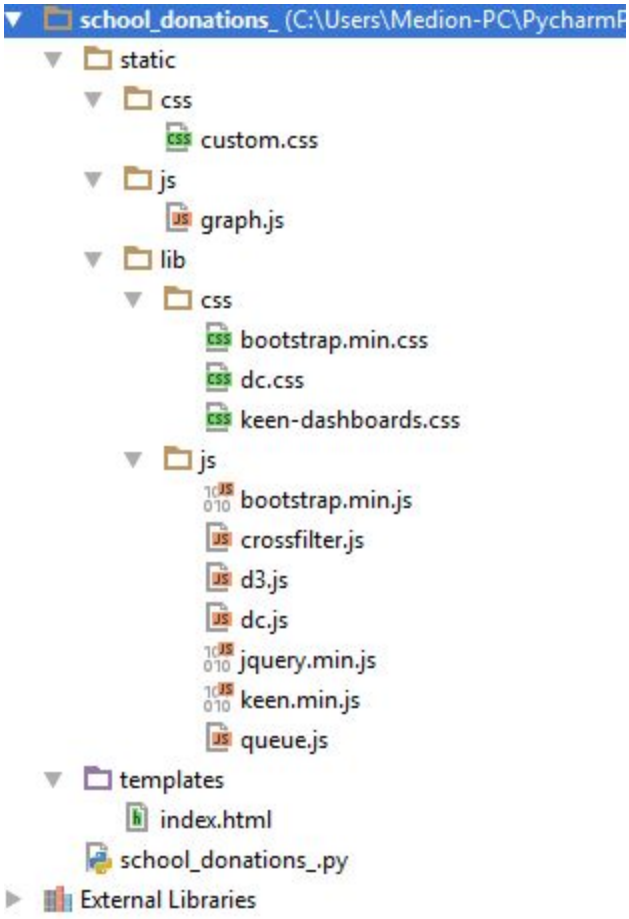
```
dc.renderAll();
```

Add some custom CSS

Finally we'll create our own css file for targeting our dashboard

Create a new directory called `css` inside the `static` directory and add a new file called `custom.css`

Your project structure should now look like below:



Add the styles below to custom.css

```
#number-projects-nd {
    font-size: 60px;
}

#total-donations-nd {
    font-size: 60px;
}



.dc-chart g.row text {
    fill: black;
```

```
    font-size: 12px;
}

#total-projects {
    background: #33CC99;
    width: 200px;
    position: relative;
    height: 60px;
    font-size: 29px;
    color: white;
    vertical-align: middle;
    text-align: center;
    padding-top: 12px;
    padding-bottom: 0px;
}

#net-donations {
    background: #33CC99;
    width: 200px;
    position: relative;
    height: 60px;
    font-size: 29px;
    color: white;
    vertical-align: middle;
    text-align: center;
    padding-top: 12px;
    padding-bottom: 0px;
}









.dc-select-menu {
    background: #33CC99;
```

```
    width: 80%;
    border: none;
    position: relative;
    height: 31px;
    font-size: 15px;
    color: white;
    vertical-align: middle;
    text-align: center;
    padding-top: 4px;
    padding-bottom: 0px;

}
```

And that's all the code needed to get up and running.

The whole processing works as follows:

1. when you type localhost:5000 into your browser URL window ,index() gets called

```
@app.route("/")
def index():
   return render_template("index.html")
```

2. The render_template() function redirects the process to the index.html page
3. At the bottom of index.html the graph.js file is loaded

```
<script src='./static/js/graphs.js' type='text/javascript'></script>
```

4. Inside graph.js the queue() function calls back to donor_projects() function in school_donations_.py .It nows how do this because of the route name it uses as a parameter

```
queue()
    .defer(d3.json, "/donorsUS/projects")
    .await(makeGraphs);
```

calls

```
@app.route("/donorsUS/projects")
def donor_projects():
….
```

5. donor_projects() then gets the data from the database and sends it back to queue() which then passes the data to the makeGraphs() function

```
function makeGraphs(error, projectsJson) {
...
```

6. The dashboard elements then use this data to populate themselves.

Notes.

- Make sure you have mongod running when testing , otherwise you'll get no data back

- Use the chrome debugger for testing your javascript. It's a busy enough project, so reduce your opportunities for pain and frustration by carefully debugging.
- Turn off caching in your chrome developer tools. This prevents further confusion and frustration of seeing out of date cached versions of your dashboard when developing
- Also run the debugger in Pycharm to debug through your python code.

What you will do: Reminder

6. You will organise the presentation layer layout to display the data a visually effective manner.
7. You will add at least one additional data dimension and associated visualization to the data already presented on the dashboard
8. You will include additional functionality that provides a Dashboard tutorial which targets each Dashboard element with an explanation of its purpose *See guidelines below this list.*
9. You will embed the dashboard into a site - either one you have already created or a new simple site instance.
10. You will make a great and wonderful thing of beauty.

New Functionality to add

Story-Boarding your dashboard using Intro.js

Using Intro.js we can attach pop-up tooltips to our html graphs (div's) and further more we can assign an order of appearance as well as style them to make it look snappier. So, essentially we can assign an interactive tool tip to our graphs and display helpful information to the user who is going to use the dashboard.

Download the zip file containing the intro.js and introjs.css file from the official page here and save them to the lib js and css folders in your app respectively. The intro.js file contains the javascript code for initializing our tool tips and their interactive features, while the introjs.css file contains the styling information for the tool tips.

Call the introjs.css file in the head of the html document.

```
<link rel="stylesheet" href="css/introjs.css" />
```

Call the intro.js file just before you close the body of the html document:

```
<script src="js/intro.js" type="text/javascript"></script>
```

We need to do just one more thing here. Create a button and initialize the Intro.js script in it.
Add this code to the to the dashboard (index.html) , possibly in the navigation bar, to create a simple html button and assign the Intro.js script start to it:

```
<button class="intro_button" type="button" autofocus
onclick="javascript:introJs().start();">Start Tour!</button>
```

Style the button to make it look less boring.

Assign tool tips to our div elements.

Locate the various div's and add this code in their definitions

```
data-step="1" data-intro="The description that you
want"style="font-weight: normal" data-position="right"
```

The properties highlighted in red are the ones we can change and need to change. Once we have a story line in mind we can assign the above code in that order to div's. Here data-step is the order in which the tool-tip for the div will get generated. So the first visual you want to display information for should have a data-step value of 1. data-intro contains the description that you want to display for that particular div. You can add html stuff like breaks and all inside the data-intro text as well. You can make the tooltip font bolder by modifying the weight property. Finally you can select the placement of the generated tool tip by using the data-position property. We should give a right data-position to the div's on the extreme left of our webpage and vice-versa otherwise the tool-tip me go out of screen. (It is possible that it won't happen with bootstrap.)

That's it. Get stuck in, be curious, and enjoy!