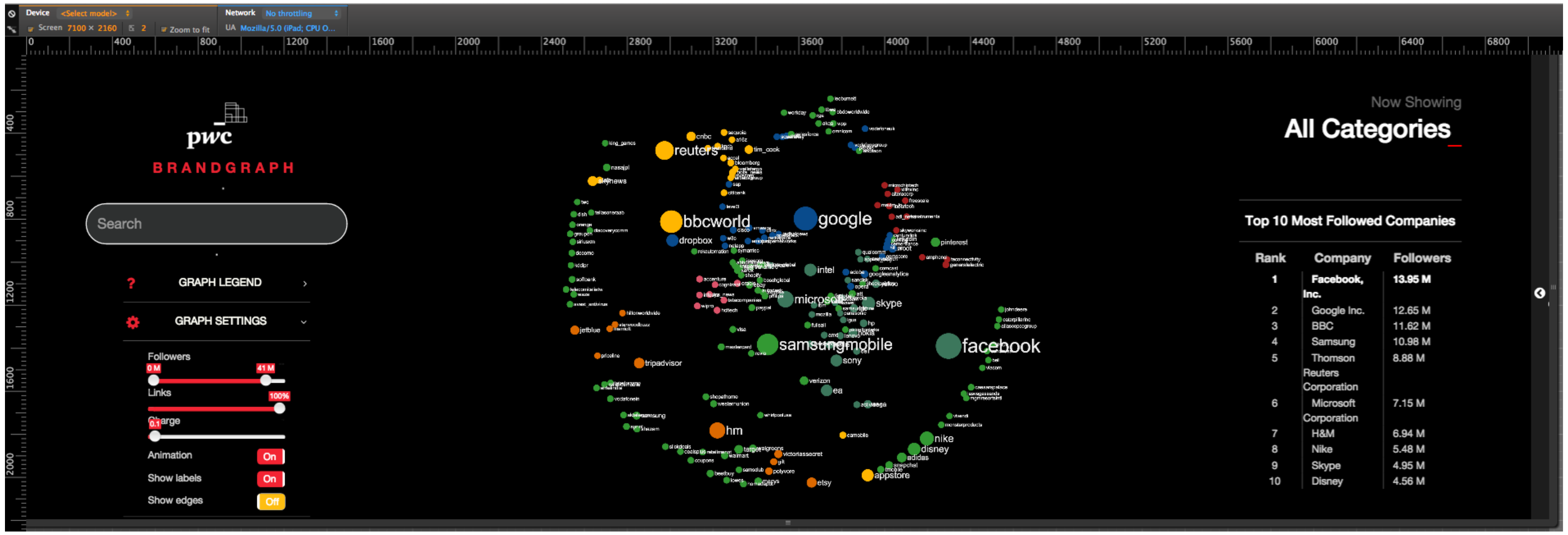


Followers 0 M – 41 M
Links 100%
Charge
Animation On
Show labels On
Show edges Off

Now Showing
All Categories

Top 10 Most Followed Companies

Rank	Company	Followers
1	**Facebook, Inc.**	**13.95 M**
2	Google Inc.	12.65 M
3	BBC	11.62 M
4	Samsung	10.98 M
5	Thomson Reuters Corporation	8.88 M
6	Microsoft Corporation	7.15 M
7	H&M	6.94 M
8	Nike	5.48 M
9	Skype	4.95 M
10	Disney	4.56 M

Table of Contents

Introduction

BrandGraph is a web application that can be used to display relationships between brands. Analyzing brands data can help user have a intuitive understanding about the brand scale, similarity, business intersection.

About this Document

We create this document as some kind of conclusion so that we can easily pick it up when we need it in new project.

Features

- Provide both desktop and wall version, user can use it both on browser and a wall device.
- Provide multiple query dimension(include brands, industries or categories) and filters.
- Provide a lot of graph parameters, which means all nodes, edges, text and even animations could be changed dynamically.

Used Technologys

- HTML/CSS/JavaScript
- Neo4j the Worldâ€™s Leading Graph Database
- Flask is a microframework for Python based on Werkzeug, Jinja2
- AngularJS is a popular structural framework for dynamic web apps
- Other Tools
    - Sigma.js - JavaScript library dedicated to graph drawing
    - Yeoman - Web application scaffolding tool
    - Git / Github- Version control tool and platform
    - Heroku - A cloud Platform-as-a-Service supporting several programming languages
    - Graphenedb - Neo4j graph database service

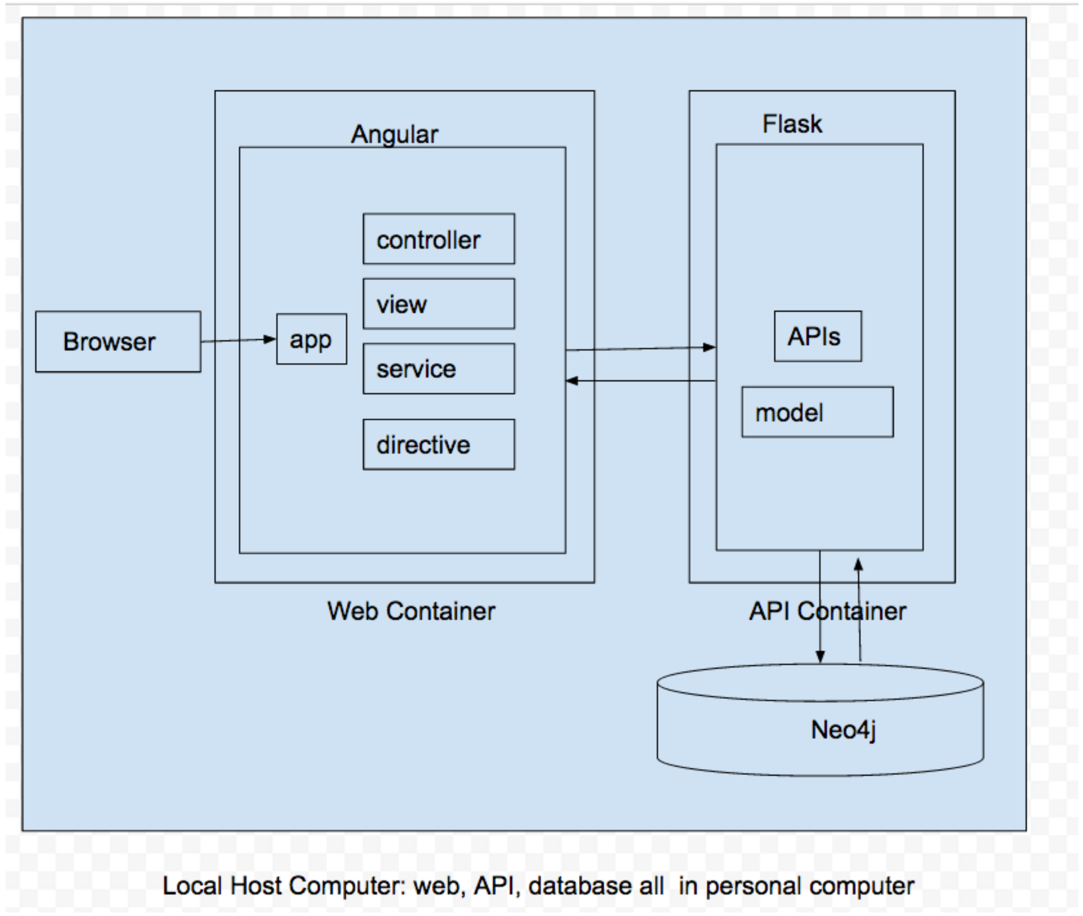
How to Run

BrandGraph's code was hosted on Github, you can config on your local environment according to the README. Or you can directly visit the deployed version on Heroku.

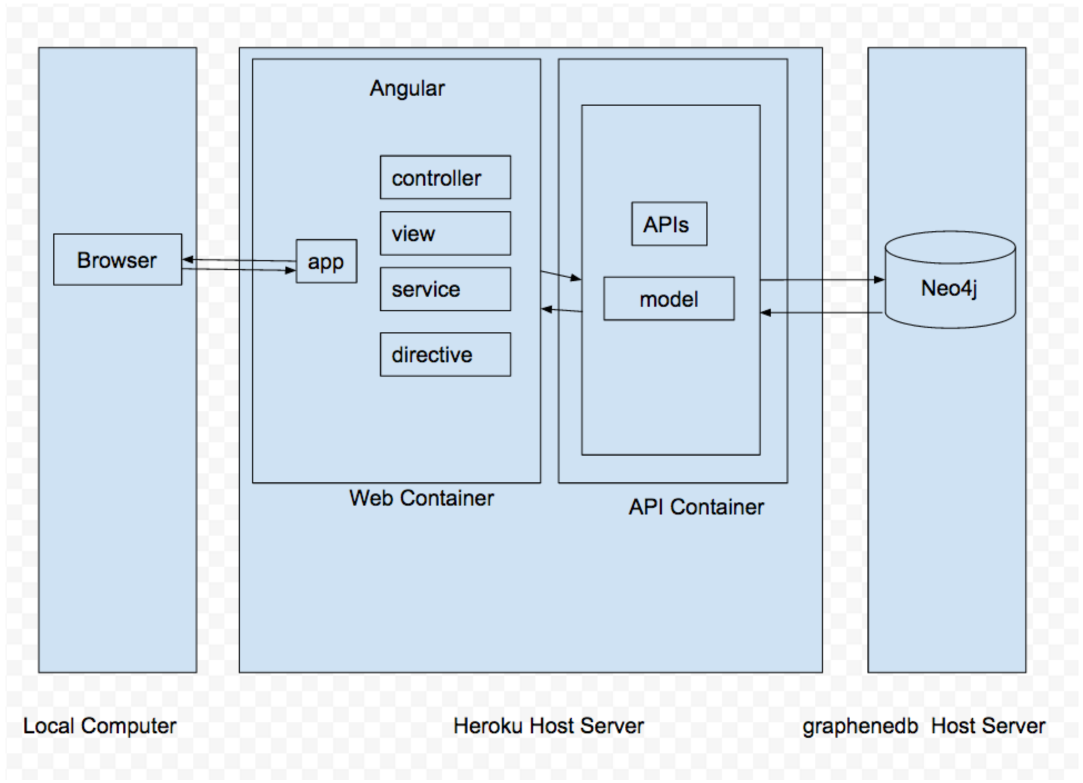
Core Implementation

Architecture

- Dev Architecture: how does the application run in our local environment



- Deployed Architecture: the deployed application



Database: Neo4j

We use Neo4j as database because it's same-like with our bussiness needs, you can download it or just refer Github README's step: **"db environment"** .

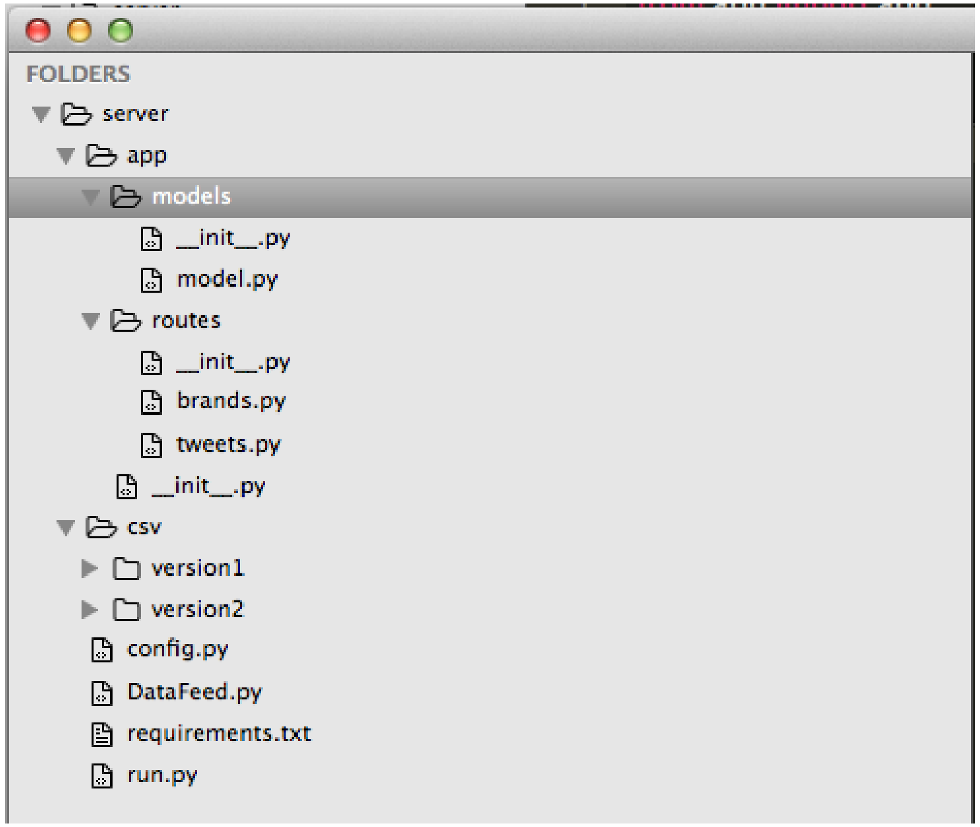
**Note:* * As the environment was ran locally, we just need to provide a database location then we can connect to it. But it was on a remote server, we may need to add username and password for access.

Server

Preparation

Make sure you have installed Python and Flask, if not please refer to the Github README's "**server environment**" part and make sure your environment right configured. After all that, let's get started to do some source code analysis!

Application Structure



```
FOLDERS
▼ server
  ▼ app
    ▼ models
        __init__.py
        model.py
    ▼ routes
        __init__.py
        brands.py
        tweets.py
      __init__.py
  ▼ csv
    ▶ version1
    ▶ version2
    config.py
    DataFeed.py
    requirements.txt
    run.py
```

- config.py: common used variables defined

- DataFeed.py: database initialization

- csv: data orignal files location

- app/__init__.py: application entrance, flask instance create here

- app/routes: routes define APIs

- app/models: query database, would been called by routes

Datafeed

We feed the database with csv files provided by customer via `python DataFeed.py`. Below would focus on DataFeed.py fragment for understanding. First, import packages would been usedï¼š

/brand_graph/server/DataFeed.py

```
# ...

'''
py2neo: a external python package used to integrate with neo4j
csv and system: python build-in module used for file read).
'''

from py2neo import neo4j, Graph, Node, Relationship
import csv
import sys
from config import NEO4J_DB_ADDR
# ...
```

As there are some system environment would be used more than one time, we recommand to write all them in to one file:

/brand_graph/server/config

```
#...
NEO4J_DB_ADDR = "http://localhost:7474/db/data/"
#...
```

Then we can create the database instance and batch writer, remember to empty the database to prevent data duplicate.

/brand_graph/server/DataFeed.py

```
graph = Graph(NEO4J_DB_ADDR)
graph.delete_all()
batch = neo4j.WriteBatch(graph)
```

After create connection, we would need to initialize the database structure:

1. Create nodes: by loop modularity.csv we get attributes and create brand, add it to batch writer then submit the batch into Neo4j.
2. Increase features: by loop twitter.csv and get more detailed brand features, query the concerned brand and update it with new features.
3. Ceate connections: by loop melt.csv and build the realationship between different brands.

/brand_graph/server/DataFeed.py

```
nodes_file = open(pathCSV + 'modularity.csv', "rb")
```

```
 # skip the header line because the csv file contains attribute name header
reader = csv.reader(nodes_file)
next(reader, None)

for id, name, group, page_rank in reader:
    index = index + 1
    brand = batch.create(Node(name= name.strip().lower(), group= group))
    batch.set_labels(brand, 'Brand')
    batch.submit()
```

Route Definition

There are three APIs been used in BrandGraph, they are list below:

Name	Method	URL
get all brands	GET	/brand_graph/api/v1.0/brands
get the brand with specified name	GET	/brand_graph/api/v1.0/brands/string:brandname
get the category with specified name	GET	/brand_graph/api/v1.0/category/string:categoryname

Here is the fragment of API: **get all brands**:

/brand_graph/server/app/routes/brands.py

```
import json
from flask.ext.cors import CORS, cross_origin
from app.models.model import Brand
from app import app

brandModel = Brand()

@cross_origin(origin='*', methods=['GET', 'POST', 'OPTIONS'], headers=['X-Requested-With', 'Content-Type', 'Origin'])
@app.route('/brand_graph/api/v1.0/brands',methods=['GET'])
def get_brands():
    brands = brandModel.get_all()
    return json.dumps(brands)
#...
```

There are some tips need to be noticed when format an API in Flask:

- Flask API is decorated with @app.route. The `app` is the Flask application instance imported from entrance module ***brand_graph/server/__init__.py***.
- As our web and server are hosted in different container, we must add corss_origin decorator.

- The data access process belong to model, so we abstract all the query process in model Brands.

Database Query

The *model.py* contains both query methods and data formated methods. Here we just try to explain how does we use py2neo query the database and obtain the data we want.

Of course the first step is new the database instance:

/brand_graph/server/app/models/model.py

```
from py2neo import Graph
from config import NEO4J_DB_ADDR

graph = Graph(NEO4J_DB_ADDR)

#...
```

Then we would define our class Brand and format the get_all methods structure:

/brand_graph/server/app/models/model.py

```
#...

class Brand(object):
    def get_all(self):
        nodes, edges, names, results_nodes = [], [], set(), {}
        try:
            # query loic code here
        except:
            pass
        return brands_results

#...
```

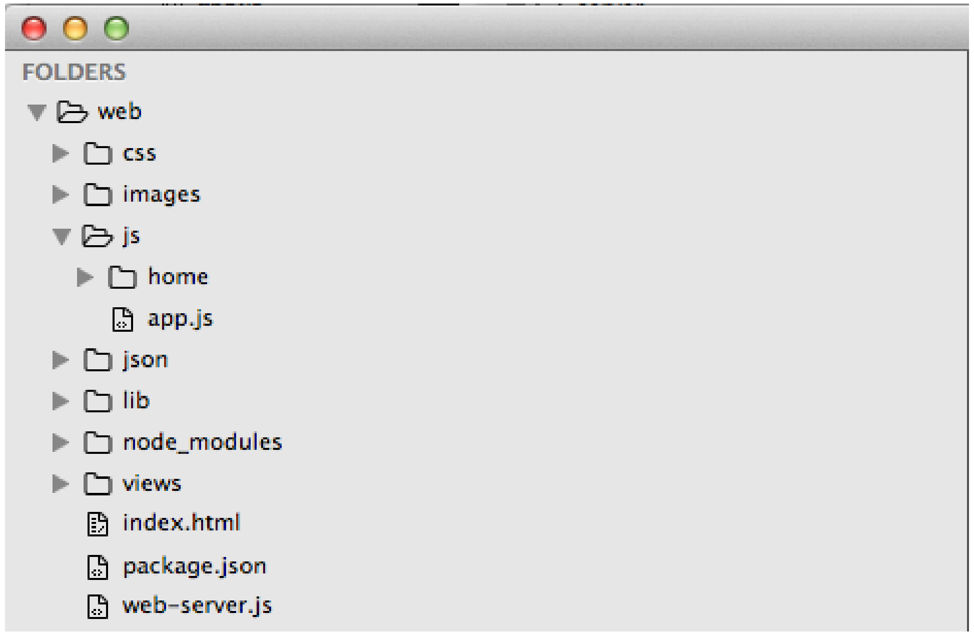
The Neo4j cypher is a querying language which can let us execute the SQL directly, below is the logic how we use `graph.cypher` to query and format response with customized methods:

```
#...

class Brand(object):
    def get_all(self):
        #...
```

```
query_nodes = "MATCH (n:Brand) WHERE exists(n.company) RETURN n.name, n.group, n.company, n.followers"
results_nodes = graph.cypher.execute(query_nodes)

for rec in results_nodes.records:
    name, group, company, followers = rec['n.name'], rec['n.group'], rec['n.company'], rec['n.followers']
    node = self.mapping_node(name, group, company, followers)
    nodes.append(node)

# the edges query and format logic here


nodes = sorted(nodes, key=itemgetter("followers"), reverse=True) # nodes sort decending by followers

#...

brands_results = {  'nodes': nodes, 'edges': edges, 'industries': self.mapping_industries()}

#...
```

Web: AngularJS Front-Side

Application Structure



```
FOLDERS
▼ web
  ▶ css
  ▶ images
  ▼ js
    ▶ home
      app.js
  ▶ json
  ▶ lib
  ▶ node_modules
  ▶ views
    index.html
    package.json
    web-server.js
```

- web-server.js: used to produce a web container

- requirements.txt: contains packages would been used by grunt tasks

- index.html: home page

- node_modules: packages installed location

- css/js/images/libs/json: as it described

User Cases

- Show Brands
  - By default we would show all brands
  - We can search by brand name, category name
  - We can click on brand and show top 10 concerned brands
- Filtess
  - Can set the tweets followers as filter
- Setting
  - Simple setting: graph edge visible, text visible, animation toggle, etc.
  - Advanced setting: more setting about graph

How to Integrate sigma.js with Angular.js

Sigma.js is a javascript library dedicated to graph drawing, for low coupling and better reusability, we write all graph-drawing code in service.js. You just need to define methods in *service.js* and expose methods would been used public.

/brand_graph/web/js/home/service.js

```
angular.module('brand_graph')
  .factory('sigma', ['$rootScope', function($rootScope) {

    var sigmaExport = {}, sigmaInstance;

    // ...

    var _draw = function() {
        // ...
    }

    var _initInstance = function() {
        sigmaInstance = // init sigma instane logic
    }
    _initInstance();

    // ...

    sigmaExport.s = sigmaInstanceï¼›
    sigmaExport.draw = _draw;ï¼›
    // ...

    return sigmaExport;
}]);
```

/brand_graph/web/js/home/home-controller.js

```
angular.module('brand_graph')
  .controller('HomeController', ['$scope', 'sigma', function ($scope, sigma) {

    // ...

    sigma.s.draw(); // use sigma serive exposed function directly

    // ...
}]);
```

Sigma.js Drawing

The Steps to Draw Graph

1. Create sigma instance with a container id and default setting

   ```
   sigmaInstance = new sigma({
       container: 'graphContainer',
           renderer: {
               container: document.getElementById('graphContainer'),
               type: 'canvas'
           },
       settings: defaults
   });
   ```

2. Draw Graph with Data

   ```
   sigmaInstance.graph.read(data);
   sigmaInstance.refresh();
   sigmaInstance.startForceAtlas2(force);
   ```

There are something need to be noticed:

1. The data's structure must satisfied some mode so that they could be known.
2. After read the data, you can still adjust nodes and edges before final refresh and drawing. Just loop the `sigmaInstance.graph.nodes()` or `sigmaInstance.graph.edges()`
3. The force variable is the paramters would been used for animation, you could refer to the official site for optional paramters.

How to use redraw the graph

If you need to redraw the graph with data, you can just do it like drawing, but remember to excute the kill and clear:

```
    sigmaInstance.killForceAtlas2();
    sigmaInstance.graph.clear();
    // draw steps in previous part
```

How to Active Nodes(hide and show)

This could be really easy to do, just loop the nodes and edges, determine it's visibility with yourself's logic.

```
sigmaInstance.graph.nodes().forEach(function(n) {
    // the showhide judge logic can be defined different
    var flag = nodeShowOrHideLogic(n);

    // we have saved the original color, label, size for reuse
    // the color could be set as transparent for hide
    n.color = flag ? n.originalColor : inactiveColor;
    n.size = flag ? n.originalSize * 2 : n.originalSize;
    n.label = flag ? n.originalLabel : '';
});
sigmaInstance.graph.edges().forEach(function(e) {
    var flag = edgeShowOrHideLogic(e);
    e.color = flag ? e.originalColor : inactiveColor;
    e.size = flag ? e.originalSize * 2 : e.originalSize;
});

sigmaInstance.refresh();
```

How to Place Nodes Around and Recover

There is a requirement in our project: Once you click a brand, place all concerned top 10 brands around it and hide the rest. What we gonna todo is change all nodes' visbible and coordinates attributes, below is the code fragment:

web/js/home/service.js

```
var placeAround = function(nodeId) {

    sigmaInstance.stopForceAtlas2();

    // connects is a tool methods define to get all connected nodes
    var activeHash = sigmaInstance.graph.connects(nodeId);

    // make it not empty for case judgement after
    activeHash[nodeId] = {};

    var activeNum = _.keys(activeHash).length;
    var radius = 1, i = 0;

    // calcuate the max x to determine the round circle
    sigmaInstance.graph.nodes().forEach(function(n) {
        if (radius < n.x) radius = n.x;
    });

    sigmaInstance.graph.nodes().forEach(function(n) {
        n.originalX = n.x;
        n.originalY = n.y;
        if (n.id === nodeId) {
            n.x = 0;
            n.y = 0;
        } else if (activeHash[n.id]) {
            n.x = Math.cos(Math.PI * 2 * i / (activeNum - 1)) * radius * 0.7;
            n.y = Math.sin(Math.PI * 2 * i / (activeNum - 1)) * radius * 0.7;
            i ++;
        }
    });

    sigmaInstance.refresh();
}
```

Toggle and Filter

To provide the user with toggle and filter function, we use different tools, here we would make brief introduction.

ion.rangeSlider

Sigma.js provide a lot of parameters for size, colors and animation. By use ion.rangeSlider we can detect the paramteter value change and trigger sigma redraw. Here is one slider example:

```
// define the element in page before instance slider
$("#slider-gravity").ionRangeSlider({
    min: 0,
    max: 10,
    from: scope.getSigmaForce('gravity'), // all value could be a calculated result
    type: 'single',
    step: 0.1,
    onFinish: function(obj) {
        // execute the simag redraw function
    }
});
```

bootstrap toggle

It's not include in bootstrap or bootstrap-tpls, so you have to download and import the bootstrap-toggle before use. After than you can delcare the input element as text input and bind on/off events on it.

/brand_graphp/web/views/components/forceComponent.html

```
<input id="linLogMode-button" type="checkbox" data-toggle="toggle">
```

/brand_graphp/web/js/home/directives.js

```
$('#linLogMode-button').bootstrapToggle('on').change(function() {
    // trigger the sigma.js drawing function
});
```

End