

4BCT1 – 2018

**Computer Science & Information Technology**

**Final Year Project**

**Supervisor**

**Dr. Owen Molloy**

[owen.molloy@nuigalway.ie](mailto:owen.molloy@nuigalway.ie)

**Author**

**Gary McMahon**

[g.mcmahon5@nuigalway.ie](mailto:g.mcmahon5@nuigalway.ie)

**14534653**

Table of Contents

[Final Year Project 3](#_Toc511125008)

[Acknowledgements 3](#_Toc511125009)

[Introduction 4](#_Toc511125010)

[Project Overview 4](#_Toc511125011)

[Problem Statement 4](#_Toc511125012)

[Solution 4](#_Toc511125013)

[Research 5](#_Toc511125014)

[Tools 5](#_Toc511125015)

[Languages 6](#_Toc511125016)

[Libraries/Frameworks 7](#_Toc511125017)

[Implementation 8](#_Toc511125018)

[Load JSON data 8](#_Toc511125019)

[Display nodes 9](#_Toc511125020)

[Draw Edges 14](#_Toc511125021)

[Draw Knowledge Nodes 20](#_Toc511125022)

[Draw Knowledge node (alternate method) 30](#_Toc511125023)

[Draw Actor Nodes 32](#_Toc511125024)

# Final Year Project

Welcome to my report for my final year project. The report will be split up into 5 parts with the final conclusions and results as the closing words.

* **Introduction** Introduce the project overview and how it will apply to the real world.
* **Research chapter** This chapter looks at some of the research undertook as part of the project and how it contributed to the end result.
* **Technical Review/ Research chapter** Discuss the underlying design decisions for the project UI that stemmed from examining the technologies and methods previously used in the project. Examine the current method used to display the graphs in the ResponSEAble codebase.
* **Implementation** chapter examines the code used in the creation of the project. Will look at descriptions of the technical issues in the project. Discuss features that were added to the project along with features that didn’t quite make the cut.
* **Results** Discuss how the project development, if final deliverables were met and evaluate the project.
* **Final conclusion**

# Acknowledgements

* Dr. Owen Molloy – Project supervisor.
* Caroline Brennan – A constant source of help and guidance for the project.
* Conor McCrossan – A constant source of help and guidance for the project.

# Introduction

I choose my final year project because it used technologies I was personally interested in and I wanted to gain more experience with. I also wanted the opportunity to work as part of a project of the scale of ResponSEAble.

## Project Overview

“ResponSEAble is an interactive website that supports the development of cost-effective ocean literacy in Europe. It is an ambitious, 15-partner project and is funded by Horizon 2020. ResponSEAble is mapping European marine research and knowledge to further our understanding of complex human-ocean relationships and the economic benefits that we derive from our seas and the ecosystems they support” [1]. The interactive website currently enables users to select and view a story. If the user has admin rights they can also edit existing stories or create new ones. Due to the complexity of the stories they are presented in a very complicated manner. My goal is to develop an interactive, intelligent user interface for exploring the knowledge in the graph database, both visually and using queries.

## Problem Statement

Version Control Visualisation

Querying and Displaying Knowledge from a Graph Database. Working with a PhD student here, help develop a querying and graph visualisation tool for the OrientDB graph database. The database is being used in an EU project (http://www.responseable.eu/), and will contain a large amount of information which needs to be queried and displayed in an engaging way. You will use browser-based technologies (e.g. js, jquery, d3.js, HTML5, CSS, ajax, REST) to develop an interactive, intelligent user interface for exploring the knowledge in the graph database, both visually and using queries.

## Solution

The purpose of this project to create a develop an interactive, intelligent user interface for exploring the knowledge in the graph database, both visually and using queries. The UI should have a logical flow to its structure that makes it easy to read. Users will be able to visually explore and examine specific stories. The interface should inform the user of how much evidence is supporting each node and the causal links between nodes. This allows users to learn from the stories. Users can select specific start and end nodes on the graph and retrieve the causal evidence that supports them being linked.

# Research

## Tools

The decision on what tools to use as part of the project was extremely important. It was critical to ensure that the correct tools were used to increase the speed and ease of development but at the same time preserve the developer’s knowledge of the inner workings of the application. This section gives a better understanding of the tools used in the development process.

**NetBeans Ide**

Used for the coding/developing the UI. Used NetBeans in conjunction with GlassFish server to run the application. This editor supports all the necessary technologies and languages required for the project. It also allows for extensions to the editor though the use of plugins. NetBeans allows for easy and readable presentation of code and the decision to use it over other IDE’s was down to personal preference.

**GitHub**

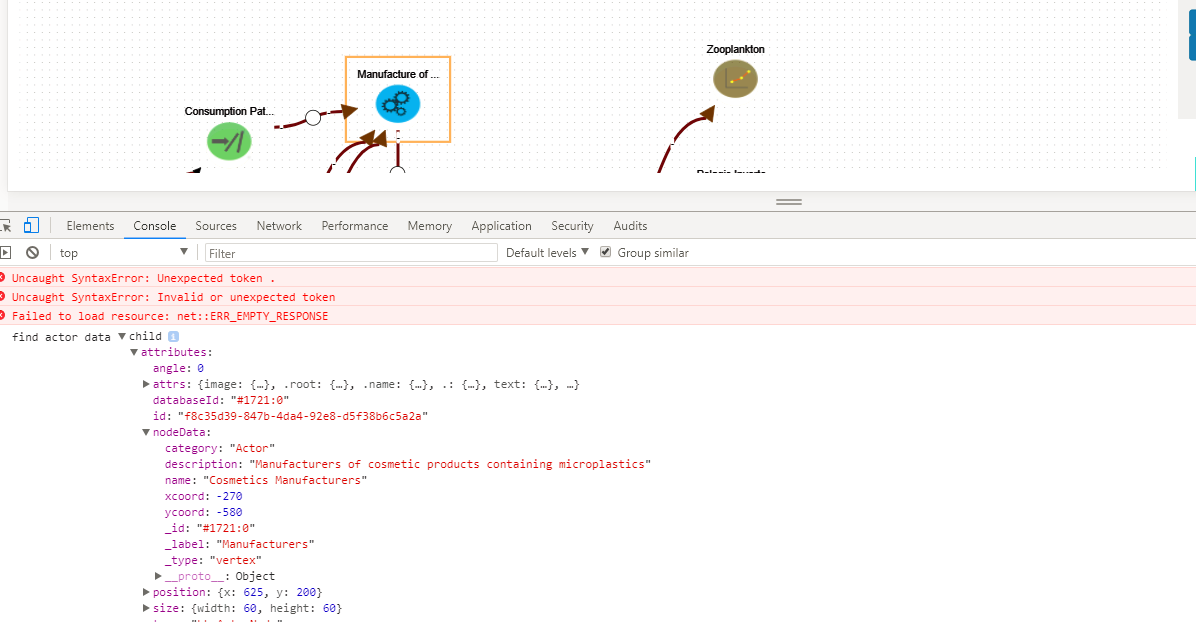
Used for version control and source management.

**Google Drive**

Secondary source of code backup in the possible event of breakdown of device that could result in the loss of code.

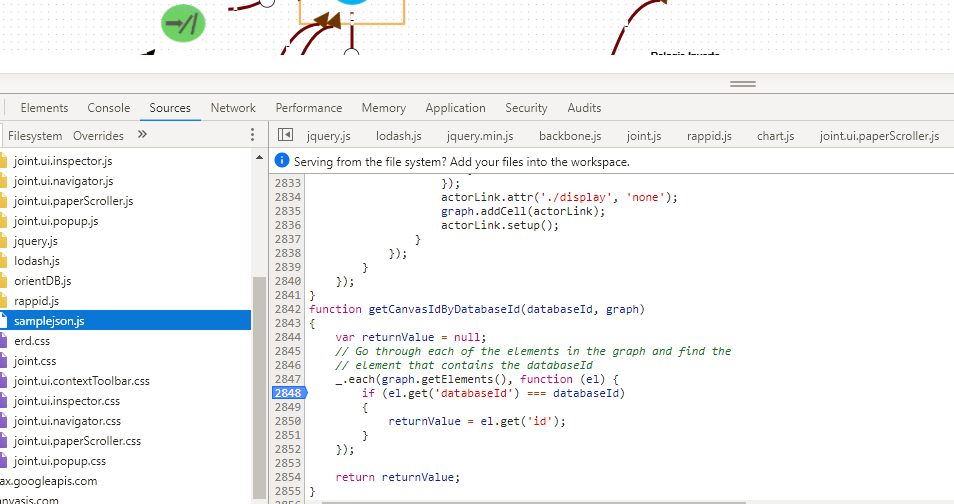
**Chrome and Dev Tools**

One of the most important tools used in the project development was Chrome and its Dev Tools. Their performance, feature set and usability is unapparelled. It’s the first point of call when an error occurs and it makes the sometimes-difficult process of debugging JavaScript that much easier. The time and effort saved by using the console tab to view simple things like the current structure of JSON data was invaluable.



*Example of examining the data contained in a node when it is rendered on the graph. Helped determine the path to access specific data points e.g. to access nodeData*

Another key feature of Chrome and its developer Tools was the ability to use the source tab to debug code in real time as you performed click events on the application.



*Example of debugging a component of the application within the projects source code with Chrome Dev Tools.*

## Languages

**HTML[https://www.w3schools.com/html/default.asp]**

HTML or Hypertext Markup Language is used to tell the browser how to display a web pages’ data to a user. HTML is supported across all major browsers which makes it the ideal language to display our content with. HTML was created to describe the contents of a page.

**JavaScript[https://www.w3schools.com/js/default.asp]**

Due to the non-static data that will be displayed on our web pages we need to use JavaScript in conjunction with HTML pages which allows for the creation of a dynamic and interactive application. JavaScript is stored in separate JavaScript files to make code readability easier. JavaScript is supported across all modern browsers.

**CSS[https://www.w3schools.com/css/default.asp]**

Cascading Style Sheets (CSS) describes how HTML elements are to be displayed on screen, paper, or in other media. CSS can be used to control the layout of multiple web pages all at once. CSS was introduced when tags such as <font> and colour attributes were added to HTML as a means to create a single source of styling instead of having to adding styling to every single page.

## Libraries/Frameworks

**JointJS[https://www.jointjs.com/]**

JointJS is JavaScript library that can be used to dynamically create static or fully interactive graphs and diagrams from JSON data retrieved using Ajax. JointJS was the main language used in the creation of the user interface. JointJS allowed for the creation of custom shapes via SVG and the rendering of potentially hundreds of elements and links with instant interaction. Zooming, animations and touch support are just some of the features of JointJS.

**SVG[https://www.w3schools.com/html/html5\_svg.asp]**

Scalable Vector Graphics (SVG) allows graphics objects, images and text to be rendered. SVG is XML based which means that every element is available within the SVG DOM and can attach JavaScript handlers to each element. In SVG, each drawn shape is remembered as an object and as a result if the attributes of an object is changed, the browser can automatically re-render the shape. SVG makes use of CSS for styling and JavaScript for scripting. The SVG library is included as standard in the JointJS Library.

**Chart.js**[http://tobiasahlin.com/blog/introduction-to-chartjs/]

Chart.js is a JavaScript open-source library that helps you easily visualize data using JavaScript. It supports 8 different types of chart and they’re all responsive. Chart.js is used to summarise the causal data between two nodes in the form of a bar chart.

**JQuery[https://www.w3schools.com/jquery/jquery\_intro.asp]**

JQuery is a free open-source JavaScript library whose purpose is to simplify the use of JavaScript on your website. The JQuery library contains the following features.

* HTML/DOM manipulation
* CSS manipulation
* HTML event methods
* Effects and animations
* AJAX

# Implementation

## Load JSON data

When a user selects a specific story to view the storyId is appended to the page url.



*StoryId appended as rid.*

The url is then examined and checked to see if the rid property is available

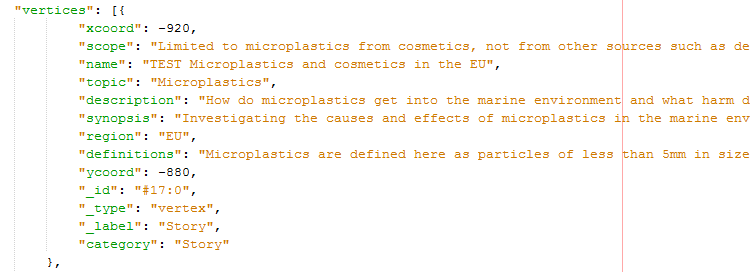


This passes control to the function that takes care of drawing the graph From here an ajax call is made from pre-defined function in the OrientDB.js file.

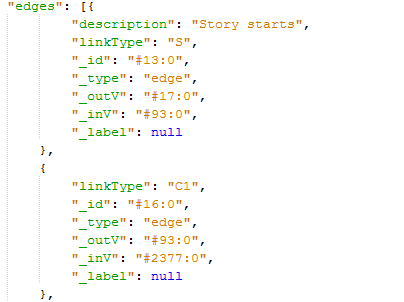


*Example of calling getStoryFromDB method*

As we can see from the screenshot above we need to pass the storyId to the function to call the correct story. The storyId represents the ‘databaseId’ of the very first element of the story. This attribute is represented by the ‘\_id’ property in the JSON data.



*Example of node JSON data returned from the getStoryFromDB () function.*



*Example of Edge JSON data returned by the getStoryFromDB () function*

## Display nodes

The first problem to be address was to create a method of displaying all nodes in a pre-determined order. In the current method of displaying stories the layout of the story depends entirely on how it was created and thus if the graph was created with little or no order it makes it very difficult to read. In the Project Definition Document, a Paper Prototype was created for the user interface for displaying a story. The key goal was to assign each Node type its own column. This was to help preserve the flow of the graph and make it more readable. It also takes the pressure off the creator of the graph, they don’t need to worry about creating the graph in a neat and readable format.



*Paper Porotype created in the original Project Definition Document. Each node type has its own column.*

When designing the order in which we expect the nodes to follow we operate under the assumption that graphs for the most part will follow the DAPSIWR (Driver 🡪 Activity 🡪 Pressure 🡪 State 🡪 Impact 🡪 Welfare 🡪 Response) format. This means that all links will follow a logical flow from one vertex type to the next. The image we see below looks visually appealing but its success lies in how the graph itself was originally built. There remains a responsibility on the creator of the graph to try preserve the flow and layout.



*Here we see that each node type has been assigned to a specific column. The readability of the graph has been improved dramatically. Note that all knowledge, actors and backwards links have been removed to further improve the readability of the graph.*

**How we achieved this**

There are 10 different categories of node that can be present in the JSON data. The first 8 node Types are identical in the method in which they are drawn.

1. Story
2. Driver
3. Activity
4. Pressure
5. State
6. Impact
7. Welfare
8. Response

The following is the method to draw the first 8 node Types. The other node Types are Actor nodes and Knowledge node which follow slightly different rules when being drawn. We will talk more about them later

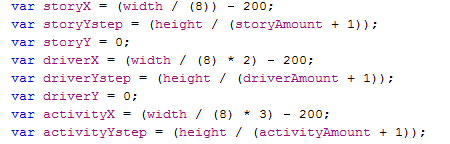
**Determine X and Y co-ordinates**

When the story is initially loaded from the DB using a pre-defined ajax call we cycle through the data to determine how much of each type of node is present. We cycle only through the vertex data as this reduces complexity.



*Code that cycles through the node of a graph to determine how much of each type is present*

The reason we do this is to allow us to determine how large our paper needs to be. As we can see from figure # nodes of type State are the most populous, from this we can then determine the max width of our paper. We want each node to have sufficient space to potentially display knowledge and not appear cluttered.



X and Y co-ordinates are assigned based on the width and height of the paper. We determine a step i.e. the vertical distance between each node type based on the number of that type of node and the overall height of the paper e.g. if there are 3 or one type of node and the paper height is 1000px then the node step will be 250. This means that each node will be placed with y-co-ordinate of 250,500 and 750 respectively. This means the nodes have the maximum space between each other possible.

**Draw Node**

For each category of node, we draw a custom JointJS shape. Each node type has the same underlying structure but with different attributes added upon creation. 

As you can see each time a node of specific type is created the y co-ordinate is incremented by the amount specified by the step value. The x co-ordinate remains the same as we want all nodes of the same type to appear in a vertical line. Each node has the shortened text value assigned to it that gives a brief description to the property of the node. The text was shortened to 15 characters as it helped declutter the graph.

As you can see as the node is created new attributes are added to it. These are the databaseId, nodeData, type, xcoord and the ycoord.

**databaseId –** This is the unique identifier that orientDB maintains to distinguish between nodes. This attribute is extremely important when we look to add edges to the graph as we require a method of determining what edges connect to what nodes. By adding the databaseId attribute here we can make is easily accessible later. We will talk more on this later.

**NodeData –** To this attribute we assign all data that is received for each node to that node on its creation. By doing this we have access to all node data in a singular place that is accessible within the DOM. Because of this it is very easy access node data at a later point. Without adding this attribute, we would not have all node data easily accessible at a singular location and would make the building of dynamic lists very difficult.

**Type –** Type is used an identifier to distinguish between the different type of nodes created e.g. knowledge, actors, storys etc. This is helpful later when we want to examine nodes in a graph and we know we don’t need to consider knowledge nodes, by having the type attribute defined we can ignore these nodes in our computations and saving resources.

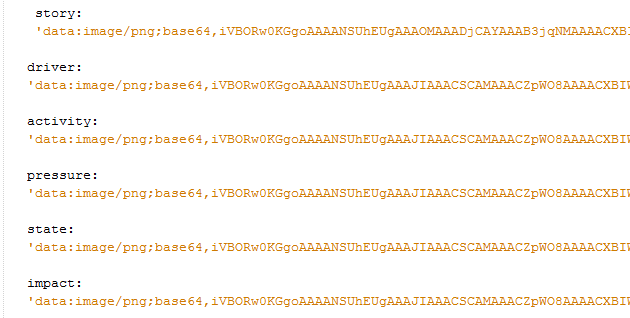
**X and Y coords –** As each node is drawn we record it x and y position. This is helpful later when we want to position knowledge and actor nodes relevant to each node. It again allows easy access to this data.

**Node Image**

Each Node category has its own specific image to represent it. Each image is mapped to a Base 64 string. From there it is mapped to a specific category name and called when necessary.



*Node specific images are loaded based on the category types*



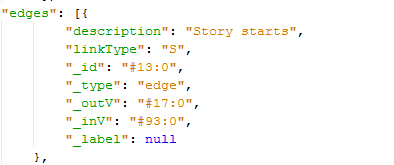
*Example of base64 images.*

## Draw Edges

Next we must draw all edges on the graph as they connect to the graph. Edges must be drawn after the nodes have been drawn as we need to obtain the canvasID to be able to pinpoint the location of the nodes on the canvas.

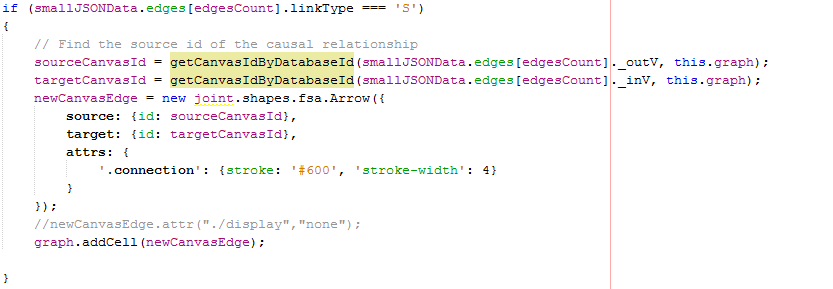
Like the nodes we loop through all edges in the JSON data. In the JSON Data there are 5 types of edges.

1. **‘S’** – this edges represents the link between a Story node and a Driver

**

*Example of JSON data for edge type ‘S’*

It is relatively straight forward to draw all we need is the canvasId of both the target and source node.



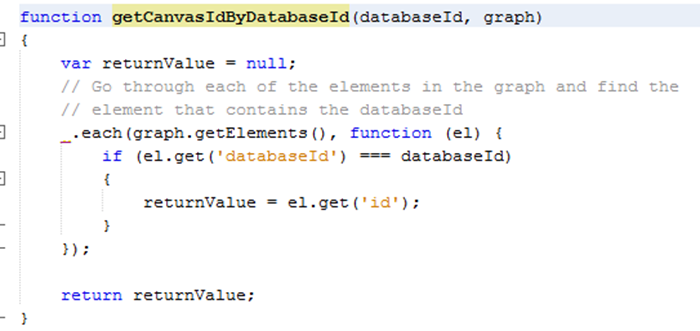
If the “linkType === ‘S’” we execute this piece of code. Since we know that all we require to draw this edge is canvasId of both the target and source node.

**Get Source/Target canvas ID**



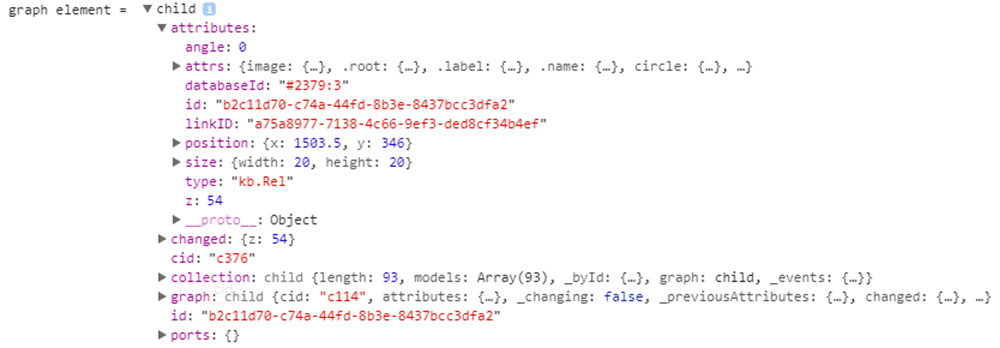
To find this we pass two parameters to the getCanvasIdByDatabaseId () function. First we pass the ‘\_outV or ‘\_inV’ databaseId property for this edge. ‘\_outV’ refers to the databaseId from which the edge originates i.e. source and ‘\_inV’ refers to the to the databaseId from which the edge terminates i.e. target. Next we pass the graph in its current state i.e. with all nodes drawn. This object has a reference to every element drawn on the canvas. We do this because the graph has a reference to the canvasId property of the drawn element which is necessary to draw the link.

Next we step into the getCanvasIdByDatabaseId () function



*getCanvasIdByDatabaseId () function*

In here we loop through each element previously added to the graph and check if the ‘databaseId’ property added when we created the node for the graph element equals the Id of the node from which our edge is originating from or terminating at. The ‘databaseId’ property is extremely important as it is our only way of verifying if the graph element is indeed the correct element from which our node originates/terminates and that we ensure we retrieve the correct canvasId. Without this property, we wouldn’t be able to correctly draw the edges between nodes. If the ‘databaseId’ from the graph element and our edge source or target match, then we retrieve the ‘id’(canvasId) of the element. The id property takes the following format id:"b2c11d70-c74a-44fd-8b3e-8437bcc3dfa2" This is a unique identifier for each element and is generated when each element is rendered. We return this value from our function and assign it to either the source or target variable. It is then used when drawing the new edge.

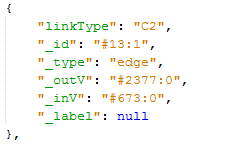


*Example of a graph element. The databaseId and id property are present and easily accessible.*

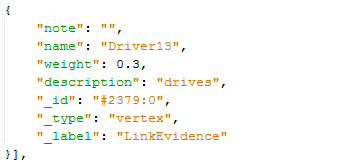
1. **& 3. ‘C1’ & ‘C2’ –** This represents the causal relationship between two nodes. This is a slightly more complicated edge to draw as you must understand the structure of the ResponSEAble graph. When we want to connect two nodes we want to be able to display causal evidence that supports the relationship between these two nodes. When a user creates a causal edge two parts are created, C1 which is the link from the source node to a special node called the ‘LinkEvidence’ node and C2 which is the link from the LinkEvidence node to the target node. When we draw the edge, we want just one edge to be drawn with the LinkEvidence node appended on top of the link. The purpose of the LinkEvidence node is to attach knowledge to further support the relationship between two nodes. Thus, C1 and C2 edges always come in pairs and map to a common LinkEvidence node.



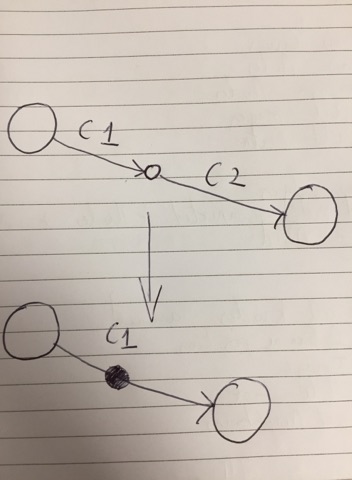
*Example of an edge of linkType C1*



*Example of an edge of type C2*

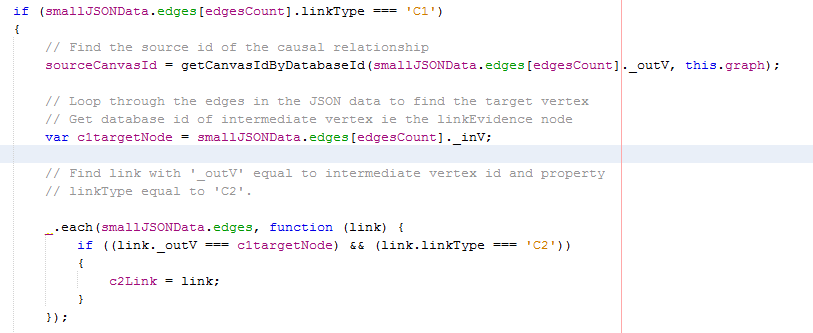
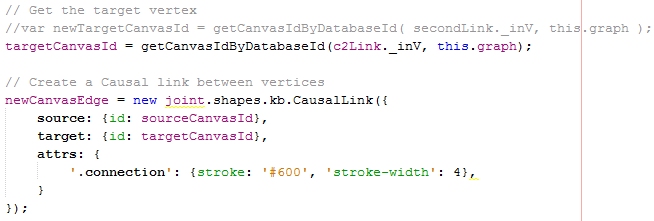


*Example of a LinkEvidence node*

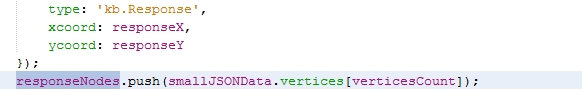


*This helps demonstrate what we want to draw. The top represents what we have in terms of JSON data and the bottom represents what we want to achieve.*

Like every other edge we draw we must find the canvasId of both the source and target nodes. However, with causal edges we need to find the source node of edge C1 and the target node of edge C2 and then draw the LinkEvidence node in the middle of this new edge.

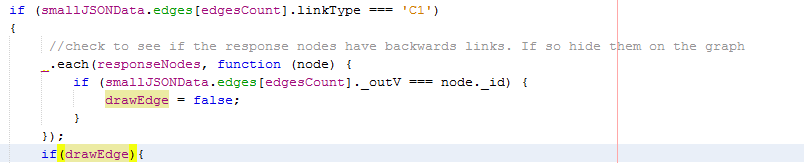
If we encounter an edge of type ‘C1’ we find the canvasId for the source node using the same function as before. Next we assign a variable called c1TargetNode to the databaseId of the target node for C1. Since we know C1 and C2 edges come in pairs and map to a common LinkEvidence node we can find the second half of the causal link by looping though the edges to find an edge of type ‘C2’ where the ‘\_outV’ is equal to the ‘\_inV’ of the ‘C1’ edge.

*Finding the target and source canvasId for a causal link*

Then we do a check to ensure there’re no backwards links i.e. links that don’t follow the DAPSIWR structure. We assume that only links originating from Response nodes will go backwards. To do this we create an array that contains all response nodes.

*If the node is of type ‘Response’ we add it to an array containing all response nodes.*

When we are drawing causal links we initially test to see if this link that goes backwards. We do this before we draw the edge to save computation in the case it is a backwards edge.



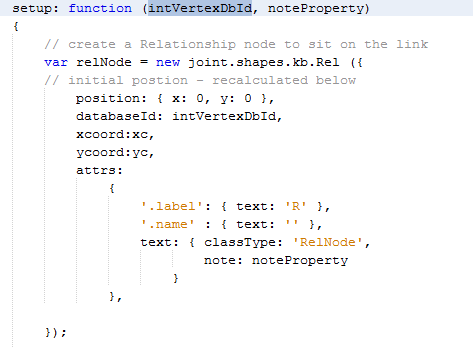
*Test to check if the \_outV of an edge is equal to the databaseId of a Response node. If it is, we set the drawEdge flag to false. The code will only proceed if the drawEdge flag is true.*

Once we have verified that the edge can be drawn on the canvas we draw it. However, once we add it to the graph we must append the LinkEvidence on top of it. After we add it to the graph we call its setup function and pass the LinkEvidence \_id to it. It is important that we pass this as we will require it later when we will to attach knowledge to it.



*Call setup function on newEdge and pass the databaseId to it.*

When we create the new node to represent the LinkEvidence we must add the databaseId of the LinkEvidence node so that we can reference this node later when we look to attach knowledge to the causal edge.



*Add the databaseId property to the new node when setup is called. Note this setup function was predefined, I merely added to it to better suit the needs of my project.*

The relNode is added to the graph. Next we need to find the co-ordinates of the source and target nodes of the causal edge. We can access these attributes as they are present in the graph element. 

*We can see here that we find the get the id of the source and target nodes and then retrieve their graph data which includes the x and y co -ordinates for each node. Note I modified this code from the existing codebase to better suit my needs.*

We use the position of the target and source node to determine the midpoint and the most suitable place for the relNode to be drawn.



*Example of causal link with relNode*

We can see here that the link between Fish Population and Impact is a causal link. The white node resting on the edge between these two edges represents the relNode and this is the point at which we will append Knowledge that supports the relationship between these two nodes.

There are 2 other types of links that are present and these are

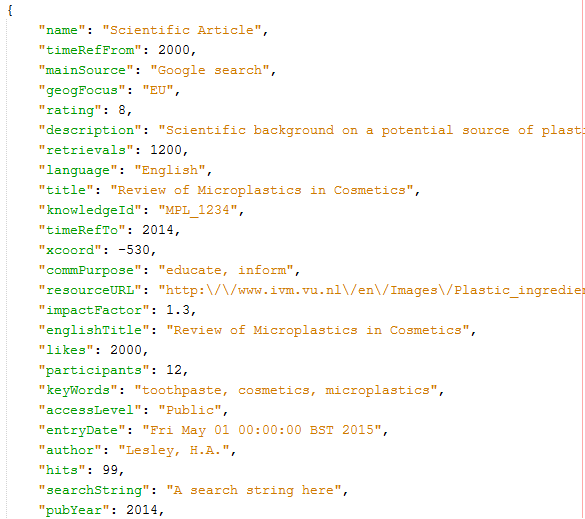
**4. ‘K’** - Knowledge Nodes

***5. ‘A’*** *-* Actor Nodes

We will talk more about these later we construct the knowledge and Actor nodes.

## Draw Knowledge Nodes & Edges

Here is the typical JSON data format for a Knowledge node



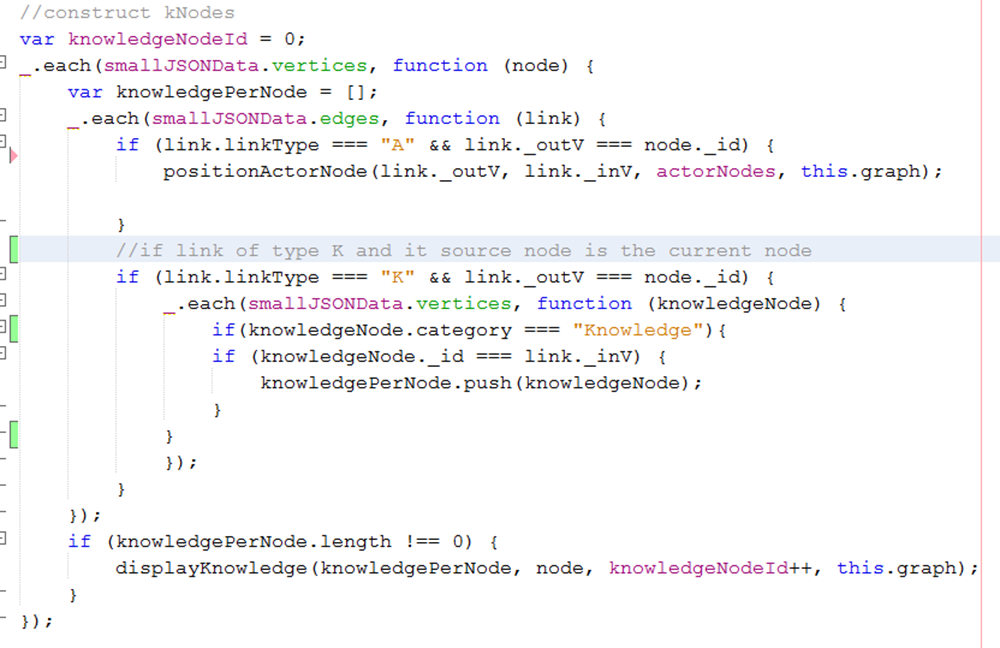


**Determine how many Knowledge nodes are attached to each node**

Knowledge nodes need to be drawn after all DAPSIWR nodes have been drawn so that we can access their canvas position and id.

We first need to determine how many knowledge nodes are connected to a single node. We do this by looping through all nodes in the JSON data and then looping through every edge in the JSON Data. If we find a link of type ‘K’(knowledge) who’s source is the current node (i.e. a relNode or a DAPSIWR node) we step into another loop that again goes through every knowledge node in the JSON data and determines if the id of the of the Knowledge node is equal to the target of the knowledge link. Essentially we first determine if a knowledge link originates from the current node and it that is true we then find the knowledge node at which that terminates. Because of this we can determine all the knowledge nodes that are attached to a single node. As we find the attached knowledge nodes we add them to an array called knowledgePerNode. Once all edges for a single node have been examined we send the knowledgePerNode array to a function that will sort and then draw all the knowledge nodes.

This is a very computationally expensive process to run as we need to examine every edge at every node. In large graphs this could have a detrimental impact on performance. For this reason, we determine what actor nodes are connected to each node in this loop also. We will talk more about that later.



*Code showing how it is determined how many knowledge nodes are connected to each node*

You can see before we test to ensure that the knowledgePerNode array is not empty before we send this to displayKnowledge. We also send extra parameters to the function.

**knowledgePerNode** - array containing the knowledge attached to single node

**node** – parent node upon which the the knowledge will be appended around.

**knowledgeNodeId** – due to the fact that multiple knowledge nodes could have to be combined into one element we needed to create a new unique identifier for the node. By incrementing this value every time we call this function we can gaurenttee the uniqueness of the id. We further add another digit later on to ensure this.

**this.graph** – We pass this so we can have an up to date copy of the graph at the time of drawing. We will need this to access the canvasId and position of the source and target nodes.

**Determine how many of each knowledge node type is present**

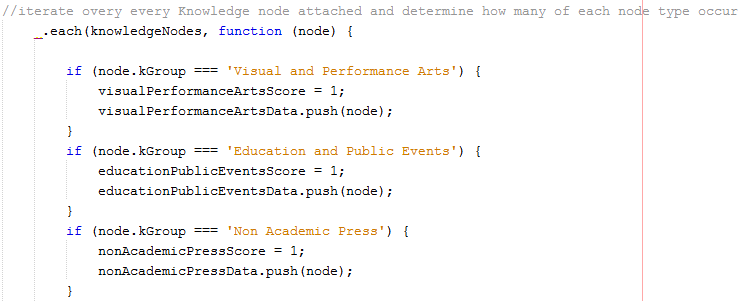
Once we pass this data to the displayKnowledge function we have to further separate the data. There are 8 different types(kGroup) of Knowledge nodes that can be returned in the JSON Data.

These are

1. Visual and Performance Arts
2. Education and Public Events
3. Non Academic Press
4. Scientific Print Media
5. Broadcast Media
6. Film
7. Other
8. Online Digital Media

Regardless of the knowledge node type the method for drawing them follows the same procedure. In the displayKnowledge finction we separate these knowledge types and how many of each type are present. For instance, there might be 3 articles belonging to KGroup ‘Visual and Performance Arts’ This means we have to combine these 3 articles of Knowledge and represent them with a single node. We also do this to determine how many how many different types of knowledge is present.

To do this we create an empty array for each knowledge type and then loop through the knowledgeNodes data.

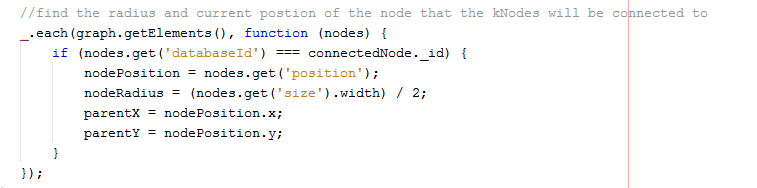


*Loop though the knowledge node data and add the node to the correct array depending on its KGroup*

The purpose of the visualPerformanceArtsScore = 1; is to keep track of whether this knowledge type is present. This is so we know how many knowledge types are present once we’ve sorted all the knowledgeNodes and can evenly space the knowledge nodes in orbit around their parent node.

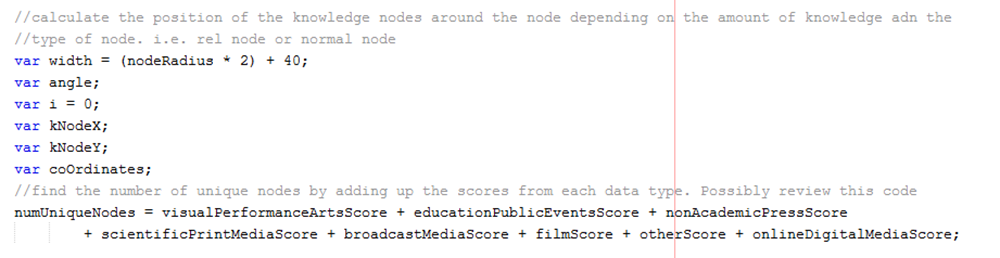
**Determine position of Knowledge node around parent**

To determine the position of the Knowledge node we must first determine the position of the source node. We do this by extracting the data belonging to the graph element whos databaseId matchs the id of the source node.



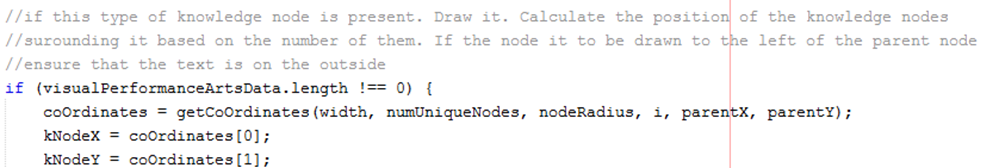
*Find the node postion and radius of the source node*

Node radius is an important attribute because it can vary in size depending on wheter it is a normal node or a relNode we are appending the knowledge to. Here we define variables necessary to find the orbit positions for knowledge nodes. We also find the number of unique knowledge nodes present by adding up all the score values.



*Define variables to find orbit positions and find the nube rof unqieu knowledge types*

Next we test each individual array to check if its not empty. The first step upon verifying its not empty is to find the co-ordinates for that knowledge node.



*Function call to get co-ordinates for each knowledge node*

The parameters we pass to this function are as follows

**Width -** width of the parent node. Important as the wider the node the greater the offset position will need to be.

**numUniqueNodes –** Count of the number of unique nodes. Determines the spacing between knowledge nodes around the orbit of the source node.

**i –** Keeps track of the number of knowledge nodes drawn. i.e. is this the first or last knowledge node to be drawn. The value of ‘i’ is updated as we leave the if statement.

**parentX –** X co-ordinate of the source node

**parentY –** Y co-ordinate of the source node.

Now that we are in the getCoOrdinates function we first find the angle (in radians) of the specific node. If it’s the first node out of 4 to be drawn, then the angle will be equal to zero. As the value of i increments the value for the angle will increase. Imagine the following scenario where we have four knowledge nodes. The angle values will be as such

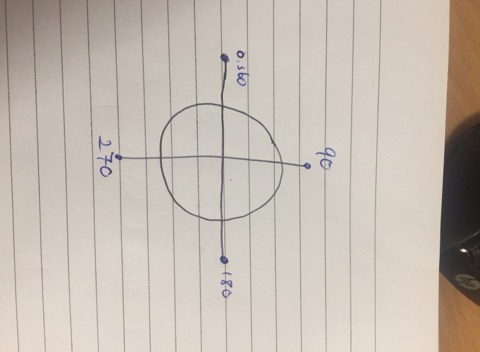
K1 = 0

K2 = (1/ (4/2) \* 3.14) = 1.57rad = 90°

K3 = 180

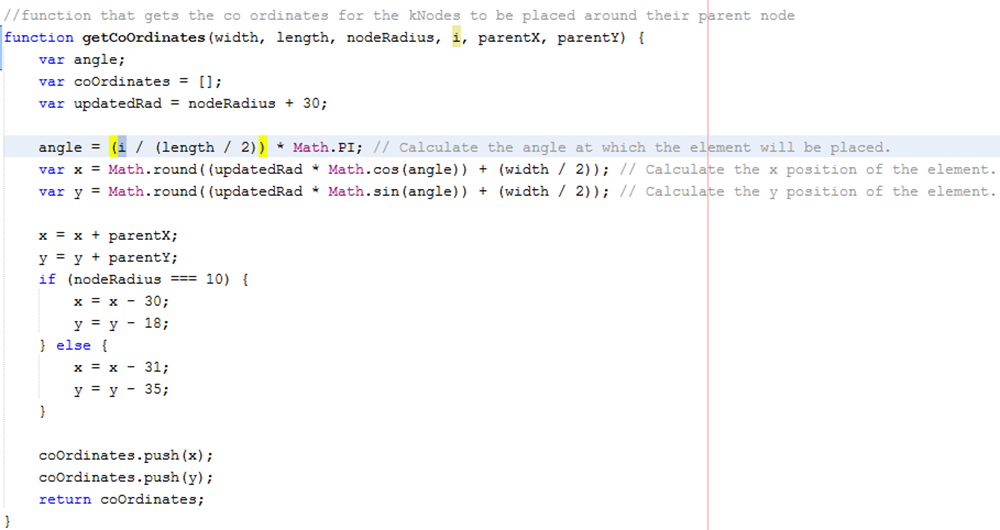
K4 = 270

This diagram helps illustrate it better.



*Illustration of where knowledge nodes will be placed if there’s 4 unique types present.*

We find the x and y position of each node using Sin(y) and Cos(x) to find the position at which to place the knowledge node. We then add this value to the parentX and parentY. Since the parentX and Y don’t represent the middle of a node on the canvas we need to offset the position of the knowledge node by a specific amount depending on whether it’s a normal node or a relNode. The values for the offset were determined through trial and error and are not very dynamic. If the width of the nodes is updated, then these values will also have to be updated.

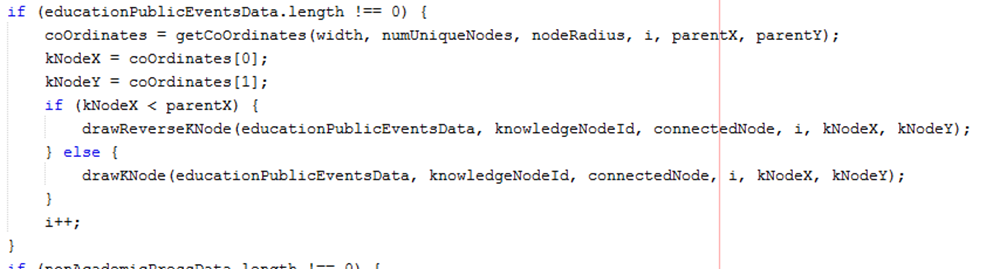


*Method to find knowledge node position based on parent size and position*

We then add the x and y co-ordinates to an array that is then sent back to the displayKnowledge function. Since we add x first it is added at position 0.

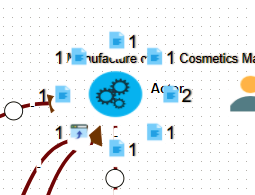
**Draw Knowledge Node**

Now that we have the co-ordinates at which our knowledge node will be drawn we can create and our knowledge node and add it to the graph.



*Code that calls the draw knowledge node function*

You can see here that we do a quick check to test if the x co-ordinate of our knowledge node is less than that of the parent node. If it is less (i.e. meaning it is drawn on the left-hand side of the parent node) we call the drawReverseKNode function. The only difference here is that we want the number representing the number of articles present in the graph element to be drawn on the left-hand side of the knowledge element. See below how knowledge nodes on the left of the parent node have the text to their left. This creates a more visually appealing look.



*Example of how knowledge is drawn around a node which it is connected to.*

We pass several parameters to the drawKNode/drawReverseKNode function.

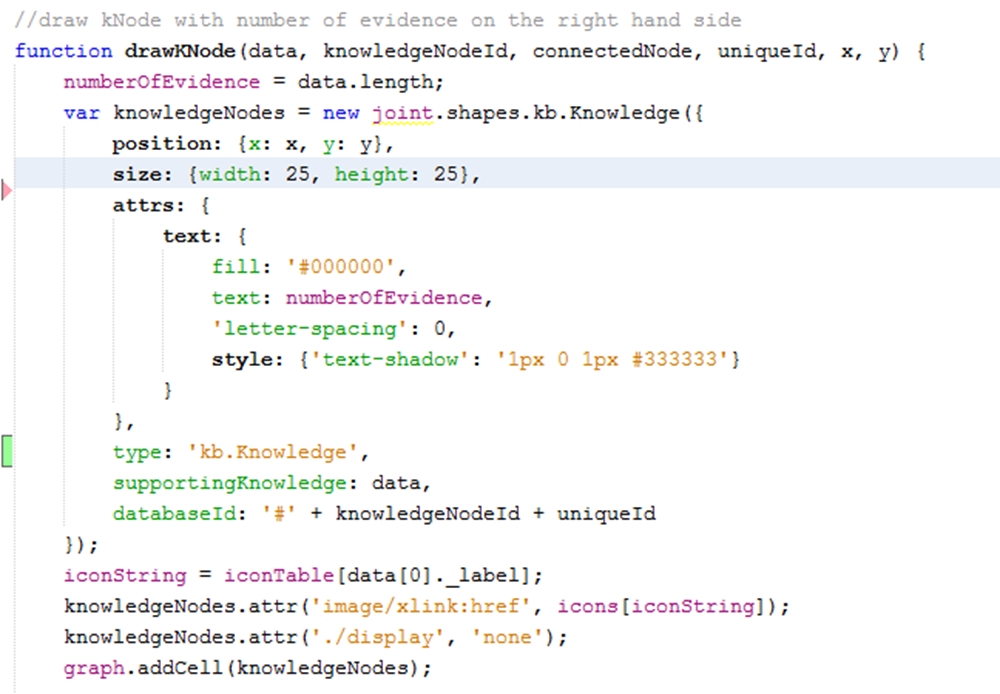
**educationPublicEventsData –** Array that contains all knowledge of a specific type. This could range from one knowledge article to many. We want this knowledge to be contained in the background of our knowledge for easy access.

**knowledgeNodeId –** this is the value passed from back when we added all knowledge for a single node to one array. This makes up one half of our unique id.

**connectedNode –** The node about which the knowledge will be orbiting or linked. We need access to its databaseId when we will be drawing the edge between the two.

**i -** This is the number of knowledge nodes that have been drawn around a single node e.g. 3 out of 4. This makes up the second half of our unique id.

Now that we have passed these parameters to the drawKNode function we can draw the new knowledge node.

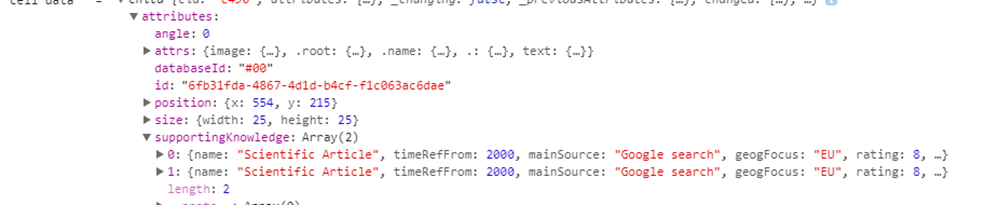


This node is drawn the same as any other. You can see we provide a the x and y co-ordinate and define several attributes specific to this node.

Notice that we set the value for text to be the length of the data array. This means that if there are 2 articles of knowledge present in the array then the user will be visually informed of the number of articles present.

**type –** Same as before this differentiates this node from other by giving it a specific type of ‘kb. Knowledge’.

**supportingKnowledge -** this contains the array of all knowledge behind the node. This array could consist of one knowledge article or many.



*Example of supportingKnowledge appended to knowledge node. 2 knowledge articles are present.*

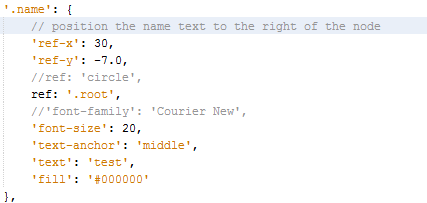
**databaseId –** like every other type of node this is a unique identifier. In the image, above we can see that the database id is maintained as #00.

**kNodeY & kNodeY –** The x and y co-ordinates for our knowledge nodes.

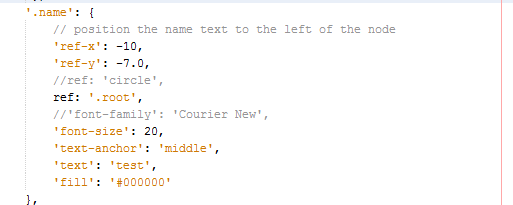
We find the correct image for this knowledge type in same manner as we do for all other node types.

We also originally set this node to display status of none. Meaning the user can’t see the knowledge node unless they wish to. This helps declutter the graph on original viewing. We then add the knowledge node to the graph.

Next we look at the element which we use to render the knowledge node, its design and layout is very like that used for the rest of the nodes. The only difference is how we display the text. i.e. the number of articles present behind a node. This code was taken from the original codebase and modified for my use.



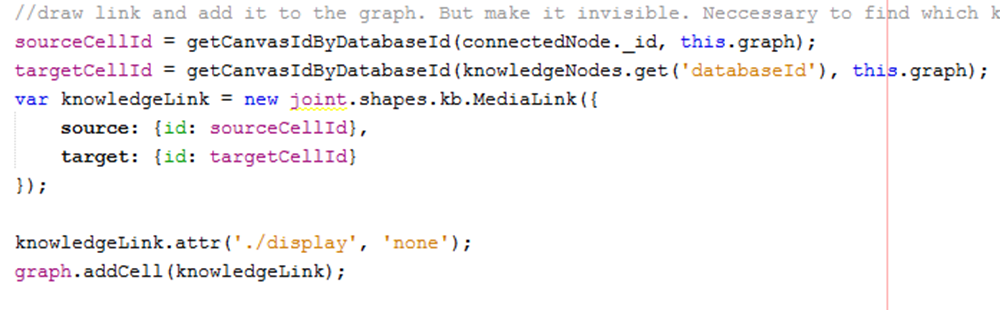
*In the normal kNode we position the text (number) to the right of the kNode.*



*In revererseKNode we position the text(number) to the left of the kNode.*

**Draw Knowledge Edge**

Once we’ve the Knowledge node drawn and added to the graph we draw the knowledge Edge connecting the knowledge node to the parent node.



*Code to draw knowledge edge*

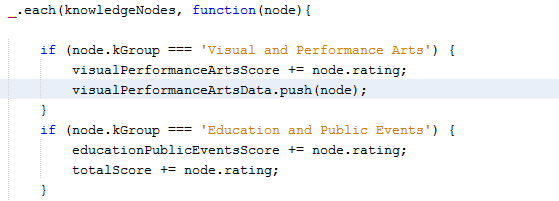
Drawing the knowledge edge is straight forward. We simply create our own edge whenever a new knowledge node is created. Since the new node doesn’t map to any JSON data we don’t need to loop though the JSON edges data and draw all edges of type ‘k’. Like with every other edge we pass in the source node id and graph data to the getCanvasIdByDatabaseId () to find the canvasId for the source and then similarly find the canvasId for the target.

We then set the display status of the knowledge edge to ‘none’. This hides it from the viewer.

Although the knowledge edge between a knowledge node and a parent is never shown it is still necessary to draw it and preserve it on the graph as we will use it later to redisplay the knowledge nodes when the user wishes to see them. It is vital for keeping track of what nodes are connected to other nodes. We will talk more on this later.

## Draw Knowledge Nodes (alternate method)

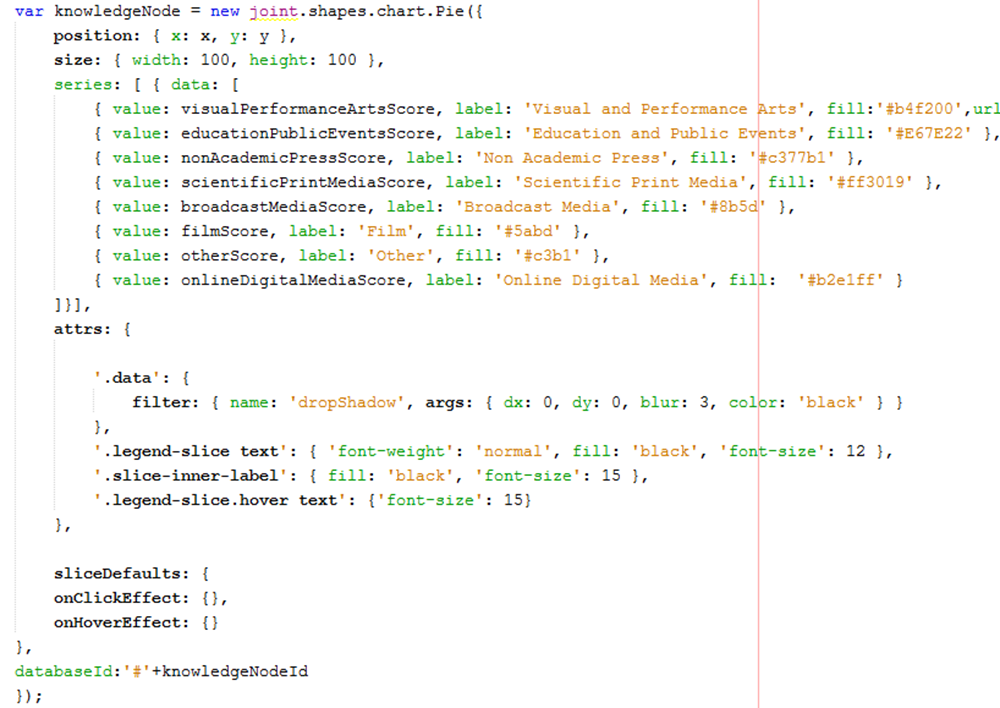
Another method considered to represent knowledge for each node was to use a chart.js pie chart to display the knowledge data. Like the method we ended up using, we determined how many knowledge nodes were linked to a single node and then further sorted the knowledge data by separating the data into its 8 types(kGroup). We combined the knowledge rating for each knowledge type and used this determine what percentage of the pie chart that knowledge type should represent.



*Determining the combined knowledge score for each Knowledge type*

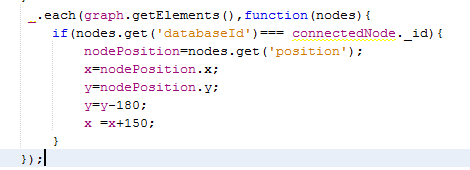
**Draw Chart.js pie chart to represent Knowledge data**

Next we drew the pie chart to represent the knowledge data.

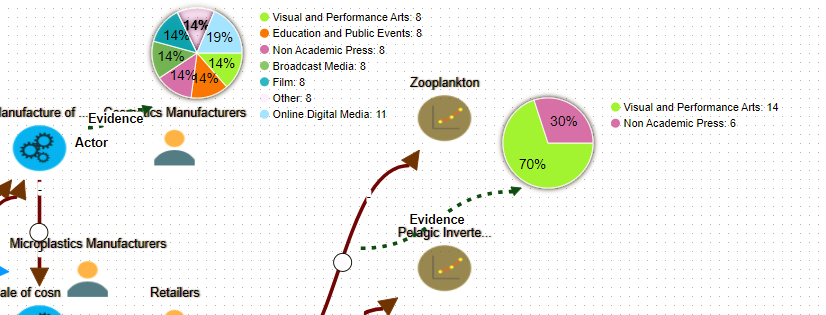


*Draw pie chart to represent knowledge node*

Since there was only one node to represent all knowledge data we didn’t need to find the orbit positions around the parent node we simply offset the co-ordinates for the new node in comparison to the parent.



*Find position for new knowledge node relevant to parent.*

We created a series value for the 8 possible types of knowledge data and the value assigned to them was equal to the combined rating for each knowledge data type. 

*Example of how pie chart looked for displaying knowledge data*

To prevent labels being displayed when there was no knowledge of a specific type present the following line of code was added to the updateLegend function in the chart.js file



*Prevent slices with value of 0 from being added to the legend*

Similarly, to prevent slices with values of 0 from being added to the pie chart the following line of code was added to the updateSlice function in chart.js



## Draw Actor Nodes

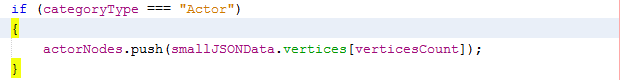
Here is the typical JSON data format for Actor nodes.



*JSON data for Actor node*

**Create an array that contains all actor nodes in the JSON data**

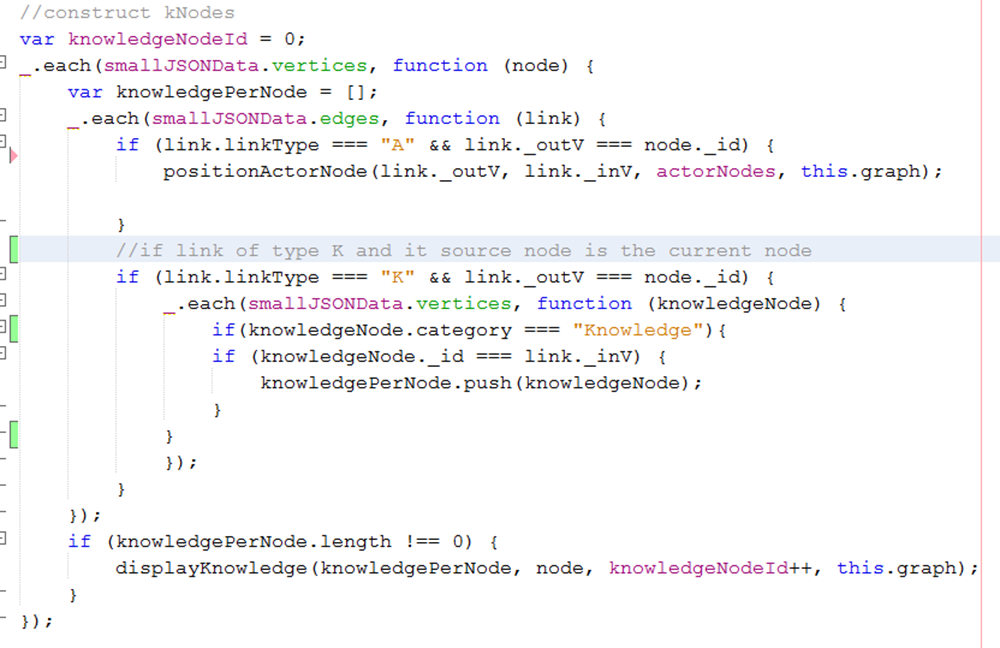
When we are looping through the JSON data vertices and drawing all the DAPSIWR nodes we add all actor nodes to an array for easy access later. It is easier to add all actor nodes when we are completing this loop as it saves on complexity by performing multiple functions in the one loop.



*Add all actor nodes present in JSON data to single array.*

**Determine if a node has a connected Actor node**

Using the same code and loops that we used to determine the if a node had knowledge appended to it we loop through all nodes in the JSON data and at every node we examine all edges. If an edge is of type ‘A’ (i.e. Actor) and its node id is equal to the id of the current node, then we know that node has an actor connected to it.



*Code that determines if a node has an Actor node connected to it.*

We then call the positionActorNode () function and pass it the following parameters.

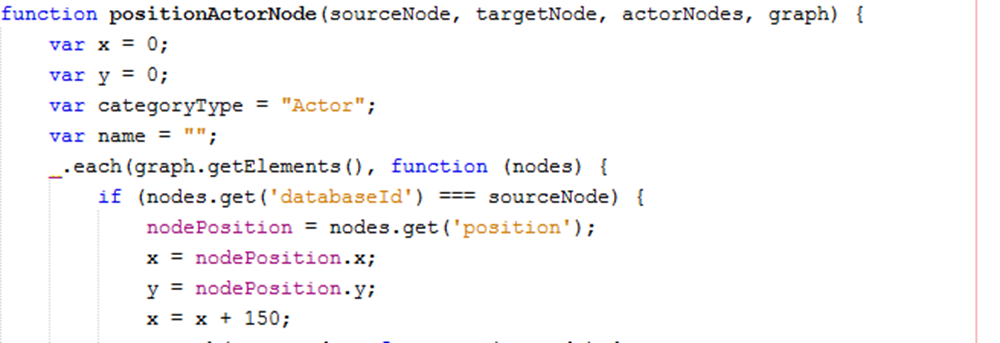
**link.\_outV** – The databaseId of the source node.

**link.\_inV –** the databaseId of the target node.

**actorNodes –** array that contains all actor nodes that are present in the JSON data.

**this.graph** – contains an up to date copy of all elements drawn on the graph.

From here we call the positionActorNode function. Like displayKnowledge () we find the position of the parent node. However instead of placing the Actor node in orbit around the parent node we just offset its x co-ordinate by 150. Since this function will only ever be called for Actor nodes we can safely hardcode the categoryType = “Actor” variable.



*Retrieve the position of the source node*

Next we loop though the ActorNodes array that we passed in and compare the databaseId of the targetNode and the ids of the ActorNodes in the array. If we have a match, then we know this which node the actorNode is connected to.

*Draw the Actor Node*

As we’ve done when we created every other type of node we added some custom properties to the node to allow for easy access to this data later when we want to view node data.

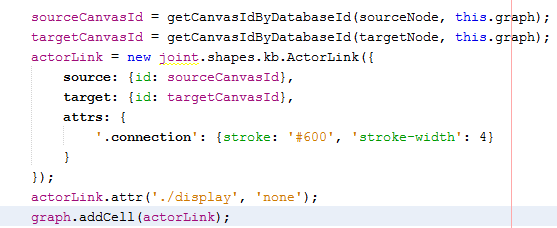
These are **type, DatabaseId** and **nodeData.**

The node image is added using the methods previously described.

Like with the knowledge node the Actor node is originally set with display status of ‘none’ so that it is invisible to the user upon initial page load. This is to help declutter the graph and try make it more readable.

**Draw Actor Edge**

Once we have drawn the Actor node we draw the Actor edge. This is very straight forward and simply requires us to find the canvasId of both the target and source node. Like the knowledge edge the Actor edge is initially set to display status of ‘none’.



*Draw Actor edge and set display status to ‘none’*

## Show connected Knowledge & Actor nodes when user clicks a node

**Determine what edges and nodes are connected to a single node**

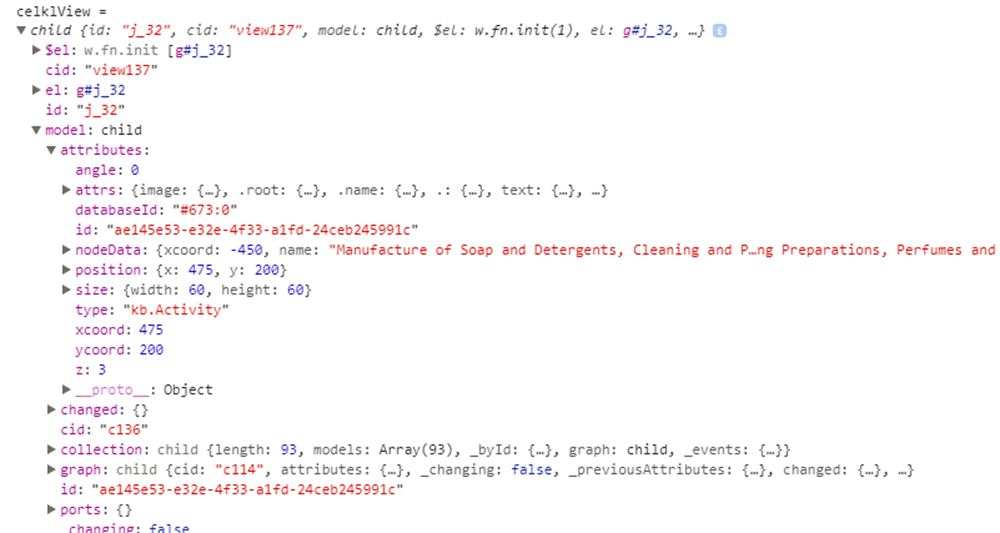
Initially when the graph is loaded the user sees only the basic graph with no knowledge or actor nodes display. When the user selects, a specific node be it either a DAPSIWR node or a relNode the connected Knowledge and Actor nodes should be displayed to the user for that node only.

We first need to register when a user has clicked a node element on the canvas.



*Detects when a user clicks on a node(cell) on the paper*

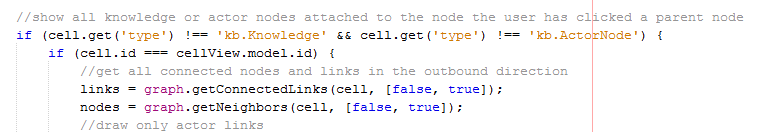
This method returns to us the cellView as seen on the paper. This contains all data the data that we appended to this node when it was drawn. We can see that it holds a reference to its databaseId, nodeData, xcoord, ycoord which are all property’s we added upon creation of the node.



*Data contained in cellView*

Now we need to find what nodes and edges are connected to specific node if we want to find all connected knowledge and Actor nodes. To do this we first need to find the cell on the graph. We do this by looping through all cells in the graph and comparing their canvasId with the canvasId of the node we selected using cellView.

*Loop through all cells in the graph*



*Compare the cell canvasId with that of the cellView canvasId and find the connected edges and nodes*

Since we know the user won’t be clicking Knowledge of Actor nodes when displaying connected nodes, we don’t need to consider these in our comparisons and can save computation time. The reason we need to find the graph element is because JointJS has built in methods for finding the connected Nodes and connected Edges for a node and we require the graph cell to use it. We call two functions to find the connected edges and nodes.

**graph.getConnectedLinks() –** Function that retrieves all edges that are connected to a single node.

**graph.getNeighbors() –** Function that retrieves all neighbouring nodes. (i.e. only one edge away)

We pass these functions two parameters.

**cell -** The node for which we want to find the connected edges and nodes.

**[false, true] -** This maps to the following [inbound, outbound]. By setting the second flag to true we retrieve only outbound edges and nodes. This helps reduce complexity.

The graph.getNeighbors() function is the reason we still must draw the knowledge edge even though the user never sees it. For this function to work there must be a link between nodes that can be traced on the graph.

**Show Knowledge and Actor Nodes**

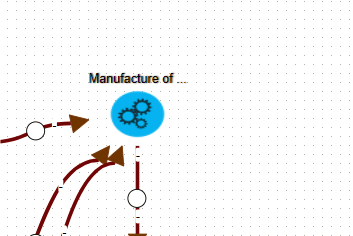
Once we have an array of all connected nodes and edges we can loop over each array.



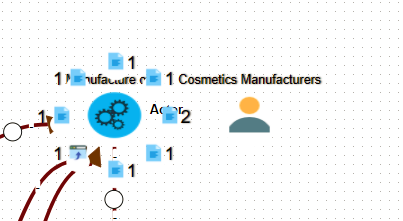
*Loop over each array and draw the connected nodes and edges*

When we are looping over the edges array we only want to show edges of type ActorLink. This is where the type property defined on the creation of edges and nodes comes in useful. We can differentiate between edges using this attribute. We first check the current display status of the ActorLink and if it is currently set to ‘none’ (the initial value) then we set it to ‘block’ (i.e. visible). Similarly, if the display status is block then we set it back to ‘none’. This allows the user to toggle the display status of Knowledge and Actor nodes around a node.

Similarly, when updating the display status of Knowledge and Actor nodes we loop through the nodes array obtained by the getNeighbors () function. We again use the ‘type’ property defined on node creation to differentiate between node types. In this case, we want to update the display status of both Actor and Knowledge nodes. The procedure is the same as for edges as we allow the user to toggle the display status.

**

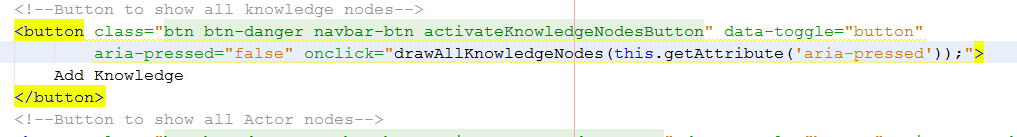
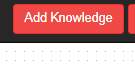
*Before click with no Knowledge or Actors displayed*



*After click event with both Actor and Knowledge nodes visible.*

## Draw all Knowledge Nodes on Button Toggle

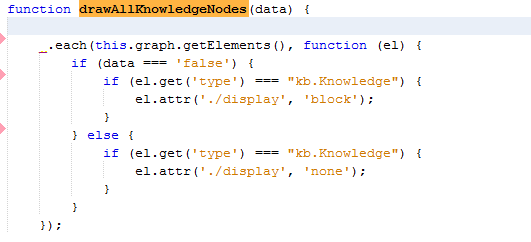
A user may wish to view all knowledge on a graph with a single click. This action is very similar to the action performed when we click on a node. For this purpose, a toggle button was added to the header that when the user clicked it forwarded control to a JavaScript function that toggled the display status of all Knowledge nodes.



*Bootstrap button that on click displays all Knowledge Nodes*

When the user clicks this button, it activates the drawAllKnowledgeNodes function and passes it the status of the toggle button. (i.e. clicked or not).

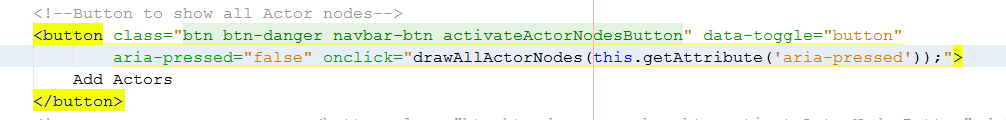
In the drawAllKnowledgeNodes function if the status of the ‘aria-pressed’ the attribute is false (status will be false on initial click. i.e. when user wants to display knowledge) we cycle through all nodes currently on the graph and if the ‘type’ is ‘kb.Knowledge’ then we set the display status for that knowledge node to ‘block’. This is again another example of the type attribute defined on node creation coming in handy. Similarly, if the user toggles the button (i.e. wants to turn the knowledge off we set the display status of all knowledge node to ‘none’).



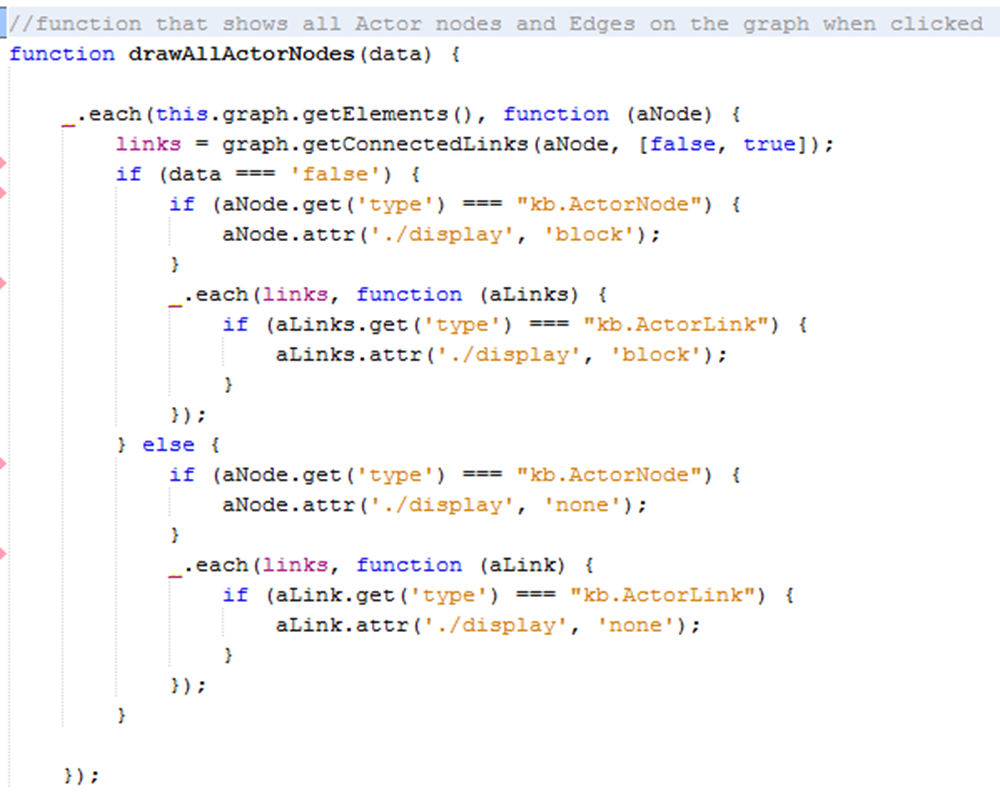
*Display/Hide all Knowledge nodes in the graph on button toggle*

## Draw all Actor Nodes and Edges on Button Toggle

A user may wish to view all Actors on a graph on a single button click. This is very similar to the action performed when the user wishes to show all Knowledge on a graph. The only difference is we must use the getConnectedLinks function to find the connected Actor Links and toggle their display status. We create a bootstrap toggle button on the navbar that when clicked activates a JavaScript function that toggles the display status of all Actor Nodes and Edges in the graph.



*Bootstrap toggle button that when clicked activates drawAllActorNodes function and toggles the display status for Actor edges and nodes.*



*Toggle display status for Actor nodes and edge.*

You can see we cycle through all nodes in the graph and get the connected links for each node. If a node is of ‘type’ ‘kb.ActorNode’ then we toggle its display status, similarly for edges if they are of ‘type’ ‘kb.ActorLink’ we also toggle their display status.

## Display Data relevant to each node

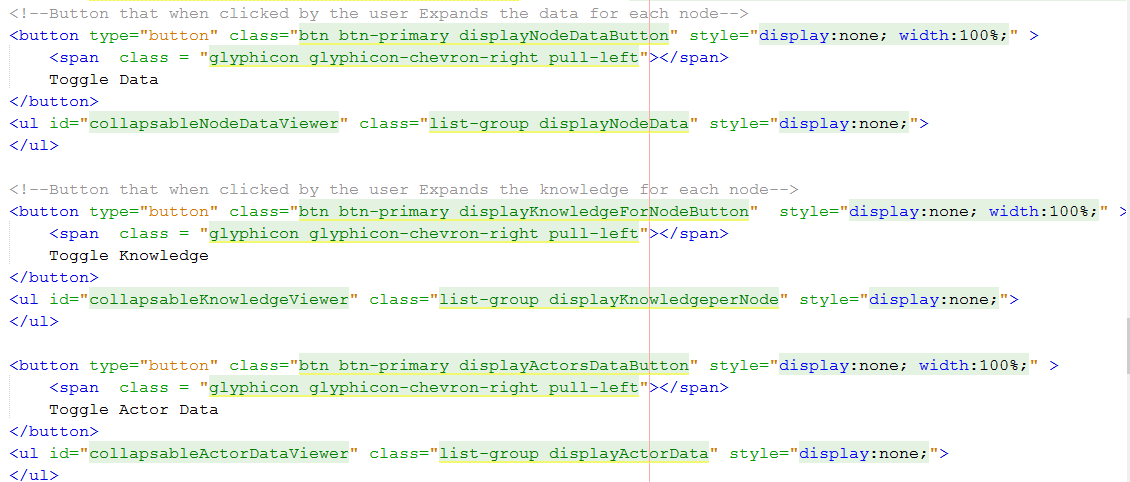
When a user selects a single node, we want to present to them all the data that is present to support this node. This data is broken down into 3 categories.

**Node Data** – This is the data that belongs to the actual node the user select. i.e. node name, description, category etc.

**Knowledge Data** – If the user has selected a node that has knowledge connected to it then we want to display this to the user. E.g. one or more of the 8 categories of knowledge are present supporting the node.

**Actor Data –** Similarly if the user has selected a node that has Actors connected to it then we also want to give the user an opportunity to view this data.

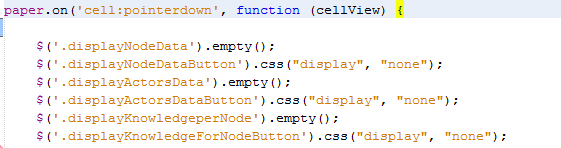
All data is dynamically appended to bootstrap list-group dropdowns that are populated as the data is read. We initially create bootstrap buttons in our index.html file and append the parent ‘list-group’ for each data type. The buttons are initially set with a display status of ‘none’ and only become visible to the user if there is data present to populate them.



*Bootstrap buttons that contain parent ‘list-group’*

**Retrieve node Data**

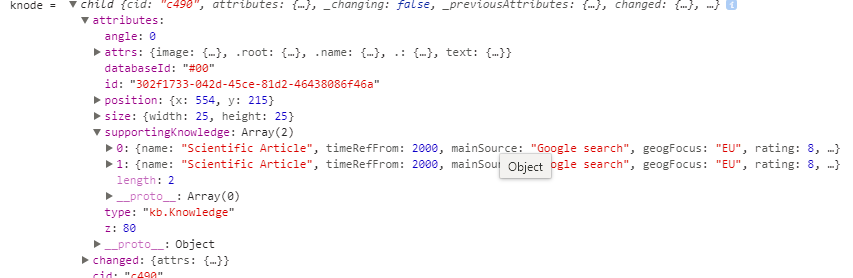
Like when the user selects a node and we toggle the display for all connected nodes when a user selects a specific node we also want to display all data relevant to this node. When a user selects a node we initially hide all buttons and empty all ‘list-groups’. This is to remove information relevant to a previous node and to also hide data categories that might not be relevant to the current node. E.g. the user may have preciously selected a node that has knowledge data and the current node does not.



*Use JQuery to hide buttons and empty divs*

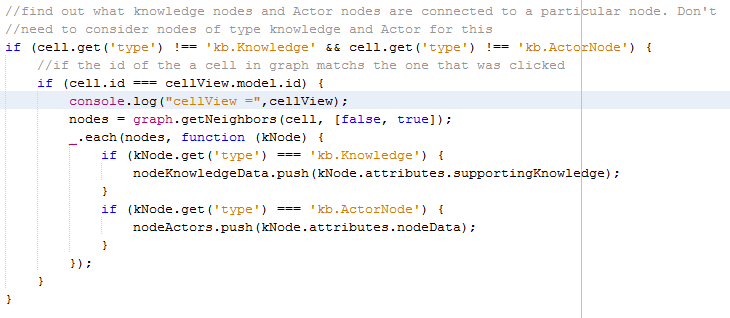
Once we have reset our list-groups for a new data set we simply cycle through all graph cells and compare their canvasId id with that of the node we selected. Once we have that we again use the getNeighbors () function to get all connected nodes. This is very like the process involved to toggle the display status for individual nodes.

Once we have an array of connected nodes we cycle through them and if any knowledge nodes are attached we extract the supportingKnowledge behind each knowledge type. To do this we access the ‘supportingKnowledge’ property defined when we originally created the knowledge node which contains all the JSON data for each category type.



*Supporting knowledge is maintained in each knowledge node and easy to access*

For each Knowledge node connected to the node the user selected we add the supporting knowledge to an array that we can then pass to our function that will populate the drop-down lists. We also add data relevant to connected Actor nodes to separate array. We similarly access the nodeData property defined on the Actor nodes creation. This contains all JSON data relevant to each actor node.

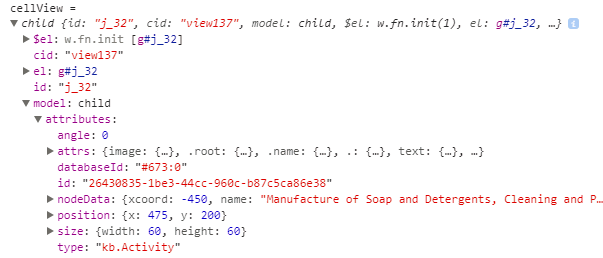


*Find all connected Knowledge and Actor nodes and add them to separate arrays*

We also access the nodeData and nodeType. Again these are properties that were defined and populated on node creation. Because these properties are accessible from the cellView we don’t need to find the graph cell to access them.

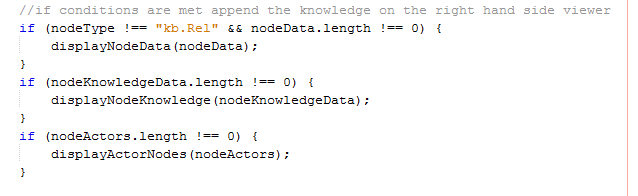


*Access nodeData and nodeType*



*Accessing nodeData and nodeType from cellView*

Once we have constructed our arrays with all connected data categories. We do a test to check if each array has data in it and if so we forward control to the appropriate function to populate the lists.



*Check to ensure array aren’t empty and call appropriate functions*

Due to relNodes or LinkEvidence nodes not having substantial data to support them we don’t display node data for them.

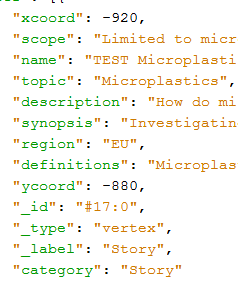
**DisplayNodeData**

We pass only nodeData to the this function. We first set the visibility status of the toggle button to visible using jQuery to access the correct button and change its property.



*Using JQuery to update button status and dynamically append each JSON pair to the list-group.*

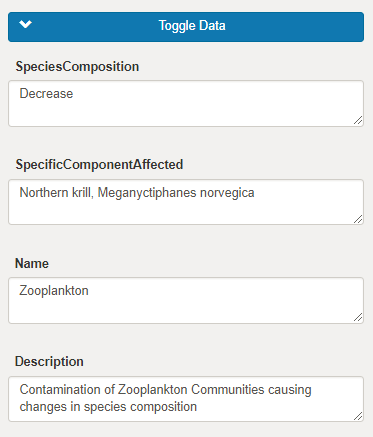
We filter out data properties \_type, \_id, xcoord and ycoord that won’t provide the user with any relevant information.



For JSON data of the above format “for (x in nodeData)” allows us to access each key value pair that is present. For the first line above x would represent “xcoord” and nodeData[x] would represent -920.

This allows us to iterate over all key value pairs and append each key value pair to the list-group. This means the JSON data format could change in the future yet still display all attributes.

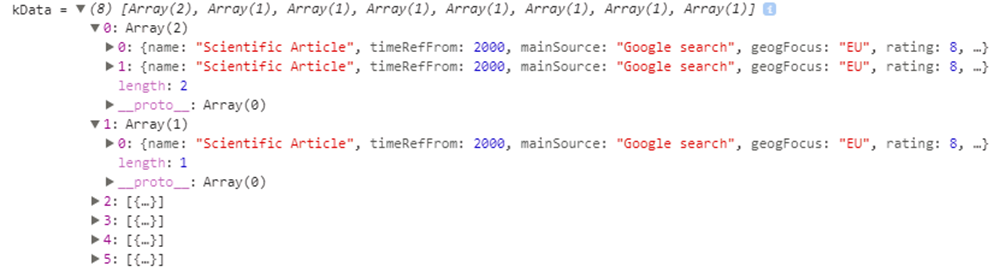
We use the value of ‘x’ as the label for our form group and use the CSS styling attribute ‘text-transform: capitalize’ to capitalize the first letter.



*Node data displayed in drop down menu.*

**DisplayKnowledgeData**

We only pass Knowledge Data to this function. This is slightly more complicated to append data to the dropdown list as we could have multiple arrays within the original array. Like with the displayNodeData we initially set the display status of the button responsible for containing this data to visible using JQuery to edit the CSS display property.



*Format of array as it arrives in DisplayKnowledgeData*

We can see that the fist value of the arrays contains multiple values. i.e. this knowledge type has more than one supporting article.

We first iterate over the outer array and use JQuery to append a new list group item for each category type present. We use the name of the knowledges kGroup (e.g. Scientific Article, Film, etc.) to distinguish between list groups. We also attach a badge that visually informs the user how many articles of this type are present at a glance.



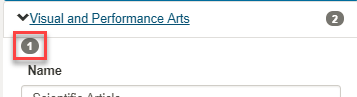
*Informs user of the knowledge category and the number of Articles present*

Within this new list group item, we then append a list group of type collapse that will contain all the data for this knowledge type. We can reference this specific list group when appending the JSON data by the “knowledgeItemNumber+i+“ class name we assign it. The value of ‘i’ increases as each new list is created ensuring each list has a unique class name and contains the correct data.



*Dynamically creating collapsible list groups that can hold Knowledge Data*

Once we have defined our list we then step into the inner array and iterate over it. We first append a badge that visually informs the user the current article they are viewing (e.g. article 1,2,3 etc.).



*Inform user that they are viewing the first article*

We then use JQuery to reference the “knowledgeItemNumber+i+“ list previously created. We can then use the same method used in DisplayNodeData to append the key value pair JSON data dynamically.

**

*Iterate over inner array and dynamically append each JSON key value pair to the list-group*

Like displayNodeData we filter out some data properties as they don’t provide relevant information to the user. Except for the resourceURL property we append all key value pairs into a text Area. With resourceURL we append a ‘href’ instead so that when the user clicks it they are brought to a new webpage that loads the knowledge article.

After each iteration of the inner array we insert a line break to further distinguish between articles.

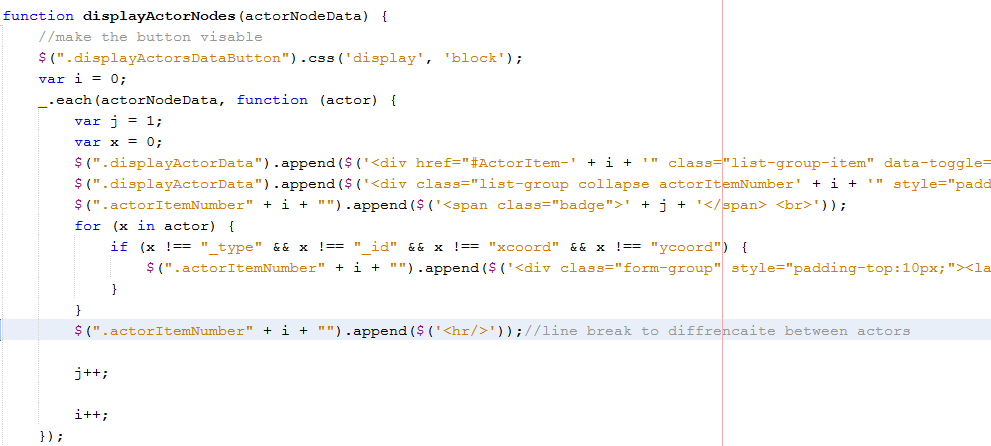
## DisplayActorData

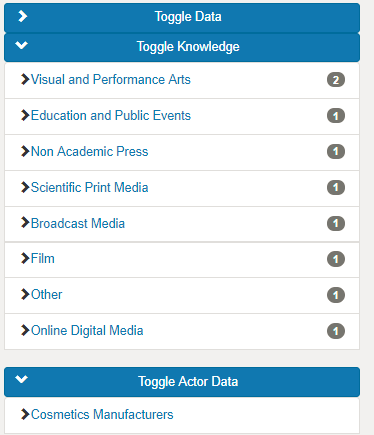
We send only Actor data to this function. Like with the other functions we first set the button display status to visible using JQuery. This follows a similar procedure as DisplayKnowledgeData as we could have many actors connected to a single node but is not as complicated as there won’t be any nested arrays like there was with DisplayKnowledgeData.



*Format of data received by DisplayActorData function*

The procedure to append Actor data is the same as in DisplayKnowledgeData. We iterate over each actor node and create a unique list-group to hold the Actor Data. We then use JQuery to access our unique list and append each key value pair while filtering out irrelevant data.

*Append Actor data to list-group*

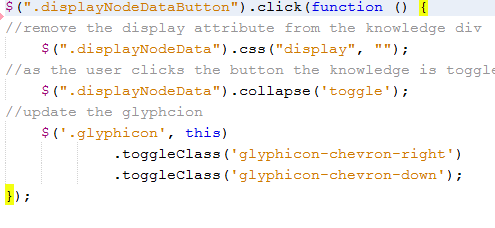


*Node Data viewer after all data has been appended*

## Detect User click on button and lists

We use JQuery to detect when a user clicks the buttons and lists that contain data relevant to the selected node.

**Detect Button Click**



*Detect Button click*

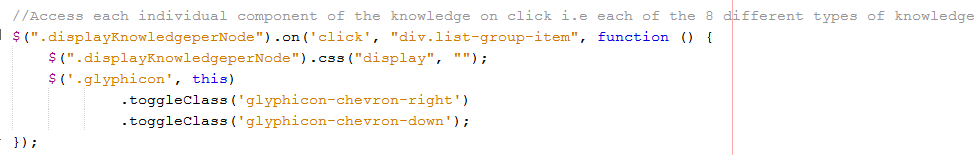
Using JQuery we can detect when a user selects specific button. When clicked, the following actions occur

* We remove all display attributes because if they are left in the cause flickering to occur as the user toggles the button.
* Toggle the button collapse.
* Change the glyphicon class to represent the current status.

We use the same method to detect all button clicks for displaying data relevant to a specific node.

**Detect list-group clicks**

When a user clicks on an element contained within the button class it is slightly more difficult to detect as we need to provide JQuery with a more specific path to the element due to it being dynamically created and the class name is not present in the HTML file.



Once we register this click event we perform the same actions as we did for the buttons in the previous step. This click event also toggles the list-group.

## Display Causal Summary

When the user selects two nodes on the graph all causal knowledge that links these two nodes should be found displayed to the user so they can view the knowledge that supports the connection between nodes.

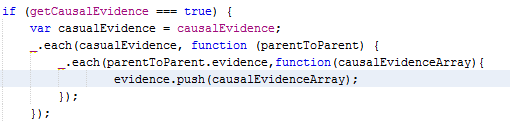
**JSON Data returned by ajax call**

**

*Format of JSON data returned for getCausalEvidence*

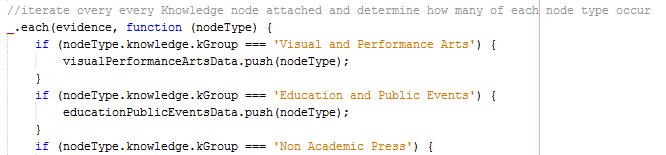
**Extract and sort Evidence from JSON data**

We first iterate through all evidence in the JSON data returned and add it all to one array. This is so that we have all returned evidence in one location.

**

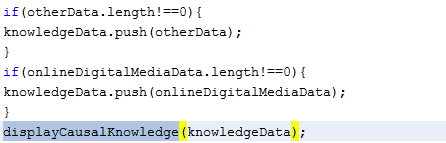
*Extract all knowledge returned in JSON data*

We then separate the evidence into separate arrays based on its knowledge category. This is very like the method used earlier when we separated the knowledge into different categories when we wanted to draw the knowledge nodes. The only difference is down to the JSON structure meaning we cant reuse the same code.

**

*Separate knowledge based on category*

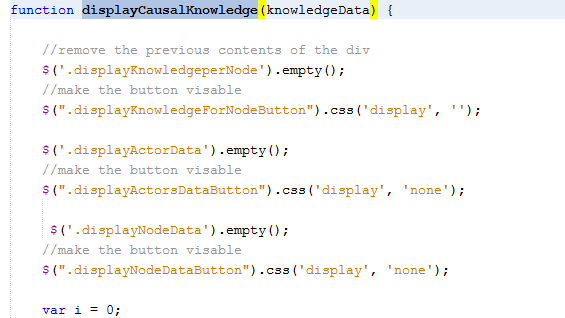
Once we have separated the knowledge based on its category we check to see if the array its contained in is not empty and if so we add it to a single array which we then send to the displayCausalKnowledge function to display the causal data.

**

*Add knowledge type to single array and send array to displayCausalKnowledge*

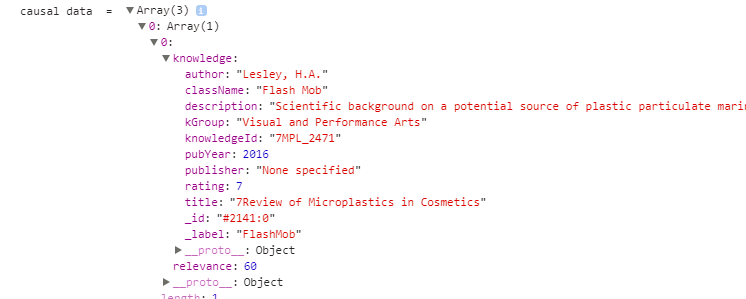
**Display Causal Knowledge**

We first ensure any data displayed before the causal Summary was generated is removed from the viewer and all buttons are hidden. Note for displayKnowledgeForNodeButton we only clear the CSS display attribute as we append the knowledge to this button.



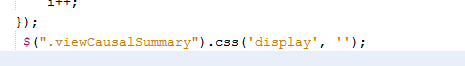
*Clear and hide all previous data*

We use nearly the same method to dynamically create and append the causal evidence to the list-groups as we used previously for when the user clicked on a node. The only difference is the structure of the JSON data meaning we can’t reuse the same code to perform this task for us.

**

*Structure of JSON for causal Data. Notice the extra ‘knowledge’ step*

We also set the status of the button reposnible for loading the summary chart to visable at the end of this function.

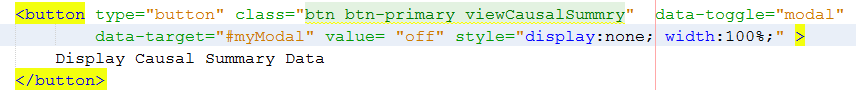


**Draw Summary Graph for Causal Data**

We also want to present the user with an option to get a quick view of the causal data supporting the connection between nodes. For this purposes we used a chart.js bar chart to display the number of articles present in the causal evidence.

**Create Button and Modal**

We first create a toggle button in our HTML file that the user will click when they want to display the causal summary. The display status is initially set to hidden as we only want this to appear when a causal evidence summary can be viewed.

**

*Bootstrap button that user clicks to view causal summary*

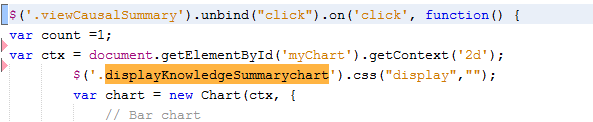
When the user clicks the button, they trigger a model window to render.



*Render modal window that will display the graph*

**Draw Graph**

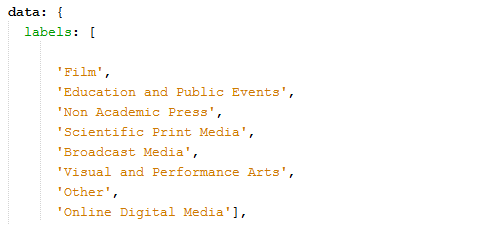
At the same time when the user clicks the viewCausalSummary button we use JQuery to detect the click, draw the chart and append it to the modal window that we just opened.



Use JQuery to detect button click and append new Chart to the canvas contained in the modal.

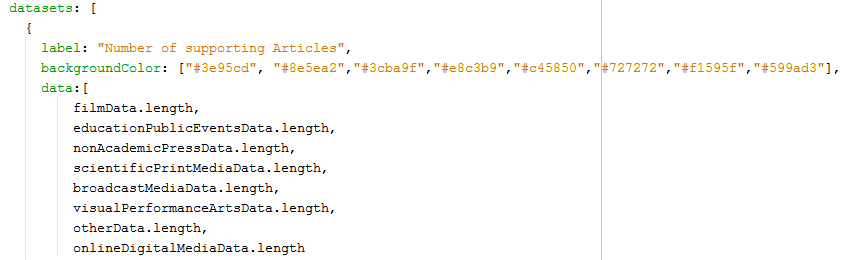
This method is still present in the displayCausalEvidence function so we have access to the causal evidence that has been sorted by category.

We now define our 8 different types of categories to be on our x-axis.

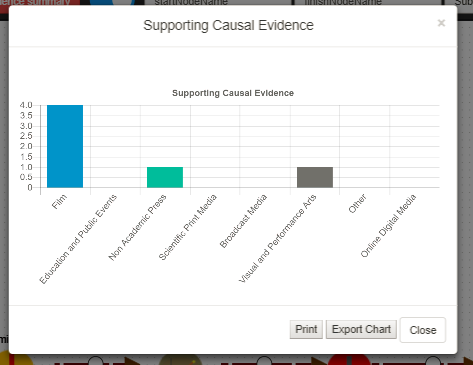
**

*Define labels (x-axis values)*

Now we add the length of each individual knowledge category to be the height of each bar chart. This visually tells the user at a glance what knowledge type and how many of each are present. If the user wants to drill down and examine the knowledge they can do that using the list-groups previously created. We also define a colour for each bar.

**

*Set the height of each bar using the amount of each knowledge category present*



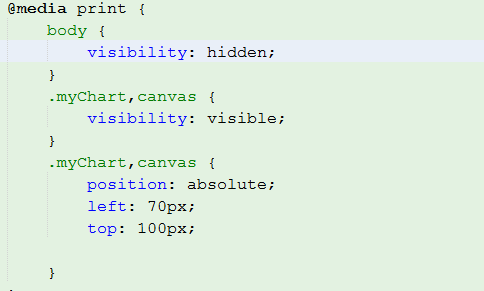
*Summary chart*

**Allow user to print the chart**

The user can print the chart by clicking the print button present on the footer of the modal window. We create a HTML button that calls window.print when clicked.

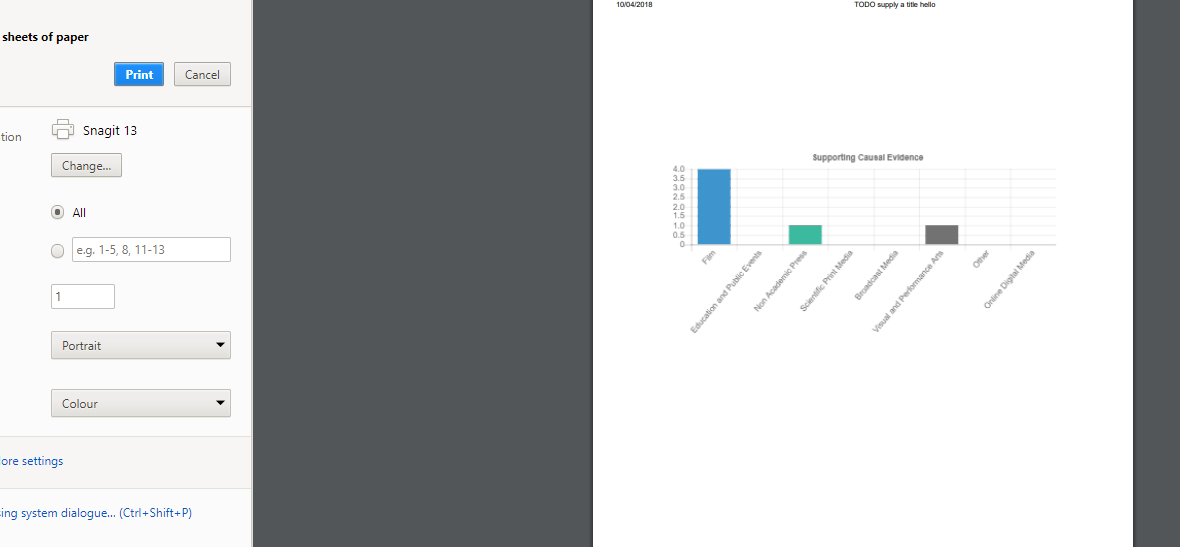


We the use the CSS @media print to only display the div that we specifically want to print which in this case is just myChart and the canvas. Everything else on the window is hidden from view. We then also insure that the divs we want to print apear in a central position of the paper.



*Hide all content except what we wish to print*

This then brings them to a new window where they can decide where they wish to print the image. By allowing the user the option to print the image the can also easily save the image to their device in any format the wish.



*Print Preview*