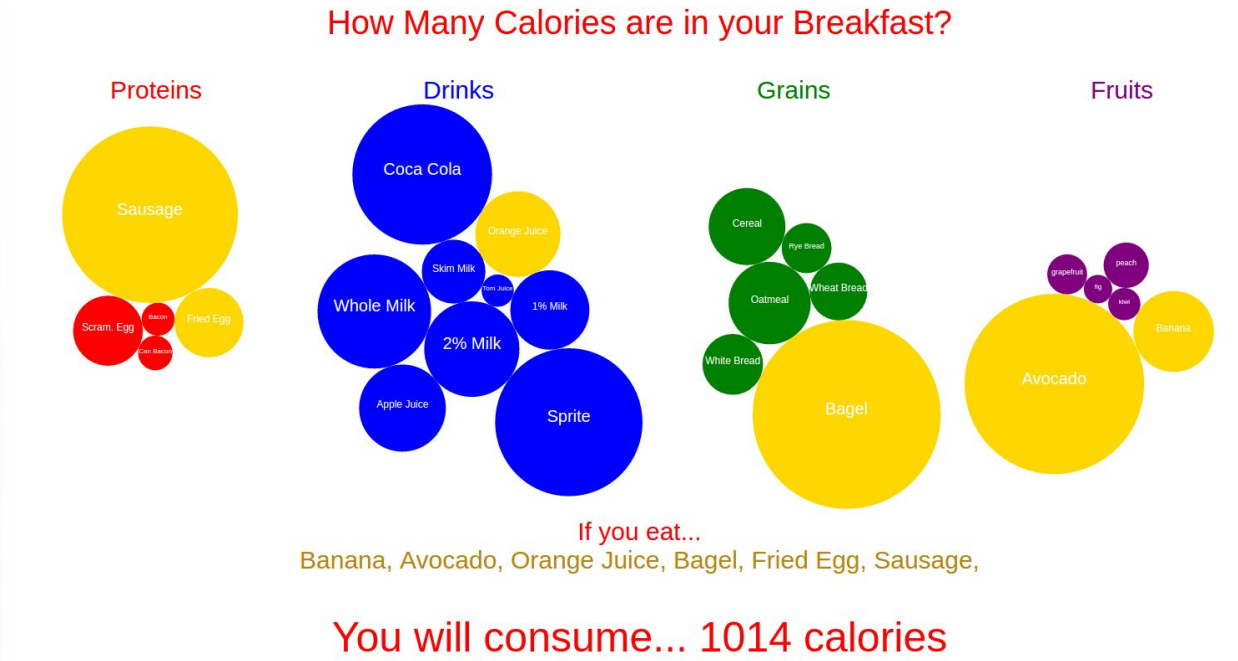
Forced Layout Node Visualization

We have provided “index.html”, “style.css”, “foods.json”, and “foods.js” files which creates a visualization where users can select among different types of breakfast foods and calculate how many calories they will consume. Although very simple, this visualization’s focus is to portray the use of d3’s interaction with a local data structure (as opposed to raw data files), multi-foci forced layouts, and node control in the d3 environment.

Download these files and preview them in brackets! Select your breakfast!

This document will walk you through the steps of creating this visualization.



Set up the Data Structure (foods.js)

The data structure is very simple, just a javascript list filled with “food” objects that d3 will render onto the visualization canvas. Set up a function called foodsModel() that takes no parameters.

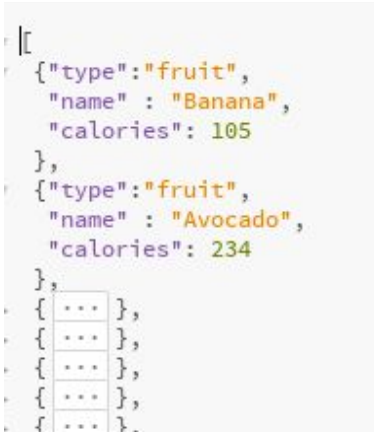
First let’s define what a food object is. For our purposes a breakfast food has a type (drink, grain, meat, or fruit), a name, and a calorie count per 1 serving - thus these are our parameters. It has a switch statement to set colors and positioning, which will become more clear later. A food returns an index, name, calorie count, type, circle radius, color, cx, cy, active. These will become clearer later when we see how d3 works with these return values.

```
function food(_type, _cal, _name) {
    var active = false;
    var color;
    var cx;
    var i;

    //switch statement to determine the color based on type
    //sets the color and the i value (used in cx calculation)
    switch (_type){
            case "meat" :
                color = "red";
                i = 1;
                break;
            case "drink" :
                color = "blue";
                i = 2;
                break;
            case "grain" :
                color = "green";
                i = 3;
                break;
            case "fruit" :
                color = "purple";
                i = 4;
                break;
            default :
                color = "black";
                break;
    }

    return{
        idx : idx++,
        name : _name,
        calories : _cal,
        type : _type,
        radius : (.46 * _cal), //scaling circle radius based on calorie count
        color : color,
        cx : (i * 380) - 200, //placing x value on page with relation to i value assigned above
        cy : 300,
        active : active,
    }
}
```

The rest of the model just returns getter and setter functions to manipulate and access the model's data from the controller.

Set up Data Set (foods.json)

Our next task is to create a json file that will fit into this food object we have created. Recall the three inputs to the food function were: type (of food), calorie count, and name. We design our json to follow these three parameters as seen below. I (tediously) inputted all this data by hand with the information I had found on the internet for calorie counts.



Set up the D3.js Controller (foods.js)

Now that we have the data structure and the data set up properly, now onto the meats and potatoes of the visualization! Create a function called the d3Controller(_model) which takes in the model as the its parameter.

Set Model from JSON

First and most importantly let's link the json we just made with the model we created in the first step! We read the json and go through each element it return and add it to the model with the addFood setter. Now the model holds all the information from the json.

```
var nodes = model.getFoods(); //establishes data as model list
```

```
//Reads foods.json in and adds each listing to the model's food list
d3.json("foods.json", function(error, data) {
      if(error) return console.warn(error);
      data.forEach(function(d) {
          model.addFood(d.type, d.calories, d.name);
      });
      attachFood();
    });
```

Create SVG Canvas on HTML

Second let's set up the svg on the page. So we use the id tag for a div in the html called "container" and arbitrary width and height values to append an svg to the html. At this point, there is a blank svg on the html with a height of 550 and a width of 1500.

```
var height = 550,
    width = 1500,
    padding = 10, // separation between nodes
    maxRadius = 20;

    //creates an svg to use as d3 canvas to the div with id "container"
    var svg = d3.select("#container").append("svg")
        .attr("width", width)
        .attr("height", height);
```

Update Function

Now we create an update method, in my case I call it attachFood(). The purpose of this method is to allow the visualization to refresh the visualization with the latest data in the model. This particular visualization doesn't utilize this purpose, but it becomes important if say *a user wants to input a new food not listed*. The following steps will need to be replicated to refresh and take place in this function.

Set Force

In this function we'll start with creating the forced layout, which makes it so nodes react to the position of other nodes. We create a variable called force that applies to all nodes in the data structure, within a particular width and height (matching svg size), and using a function called tick to allow nodes to be dragged around (we'll create this next). Now all the nodes will be under the influence of this forced layout feature.

```
//creates force layout on page that applies to all nodes and uses the tick function below
var force = d3.layout.force()
    .nodes(nodes)          //adds nodes
    .size([width, height]) //width and height match that of svg
    .gravity(0)
    .charge(0)
    .on("tick", tick)      //sets tick function
    .start();
```

Set Up Tick, Collide, and Gravity Functions

In this step, we will create the cool spring like and dragging features seen in the final visualization. In the previous step, I mentioned the tick function which would allow the food nodes to be dragged around. The circle and text values in the function will be defined in the next step when we add all the nodes to the svg. But what this is doing is changing the x and y values of the node's attributes based on the location of the node's data structure x and y values - which changes to wherever it is on the svg.

The gravity function changes the strength to which nodes are attracted to each other. The collide function forces collisions to accommodate to other nodes around it. The types of nodes will all be group on the same point, so the collide function makes them all push against each other towards that point, making it seem like a foci point. If we look back at the food structure, its x and y values were created with this in mind.

```
//Handles the movement of all of the nodes and their specific attributes
function tick(e) {
  circle
      .each(gravity(.2 * e.alpha))
      .each(collide(.5))
      .attr("cx", function(d) { return d.x; })
      .attr("cy", function(d) { return d.y; });

  text
      .each(gravity(.2 * e.alpha))
      .attr("x", function(d) { return d.x; })
      .attr("y", function(d) { return d.y; });
}

function gravity(alpha) {
  return function(d) {
    d.y += (d.cy - d.y) * alpha;
    d.x += (d.cx - d.x) * alpha;
  };
}

// Resolve collisions between nodes.
// see https://bl.ocks.org/mbostock/1804919
function collide(alpha) {
  var quadtree = d3.geom.quadtree(nodes);
  return function(d) {
    var r = d.radius + maxRadius + padding,
        nx1 = d.x - r,
        nx2 = d.x + r,
        ny1 = d.y - r,
        ny2 = d.y + r;
    quadtree.visit(function(quad, x1, y1, x2, y2) {
      if (quad.point && (quad.point !== d)) {
        var x = d.x - quad.point.x,
            y = d.y - quad.point.y,
            l = Math.sqrt(x * x + y * y),
            r = d.radius + quad.point.radius + (d.color !== quad.point.color) * padding;
        if (l < r) {
          l = (l - r) / l * alpha;
          d.x -= x *= l;
          d.y -= y *= l;
          quad.point.x += x;
          quad.point.y += y;
        }
      }
      return x1 > nx2 || x2 < nx1 || y1 > ny2 || y2 < ny1;
    });
  };
}
```

Append Nodes to SVG

We are now ready to append all of the nodes from the model onto the svg canvas! Create a variable called node which selects all of the nodes we captured from the model and append a “g” to each of them. The “g” is a d3 svg object that can be customized to contain all the text, images, or other svgs you desire.

Next you append the circle and text for the node you are creating. Provide the nodes with attributes, styles, and the ability to drag. You can view my particular design below. Next put an

onclick listener on the circle, which will allow you to make it active, change it's color when clicked, etc. Follow a similar process for the text as I did below. This is where you use all of those return values in the food object! As you see I made a few helper function to clean my code up a bit. Now you have built the visualization!

```
//creates a "g" element for each node
var node = svg.selectAll(".node")
    .data(nodes)
    .enter().append("g")


//appends a circle with information to each node
var circle = node.append("circle")
    .call(force.drag)
    .attr("class", "node")
    .attr("id", function(d) { return d.name; })
    .attr("r", function(d) { return d.radius; })
    .style("fill", function(d) { return color(d); })
    .on("click", function (d){
                        if(d.active) d.active = false;
                        else         d.active = true;

                        console.log(d.radius);
                        d3.select(this).style("fill", function(d) { return color(d);})
                        updateCalories();
    });

//appends formatted "name" text to each node
var text =
  node.append("text")
        .attr("x", function (d) { return d.cx })
        .attr("y", function (d) { return d.cy })
        .style("text-anchor", "middle")
        .text( function (d) { return d.name; })
        .attr("font-family", "sans-serif")
        .attr("font-size", function(d){return fontSize(d);})
        .attr("fill", "white");


//Changes the color of the circle if selected
function color(d){
    if(d.active) return "gold";
    else         return d.color;
}

//Sample function to change font size according to radius of circle
function fontSize(d){
    var radius = d.radius;
    if(radius < 21) return 8;
    else if(radius < 31 && radius > 21) return 9;
    else if(radius < 52 && radius > 31) return 12;
    else            return 20;
}
```

Conclusion

So we have built this visualization! This tutorial gives you all the basic setup of the visualization but perhaps leaves out some of the minor details. The best way to learn d3 is by playing around with it, so by all means try things out and see how it works! Hopefully this provides groups who will use nodes, user generated data, or multi foci a good starting place to build their final visualization!