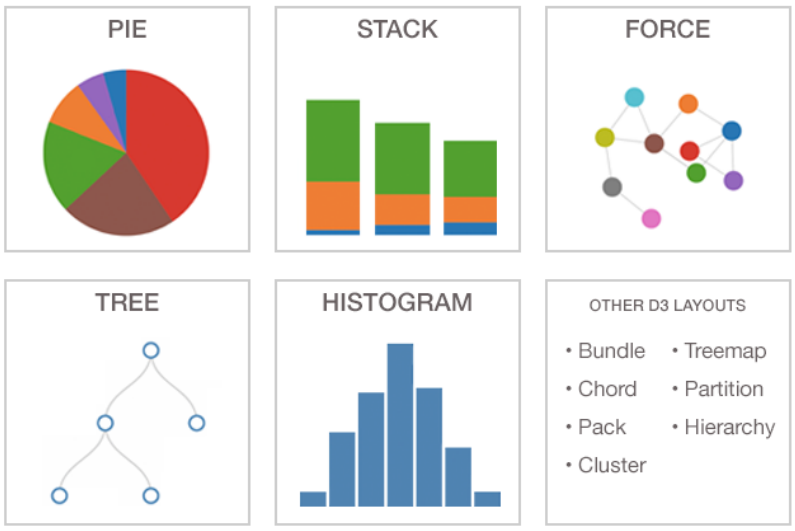
Lab 5

D3 Layouts

> The D3 layout methods have no direct visual output. Rather, D3 layouts take data that you provide and re-map or otherwise transform it, thereby generating new data that is more convenient for a specific task. (Scott Murray)

D3 offers a number of different layouts, each with distinct characteristics. Layouts are invoked using *d3.layout*:



Each layout may have distinct features not shared by other layouts, so make sure to consult the D3 documentation (https://github.com/d3/d3-3.x-api-reference/blob/master/Layouts.md) for implementation details. You will learn more about a few selected layouts in this lab.

Pie Layout

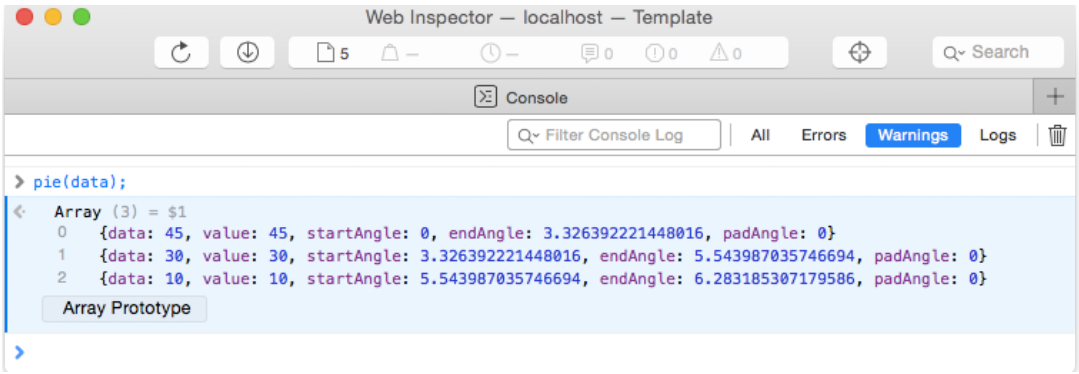
The ***d3.layout.pie()*** methods can be used to compute the start and end angles of arcs that comprise a pie or donut chart.

Example:

```
// Initialize data
var data = [45,30,10];

// Define a default pie layout
var pie = d3.layout.pie();

// Call the pie function
pie(data);
```



The D3 pie layout takes a dataset and creates an array of objects. Each of those objects contains a value from the original dataset, along with additional data, like startAngle and endAngle.

That's all there is to the D3 pie layout. It has no visual output, but transforms the input data in a way that it is much more convenient for drawing a pie chart.

Now, we'll draw a pie chart. We use the function d3.svg.arc() to generate the paths for the pie segments. Take a few minutes to look through the following code example:

```
// Define a default pie layout
var pie = d3.layout.pie();
```

```
// Pie chart settings
var outerRadius = width / 2;
var innerRadius = 0;      // Relevant for donut charts
```

```
// Path generator for the pie segments
var arc = d3.svg.arc()
    .innerRadius(innerRadius)
    .outerRadius(outerRadius);
```

```
// Append a group for each pie segment
var g = svg.selectAll(".arc")
    .data(pie(data))
    .enter()
    .append("g")
    .attr("class", "arc");
```

```
// Use the path generator to draw the arcs
g.append("path")
    .attr("d", arc)
    .style("fill", function(d, index) { return color(index); });
```

Labels:

```
// Position the labels by using arc.centroid(), which calculates the center for each pie chart segment
g.append("text")
    .attr("transform", function(d) { return "translate(" + arc.centroid(d) + ")"; })
    .attr("text-anchor", "middle")
    .attr("fill", "#fff")
    .text(function(d) { return d.value; });
```

Force Layouts

The force layout or force-directed layout is typically used to create network graphs, or also called node-link-diagrams. It consists of nodes and edges (links connecting the nodes) and helps us to visualize networks and the relationships between objects (e.g., social networks, relationships between politicians, protein–protein interaction networks, business relations, ...).

The name force-directed layout comes from the fact that these layouts use simulations of physical forces to arrange elements on the screen. The goal is to reduce the number of crossing edges, so that is easy for the user to analyze the whole network.

Activity:

```
// Initialize force layout

var force = d3.layout.force()
    .charge(-120)
    .linkDistance(30)
    .size([width, height]);



// Load data
d3.json("data/airports.json", function(data) {



});



// Inside Load data function
  console.log(data)



//force
  force
      .nodes(data.nodes)
      .links(data.links)
      .start();



  // Draw the edges (SVG lines)
  var link = svg.selectAll(".link")
      .data(data.links)
      .enter()
      .append("line")
      .attr("class", "link");
```

```
// Draw the nodes (SVG circles)
var node = svg.selectAll(".node")
    .data(data.nodes)
    .enter()
    .append("circle")
    .attr("class", "node")
    .attr("r", 5)
    .attr("fill", function(d){
      if(d.country == "United States")
        return "blue";
      else
        return "red";
    })
    .call(force.drag);
```

```
// Append a <title>-tag to each node (browser tooltips)
node.append("title")
    .text(function(d) { return d.name; });
```

```
// Update the coordinates on every tick

force.on("tick", function() {
   link.attr("x1", function(d) { return d.source.x; })
       .attr("y1", function(d) { return d.source.y; })
       .attr("x2", function(d) { return d.target.x; })
       .attr("y2", function(d) { return d.target.y; });

   node.attr("cx", function(d) { return d.x; })
       .attr("cy", function(d) { return d.y; });
});
```

GeoJSON

Example:

```
{
    "type" : "FeatureCollection",
    "features" : [
        {
          "type": "Feature",
          "geometry": {
            "type": "Point",
            "coordinates": [51.507351, -0.127758]
          },
          "properties": {
            "name": "London"
          }
        },
        {
            ...
        }
    ]
}
```

In this example we have a feature which represents a single geographical point. The coordinates of the point are specified as an array with longitude and latitude values (`[-0.127758, 51.507351]`). In GeoJSON the first element indicates the longitude, the second element the latitude value.

In many more cases, GeoJSON files contain complex polygon data that represent the boundaries of multiple regions or countries instead of a plain list of points:

```
"geometry": {
    "type": "MultiPolygon",
    "coordinates": [[[[-131.602021,55.117982],
        [-131.569159,55.28229],[-131.355558,55.183705],
        [-131.38842,55.01392],[-131.645836,55.035827], ...
    ]]]
}
```

Depending on the resolution of the dataset, each feature will include more or less longitude/latitude pairs. As you can imagine, the size of a GeoJSON file becomes tremendously high if you store the boundaries of a whole continent in high resolution.

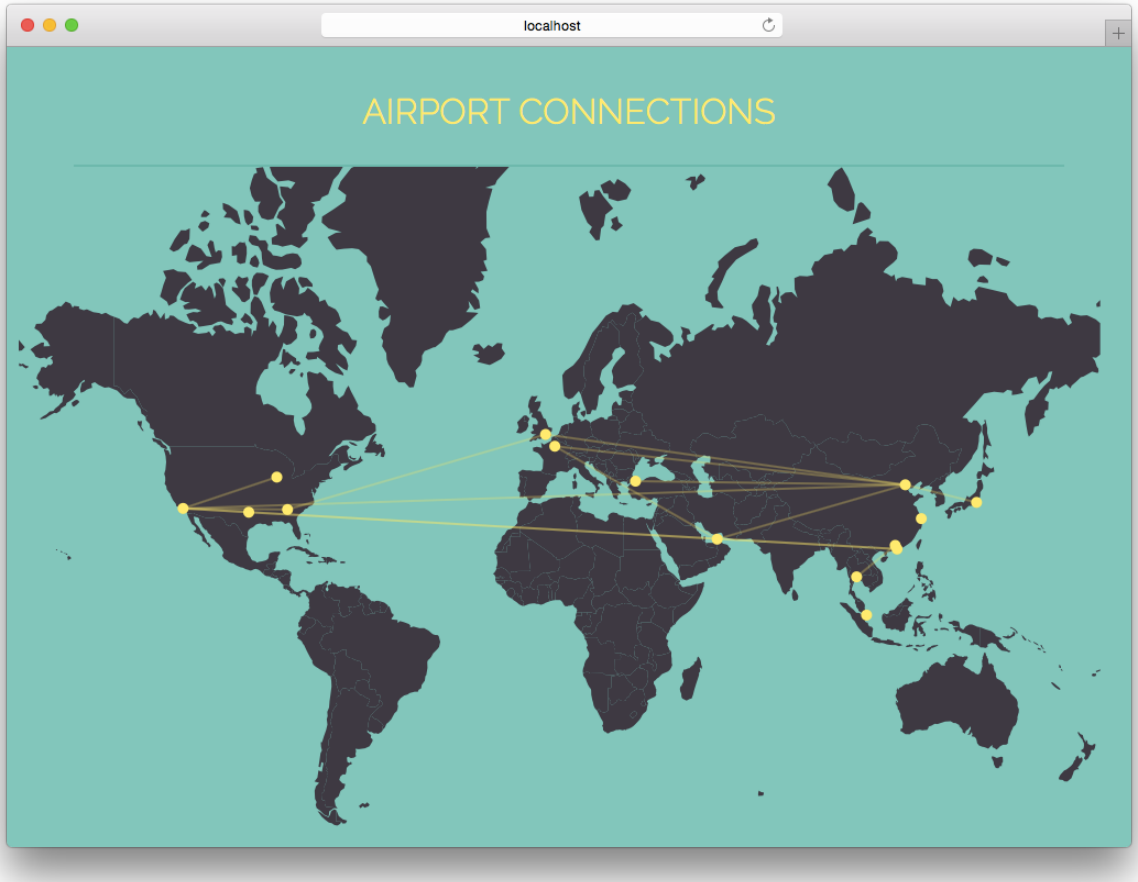
TopoJSON

TopoJSON is an extension of GeoJSON that encodes topology.

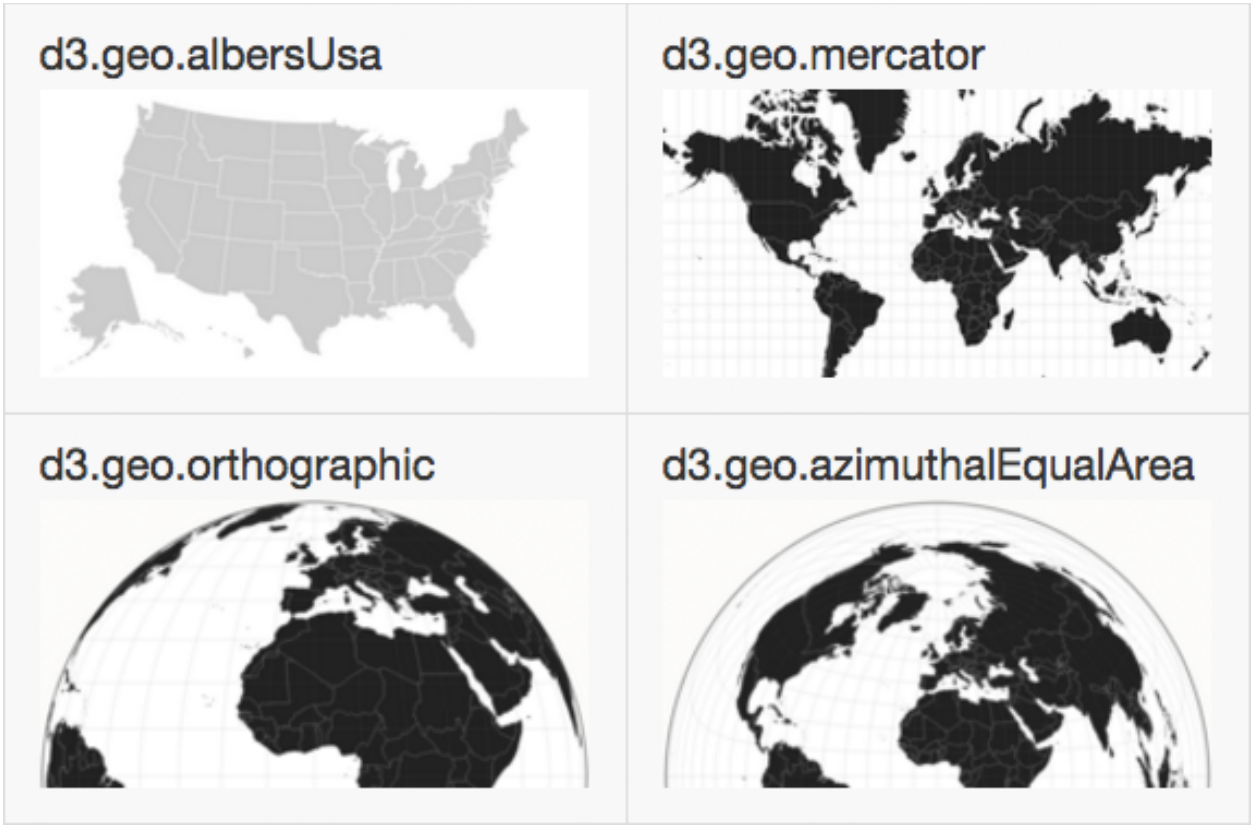
Depending on your needs, you will probably find appropriate TopoJSON files online (e.g., the US and the world atlas: https://github.com/mbostock/topojson/tree/master/examples)

→ → → Whenever you want to use a TopoJSON file in D3, you will need the TopoJSON JavaScript library to convert the data to GeoJSON for display in a web browser: http://d3js.org/topojson.v1.min.js

Activity



Workflow to implement a map with D3



Create projection ⇒ Create D3 geo path ⇒ Map TopoJSON data to the screen

```
// Projection-settings for orthographic (alternative)

  var projection = d3.geo.orthographic()
      .scale(280)
      .translate([width / 2, height / 2])
      .clipAngle(90)
      .rotate([-25.0, -38.0, -0.2])
      .precision(.1);



// D3 geo path generator (maps geodata to SVG paths)
var path = d3.geo.path()
    .projection(projection);
```

```
// inside function renderMap(error, topology, data) {…}
// Render the world atlas by using the path generator


 svg.selectAll("path")
    .data(world)
    .enter()
    .append("path")
    .attr("d", path);



 // Create a marker on the map for each airport

  svg.selectAll(".airport")
      .data(data.nodes)
    .enter().append("circle")
      .attr("class", "airport")
      .attr("r", 5)
      .attr("transform", function(d) {
        return "translate(" + projection([d.longitude,d.latitude]) + ")";
      });
```