EBook Gratis

APRENDIZAJE d3.js

Free unaffiliated eBook created from
Stack Overflow contributors.

#d3.js

Tabla de contenido

Acerca de ... 1

Capítulo 1: Empezando con d3.js ... 2

Observaciones ... 2

Versiones ... 2

Examples ... 2

Instalación ... 3

Descarga Direct Script ... 3

NPM ... 3

CDN ... 3

GITHUB ... 3

Gráfico de barras simple ... 3

index.html ... 3

chart.js ... 4

¡Hola Mundo! ... 5

¿Qué es D3? Documentos controlados por datos. ... 5

Gráfico D3 simple: ¡Hola mundo! ... 7

Capítulo 2: Conceptos básicos de SVG utilizados en la visualización de D3.js ... 10

Examples ... 10

Sistema coordinado ... 10

los Elemento ... 10

los Elemento ... 11

los Elemento ... 11

Añadir correctamente un elemento SVG ... 12

SVG: el orden de dibujo ... 14

Capítulo 3: Despachando eventos con d3.dispatch ... 17

Sintaxis ... 17

Observaciones ... 17

Examples ... 17

uso simple ... 17

Capítulo 4: En eventos ... 18

Sintaxis 18
Observaciones 18
Examples 18
Adjuntar eventos básicos en selecciones 18
Eliminar escucha de eventos 19
Capítulo 5: Enfoques para crear gráficos d3.js responsivos 20
Sintaxis 20
Examples 20
Utilizando bootstrap 20
index.html 20
chart.js 20
Capítulo 6: Gráficos SVG usando D3 js 22
Examples 22
Usando D3 js para crear elementos SVG 22
Capítulo 7: Haciendo robusto, responsivo y reutilizable (r3) para d3 23
Introducción 23
Examples 23
Gráfico de dispersión 23
¿Qué hace un gráfico? 23
Preparar 24
Configuración 24
Funciones de ayuda 24
index.html 25
Haciendo nuestro diagrama de dispersión 25
make_margins (en make_margins.js) 25
make_buttons (en make_buttons.js) 26
make_title (en make_title.js) 26
make_axes (en make_axes.js) 26
Finalmente nuestro diagrama de dispersión 26
Cuadro de caja y bigotes 27
Gráfico de barras 27

Capítulo 8: patrón de actualización 28
Sintaxis 28
Examples 28
Actualización de los datos: un ejemplo básico de selecciones de ingreso, actualización y s 28
Fusionar selecciones 30
Capítulo 9: Proyecciones D3 33
Examples 33
Proyecciones de Mercator 33
Proyecciones Albers 38
Propiedades generales 38
Eligiendo paralelos 41
Centrado y Rotación 41
Parámetros predeterminados 45
Resumen 45
Proyecciones equidistantes azimutales 45
Propiedades generales: 45
Centrado y Rotación: 47
Capítulo 10: Trozos escogidos 50
Sintaxis 50
Observaciones 50
Examples 50
Selección básica y modificaciones 50
Diferentes selectores 51
Selección simple de datos limitados 51
El papel de los marcadores de posición en las selecciones "entrar" 51
Usando "esto" con una función de flecha 54
La función de flecha 55
Los argumentos segundo y tercero combinados 55
Capítulo 11: Usando D3 con JSON y CSV 57
Sintaxis 57
Examples 57

Cargando datos de archivos CSV 57
Uno o dos parámetros en la devolución de llamada: manejo de errores en d3.request () 58
Capítulo 12: Usando D3 con otros frameworks 61
Examples 61
Componente D3.js con ReactJS 61
d3_react.html 61
d3_react.js 61
D3js con Angular 63
Gráfico de D3.js con Angular v1 65
Creditos 67

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: d3-js

It is an unofficial and free d3.js ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official d3.js.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con d3.js

Observaciones

D3.js es una biblioteca de JavaScript para manipular documentos basados en datos. D3 le ayuda a dar vida a los datos mediante HTML, SVG y CSS. El énfasis de D3 en los estándares web le brinda todas las capacidades de los navegadores modernos sin atarse a un marco propietario, combinando poderosos componentes de visualización y un enfoque basado en datos para la manipulación de DOM.

El sitio web oficial: https://d3js.org/ Muchos ejemplos aquí: http://bl.ocks.org/mbostock

Versiones

Versión	Fecha de lanzamiento
v1.0.0	2011-02-11
2.0	2011-08-23
3.0 "baja"	2012-12-21
v4.0.0	2016-06-28
v4.1.1	2016-07-11
v4.2.1	2016-08-03
v4.2.2	2016-08-16
v4.2.3	2016-09-13
v4.2.4	2016-09-19
v4.2.5	2016-09-20
v4.2.6	2016-09-22
v4.2.7	2016-10-11
v4.2.8	2016-10-20
v4.3.0	2016-10-27

Examples

Instalación

Hay varias formas de descargar y usar D3.

Descarga Direct Script

1. Descargar y extraer d3.zip
2. Copie la carpeta resultante donde guardará las dependencias de su proyecto
3. Consulte d3.js (para desarrollo) o d3.min.js (para producción) en su HTML: `<script type="text/javascript" src="scripts/d3/d3.js"></script>`

NPM

1. Inicialice NPM en su proyecto si aún no lo ha hecho: `npm init`
2. NPM instala D3: `npm install --save d3`
3. Consulte d3.js (para desarrollo) o d3.min.js (para producción) en su HTML: `<script type="text/javascript" src="node_modules/d3/build/d3.js"></script>`

CDN

1. Consulte d3.js (para desarrollo) o d3.min.js (para producción) en su HTML: `<script type="text/javascript" src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.1.1/d3.js"></script>`

GITHUB

1. Obtenga cualquier versión de d3.js (para desarrollo) o d3.min.js (para producción) de Github: `<script type="text/javascript" src="https://raw.githubusercontent.com/d3/d3/v3.5.16/d3.js"></script>`

Para enlazar directamente a la última versión, copie este fragmento de código:

```
<script src="https://d3js.org/d3.v4.min.js"></script>
```

Gráfico de barras simple

index.html

```
<!doctype html>
<html>
  <head>
    <title>D3 Sample</title>
  </head>
  <body>
    <!-- This will serve as a container for our chart. This does not have to be a div, and can
in fact, just be the body if you want. -->
    <div id="my-chart"></div>

    <!-- Include d3.js from a CDN. -->
    <script type="text/javascript"
```

```
src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.1.1/d3.js"></script>

    <!-- Include our script that will make our bar chart. -->
    <script type="text/javascript" src="chart.js"></script>
  </body>
</html>
```

chart.js

```
// Sample dataset. In a real application, you will probably get this data from another source
such as AJAX.
var dataset = [5, 10, 15, 20, 25]

// Sizing variables for our chart. These are saved as variables as they will be used in
calculations.
var chartWidth = 300
var chartHeight = 100
var padding = 5

// We want our our bars to take up the full height of the chart, so, we will apply a scaling
factor to the height of every bar.
var heightScalingFactor = chartHeight / getMax(dataset)

// Here we are creating the SVG that will be our chart.
var svg = d3
  .select('#my-chart')           // I'm starting off by selecting the container.
    .append('svg')               // Appending an SVG element to that container.
    .attr('width', chartWidth)   // Setting the width of the SVG.
    .attr('height', chartHeight) // And setting the height of the SVG.

// The next step is to create the rectangles that will make up the bars in our bar chart.
svg
  .selectAll('rect')                                          // I'm selecting all of the
rectangles in the SVG (note that at this point, there actually aren't any, but we'll be
creating them in a couple of steps).
  .data(dataset)                                              // Then I'm mapping the dataset
to those rectangles.
  .enter()                                                    // This step is important in
that it allows us to dynamically create the rectangle elements that we selected previously.
    .append('rect')                                           // For each element in the
dataset, append a new rectangle.
      .attr('x', function (value, index) {                    // Set the X position of the
rectangle by taking the index of the current item we are creating, multiplying it by the
calculated width of each bar, and adding a padding value so we can see some space between
bars.
          return (index * (chartWidth / dataset.length)) + padding
        })
      .attr('y', function (value, index) {                    // Set the rectangle by
subtracting the scaled height from the height of the chart (this has to be done becuase SVG
coordinates start with 0,0 at their top left corner).
        return chartHeight - (value * heightScalingFactor)
      })
      .attr('width', (chartWidth / dataset.length) - padding) // The width is dynamically
calculated to have an even distribution of bars that take up the entire width of the chart.
      .attr('height', function (value, index) {               // The height is simply the
value of the item in the dataset multiplied by the height scaling factor.
        return value * heightScalingFactor
      })
      .attr('fill', 'pink')                                   // Sets the color of the bars.
```

```
/**
 *  Gets the maximum value in a collection of numbers.
 */
function getMax(collection) {
  var max = 0

  collection.forEach(function (element) {
    max = element > max ? element : max
  })

  return max
}
```

Código de ejemplo disponible en https://github.com/dcsinnovationlabs/D3-Bar-Chart-Example

Demo disponible en https://dcsinnovationlabs.github.io/D3-Bar-Chart-Example/

¡Hola Mundo!

Cree un archivo `.html` que contenga este fragmento de código:

```
<!DOCTYPE html>
<meta charset="utf-8">
<body>
<script src="//d3js.org/d3.v4.min.js"></script>
<script>

d3.select("body").append("span")
    .text("Hello, world!");

</script>
```

Vea este fragmento en acción en este JSFiddle .

¿Qué es D3? Documentos controlados por datos.

Estamos tan acostumbrados al nombre **D3.js** que es posible olvidar que D3 es en realidad **DDD** (**D** ata- **D** **D** riven **D** Ouments). Y eso es lo que D3 hace bien, un enfoque basado en datos para la manipulación de DOM (Document Object Model): D3 enlaza datos a elementos DOM y manipula esos elementos en función de los datos delimitados.

Veamos una característica muy básica de D3 en este ejemplo. Aquí, no añadiremos ningún elemento SVG. En su lugar, usaremos un SVG ya presente en la página, algo como esto:

```
<svg width="400" height="400">
    <circle cx="50" cy="50" r="10"></circle>
    <circle cx="150" cy="50" r="10"></circle>
    <circle cx="210" cy="320" r="10"></circle>
    <circle cx="210" cy="30" r="10"></circle>
    <circle cx="180" cy="200" r="10"></circle>
</svg>
```

Este es un SVG bastante básico, con 5 círculos. En este momento, esos círculos no están

"vinculados" a ningún dato. Veamos esta última alegación:

En nuestro código, escribimos:

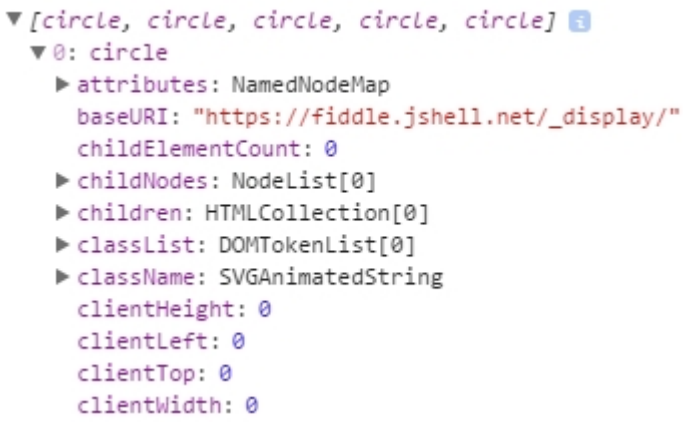
```
var svg = d3.select("svg");
var circles = svg.selectAll("circle");
console.log(circles.nodes());
```

Aquí, `d3.select("svg")` devuelve un objeto d3 que contiene la etiqueta `<svg width="400" height="400"></svg>` y todas las etiquetas secundarias, las `<circle>` s. Tenga en cuenta que si existen varias etiquetas `svg` en la página, solo se seleccionará la primera. Si no desea esto, también puede seleccionar por ID de etiqueta, como `d3.select("#my-svg")` . El objeto d3 tiene propiedades y métodos integrados que usaremos mucho más adelante.

`svg.selectAll("circle")` crea un objeto a partir de todos los elementos `<circle></circle>` dentro de la etiqueta `<svg>` . Puede buscar a través de múltiples capas, por lo que no importa si las etiquetas son hijos directos.

`circles.nodes()` devuelve las etiquetas de círculo con todas sus propiedades.

Si miramos la consola y elegimos el primer círculo, veremos algo como esto:



Primero, tenemos `attributes` , luego `childNodes` , luego `children` , etc., pero no datos.

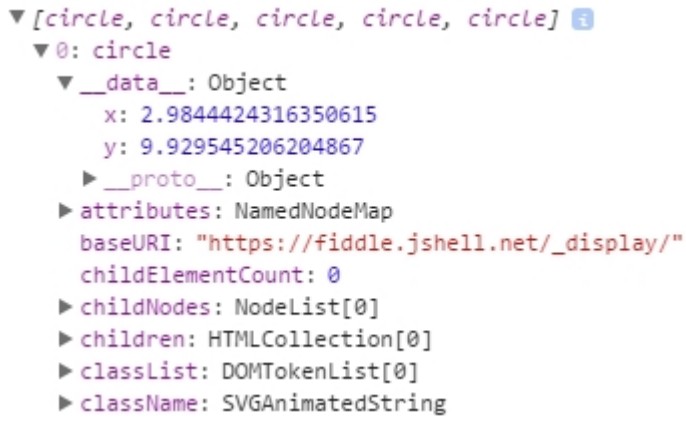
Unamos algunos datos

Pero, ¿qué pasa si unimos los *datos* a estos elementos DOM?

En nuestro código, hay una función que crea un objeto con dos propiedades, `x` y `y` , con los valores numéricos (este objeto está dentro de una matriz, comprobar el violín a continuación). Si enlazamos estos datos a los círculos ...

```
circles.data(data);
```

Esto es lo que vamos a ver si inspeccionamos la consola:



¡Tenemos algo nuevo justo antes de los `attributes` ! Algo llamado `__data__` ... y mira: ¡los valores de `x` e `y` están ahí!

Podemos, por ejemplo, cambiar la posición de los círculos en función de estos datos. Echa un vistazo a este violín .

Esto es lo que D3 hace mejor: vincular datos a elementos DOM y manipular esos elementos DOM basados en los datos acotados.

Gráfico D3 simple: ¡Hola mundo!

Pegue este código en un archivo HTML vacío y ejecútelo en su navegador.

```
<!DOCTYPE html>

<body>

<script src="https://d3js.org/d3.v4.js"></script>    <!-- This downloads d3 library -->

<script>
//This code will visualize a data set as a simple scatter chart using d3. I omit axes for
simplicity.
var data = [        //This is the data we want to visualize.
                    //In reality it usually comes from a file or database.
  {x: 10,    y: 10},
  {x: 10,    y: 20},
  {x: 10,    y: 30},
  {x: 10,    y: 40},
  {x: 10,    y: 50},
  {x: 10,    y: 80},
  {x: 10,    y: 90},
  {x: 10,    y: 100},
  {x: 10,    y: 110},
  {x: 20,    y: 30},
  {x: 20,    y: 120},
  {x: 30,    y: 10},
  {x: 30,    y: 20},
  {x: 30,    y: 30},
  {x: 30,    y: 40},
  {x: 30,    y: 50},
  {x: 30,    y: 80},
  {x: 30,    y: 90},
  {x: 30,    y: 100},
  {x: 30,    y: 110},
```

```
{x: 40,    y: 120},
{x: 50,    y: 10},
{x: 50,    y: 20},
{x: 50,    y: 30},
{x: 50,    y: 40},
{x: 50,    y: 50},
{x: 50,    y: 80},
{x: 50,    y: 90},
{x: 50,    y: 100},
{x: 50,    y: 110},
{x: 60,    y: 10},
{x: 60,    y: 30},
{x: 60,    y: 50},
{x: 70,    y: 10},
{x: 70,    y: 30},
{x: 70,    y: 50},
{x: 70,    y: 90},
{x: 70,    y: 100},
{x: 70,    y: 110},
{x: 80,    y: 80},
{x: 80,    y: 120},
{x: 90,    y: 10},
{x: 90,    y: 20},
{x: 90,    y: 30},
{x: 90,    y: 40},
{x: 90,    y: 50},
{x: 90,    y: 80},
{x: 90,    y: 120},
{x: 100,    y: 50},
{x: 100,    y: 90},
{x: 100,    y: 100},
{x: 100,    y: 110},
{x: 110,    y: 50},
{x: 120,    y: 80},
{x: 120,    y: 90},
{x: 120,    y: 100},
{x: 120,    y: 110},
{x: 120,    y: 120},
{x: 130,    y: 10},
{x: 130,    y: 20},
{x: 130,    y: 30},
{x: 130,    y: 40},
{x: 130,    y: 50},
{x: 130,    y: 80},
{x: 130,    y: 100},
{x: 140,    y: 50},
{x: 140,    y: 80},
{x: 140,    y: 100},
{x: 140,    y: 110},
{x: 150,    y: 50},
{x: 150,    y: 90},
{x: 150,    y: 120},
{x: 170,    y: 20},
{x: 170,    y: 30},
{x: 170,    y: 40},
{x: 170,    y: 80},
{x: 170,    y: 90},
{x: 170,    y: 100},
{x: 170,    y: 110},
{x: 170,    y: 120},
{x: 180,    y: 10},
```

```
  {x: 180,    y: 50},
  {x: 180,    y: 120},
  {x: 190,    y: 10},
  {x: 190,    y: 50},
  {x: 190,    y: 120},
  {x: 200,    y: 20},
  {x: 200,    y: 30},
  {x: 200,    y: 40},
  {x: 210,    y: 80},
  {x: 210,    y: 90},
  {x: 210,    y: 100},
  {x: 210,    y: 110},
  {x: 210,    y: 120},
  {x: 220,    y: 80},
  {x: 220,    y: 120},
  {x: 230,    y: 80},
  {x: 230,    y: 120},
  {x: 240,    y: 90},
  {x: 240,    y: 100},
  {x: 240,    y: 110},
  {x: 270,    y: 70},
  {x: 270,    y: 80},
  {x: 270,    y: 90},
  {x: 270,    y: 100},
  {x: 270,    y: 120}
];

//The following code chains a bunch of methods. Method chaining is what makes d3 very simple
and concise.
d3.select("body").append("svg").selectAll()  //'d3' calls the d3 library
                                             //'.select' selects the object (in this case the
body of HTML)
                                             //'.append' adds SVG element to the body
                                             //'.selectAll()' selects all SVG elements
    .data(data)                              //'.data' gets the data from the variable 'data'
  .enter().append("circle")                  //'.enter' enters the data into the SVG
                                             //the data enter as circles with
'.append("circle")'
    .attr("r", 3)                            //'.attr' adds/alters atributes of SVG,
                                             //such as radius ("r"), making it 3 pixels
    .attr("cx", function(d) { return d.x; }) //coordinates "cx" (circles' x coordinates)
    .attr("cy", function(d) { return d.y; }) //coordinates "cy" (circles' y coordinates)
    .style("fill", "darkblue");              //'.style' changes CSS of the SVG
                                             //in this case, fills circles with "darkblue"
color

</script>
```

Aquí hay un JSFiddle del gráfico.

También puede descargar el archivo HTML ya creado desde GitHub .

El siguiente paso para aprender d3 puede ser seguir el tutorial de Mike Bostock (el creador de d3) para crear un gráfico de barras desde cero .

Lea Empezando con d3.js en línea: https://riptutorial.com/es/d3-js/topic/876/empezando-con-d3-js

Capítulo 2: Conceptos básicos de SVG utilizados en la visualización de D3.js

Examples

Sistema coordinado

En un sistema de coordenadas matemático normal, el punto x = 0, y = 0 está en la esquina inferior izquierda de la gráfica. Pero en el sistema de coordenadas SVG, este punto (0,0) está en la esquina superior izquierda del 'lienzo', es similar a CSS cuando se especifica la posición en absoluto / corregir y se usa la parte superior e izquierda para controlar el Punto exacto del elemento.

Es esencial tener en cuenta que a medida que y aumenta en SVG, las formas se mueven hacia abajo.

Digamos que vamos a crear un diagrama de dispersión con cada punto correspondiente al valor de ax y el valor de y. Para escalar el valor, necesitamos establecer el dominio y el rango de esta manera:

```
d3.svg.scale()
  .range([0, height])
  .domain([0,max])
```

Sin embargo, si solo mantiene la configuración de esta manera, los puntos se basarán en el borde horizontal superior en lugar de la línea horizontal inferior como esperábamos.

Lo bueno de d3 es que puede cambiarlo fácilmente con un simple ajuste en la configuración del dominio:

```
d3.scale.linear()
  .range([height, 0])
  .domain([0, max])
```

Con el código anterior, el punto cero del dominio corresponde a la altura del SVG, que es la línea inferior del gráfico a los ojos del espectador, mientras tanto, el valor máximo de los datos de origen será el correspondiente al punto cero del SVG sistema de coordenadas, que el valor máximo para los espectadores.

los Elemento

`<rect>` representa un rectángulo, aparte de las propiedades estéticas como trazo y relleno, el rectángulo se definirá por ubicación y tamaño.

En cuanto a la ubicación, está determinada por los atributos x e y. La ubicación es relativa a la

matriz del rectángulo. Y si no especifica el atributo x o y, el valor predeterminado será 0 en relación con el elemento padre.

Después de especificar la ubicación, o más bien el 'punto de inicio' del rect, lo siguiente es especificar el tamaño, que es esencial si realmente desea dibujar algo en el lienzo, es decir, si no lo hace especifique los atributos de tamaño o el valor se establece en 0, no verá nada en el lienzo.

Caso: gráfico de barras

Continúe con el primer escenario, los ejes y, pero esta vez, vamos a tratar de dibujar un gráfico de barras.

Asumiendo que la configuración de la escala y es la misma, el eje y también está correctamente establecido, la única diferencia entre el diagrama de dispersión y este gráfico de barras es que debemos especificar el ancho y la altura, particularmente la altura. Para ser más específicos, ya tenemos el 'punto de partida', el resto es usar cosas como para la altura:

```
.attr("height", function(d){
  return (height - yScale(d.value))
})
```

los Elemento

`<svg>` elemento `<svg>` es el elemento raíz, o el lienzo, ya que estamos dibujando gráficos en él.

Los elementos SVG se pueden anidar uno dentro del otro, y de esta manera, las formas SVG se pueden agrupar, mientras tanto, todas las formas anidadas dentro de un elemento se posicionarán con relación a su elemento de encierro.

Una cosa que podría ser necesario mencionar es que, no podemos anidar dentro de otra, no funcionará.

Caso: Cartas Múltiples

Por ejemplo, este cuadro de donas múltiples está formado por varios elementos, que contienen un cuadro de anillos respectivamente. Esto se puede lograr usando el elemento, pero en este caso donde solo queremos poner el gráfico de anillos uno a uno uno al lado del otro, es más conveniente.

Una cosa a tener en cuenta es que no podemos usar el atributo de transformación en, sin embargo, podemos usar x, y para la posición.

los Elemento

SVG no admite actualmente saltos de línea automáticos o ajuste de palabras, que es cuando se trata de rescatar. El elemento coloca nuevas líneas de texto en relación con la línea de texto anterior. Y al usar dx o dy dentro de cada uno de estos tramos, podemos colocar la palabra en relación con la palabra anterior.

Caso: Anotación sobre ejes.

Por ejemplo, cuando queremos agregar una anotación en y Ax:

```
svg.append("g")
  .attr("class", "y axis")
  .call(yAxis)
.append("text")
  .attr("transform", "rotate(-90)")
  .attr("y", 6)
  .attr("dy", ".71em")
  .style("text-anchor", "end")
  .text("Temperature (°F)");
```

Añadir correctamente un elemento SVG

Este es un error relativamente común: usted creó un elemento `rect` , por ejemplo, en un gráfico de barras, y desea agregar una etiqueta de texto (digamos, el valor de esa barra). Entonces, usando la misma variable que usaste para agregar el `rect` y definir sus posiciones `x` e `y` , agregas el elemento de `text` . Muy lógica, puedes pensar. Pero esto no funcionará.

¿Cómo se produce este error?

Veamos un ejemplo concreto, un código muy básico para crear un gráfico de barras (aquí se toca el violín):

```
var data = [210, 36, 322, 59, 123, 350, 290];

var width = 400, height = 300;

var svg = d3.select("body")
    .append("svg")
    .attr("width", width)
    .attr("height", height);

var bars = svg.selectAll(".myBars")
    .data(data)
    .enter()
    .append("rect");

bars.attr("x", 10)
    .attr("y", function(d,i){ return 10 + i*40})
    .attr("width", function(d){ return d})
    .attr("height", 30);
```

Lo que nos da este resultado:

Pero desea agregar algunos elementos de texto, tal vez un valor simple a cada barra. Entonces, haces esto:

```
bars.append("text")
   .attr("x", 10)
   .attr("y", function(d,i){ return 10 + i*40})
   .text(function(d){ return d});
```

Y, voilà: ¡no pasa nada! Si lo dudas, aquí tienes el violín .

"¡Pero estoy viendo la etiqueta!"

Si inspeccionas el SVG creado por este último código, verás esto:

```
▼<rect x="10" y="250" width="290" height="30">
   <text x="10" y="250">290</text>
 </rect>
```

Y en este punto, mucha gente dice: "¡Pero estoy viendo la etiqueta de texto, está adjunta!". Sí, lo es, pero esto no significa que funcionará. ¡Puedes añadir *cualquier cosa* ! Vea esto, por ejemplo:

```
svg.append("crazyTag");
```

Te dará este resultado:

```
<svg>
    <crazyTag></crazyTag>
</svg>
```

Pero no esperas ningún resultado solo porque la etiqueta está ahí, ¿verdad?

Añadir elementos SVG de la manera correcta

Aprende qué elementos SVG pueden contener los niños, leyendo las especificaciones . En nuestro último ejemplo, el código no funciona porque los elementos `rect` no pueden contener

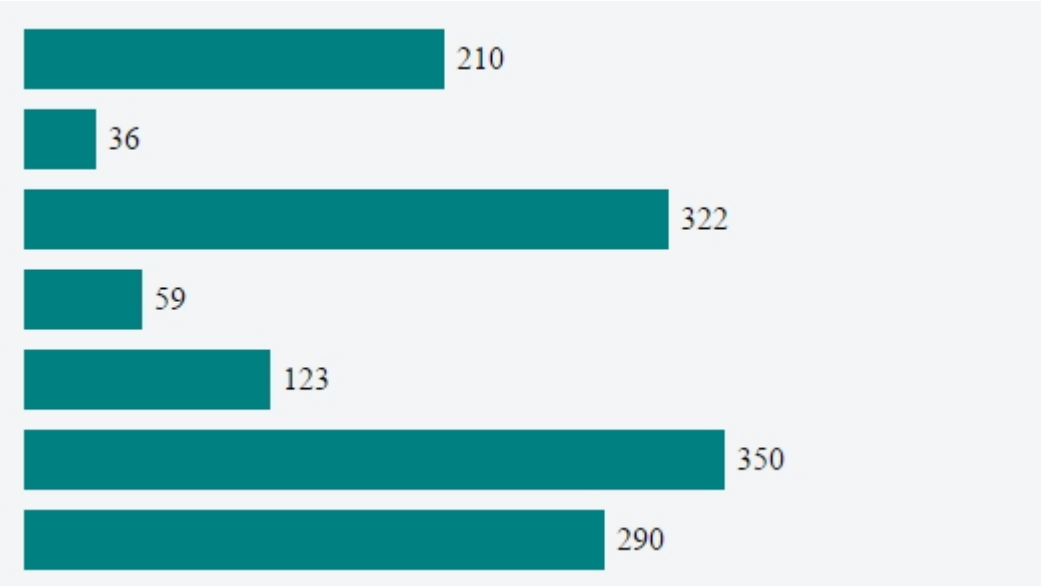
elementos de `text` . Entonces, ¿cómo mostrar nuestros textos?

Cree otra variable y agregue el `text` al SVG:

```
var texts = svg.selectAll(".myTexts")
    .data(data)
    .enter()
    .append("text");

texts.attr("x", function(d){ return d + 16})
    .attr("y", function(d,i){ return 30 + i*40})
    .text(function(d){ return d});
```

Y este es el resultado:



Y aquí está el violín .

SVG: el orden de dibujo

Esto es algo que puede ser frustrante: realiza una visualización con D3.js pero el rectángulo que desea en la parte superior está oculto detrás de otro rectángulo, o la línea que planeaba estar detrás de algún círculo en realidad está sobre él. Intenta resolver esto utilizando el *índice z* en su CSS, pero no funciona (en SVG 1.1).

La explicación es simple: en un SVG, el orden de los elementos define el orden de la "pintura", y el orden de la pintura define quién va arriba.

> Los elementos en un fragmento de documento SVG tienen un orden de dibujo implícito, y los primeros elementos en el fragmento de documento SVG se "pintan" primero. Los elementos posteriores se pintan sobre los elementos previamente pintados.

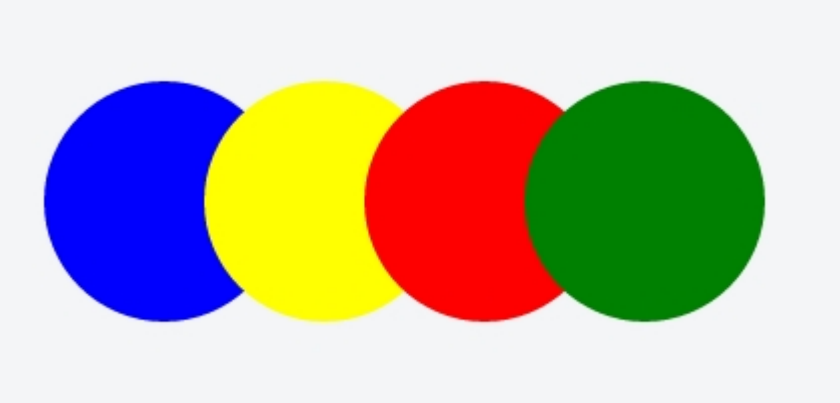
Entonces, supongamos que tenemos este SVG:

```
<svg width="400" height=200>
```

```
    <circle cy="100" cx="80" r="60" fill="blue"></circle>
    <circle cy="100" cx="160" r="60" fill="yellow"></circle>
    <circle cy="100" cx="240" r="60" fill="red"></circle>
    <circle cy="100" cx="320" r="60" fill="green" z-index="-1"></circle>
</svg>
```

Tiene cuatro círculos. El círculo azul es el primero "pintado", por lo que estará debajo de todos los demás. Luego tenemos el amarillo, luego el rojo, y finalmente el verde. El verde es el último, y estará en la parte superior.

Así es como se ve:



Cambiando el orden de los elementos SVG con D3

Entonces, ¿es posible cambiar el orden de los elementos? ¿Puedo hacer el círculo rojo delante del círculo verde?

Sí. El primer enfoque que debe tener en cuenta es el orden de las líneas en su código: dibuje primero los elementos del fondo, y luego en el código los elementos del primer plano.

Pero podemos cambiar dinámicamente el orden de los elementos, incluso después de que fueron pintados. Hay varias funciones de JavaScript simples que puede escribir para hacer esto, pero D3 ya tiene 2 características interesantes, `selection.raise()` y `selection.lower()` .

Según la API:

> selection.raise (): Reinserta cada elemento seleccionado, en orden, como el último hijo de su padre. selection.lower (): Reinserta cada elemento seleccionado, en orden, como el primer hijo de su padre.

Entonces, para mostrar cómo manipular el orden de los elementos en nuestro SVG anterior, aquí hay un código muy pequeño:

```
d3.selectAll("circle").on("mouseover", function(){
    d3.select(this).raise();
});
```

¿Qué hace? Selecciona todos los círculos y, cuando el usuario se desplaza sobre un círculo, selecciona ese círculo en particular y lo lleva al frente. ¡Muy simple!

Y aquí está el JSFiddle con el código en vivo.

Lea Conceptos básicos de SVG utilizados en la visualización de D3.js en línea: https://riptutorial.com/es/d3-js/topic/2537/conceptos-basicos-de-svg-utilizados-en-la-visualizacion-de-d3-js

Capítulo 3: Despachando eventos con d3.dispatch

Sintaxis

- d3 **dispatch** - crea un despachador de eventos personalizado.
- envío. **en** - registre o anule el registro de un detector de eventos.
- envío. **Copiar** : crea una copia de un despachador.
- envío. **llamada** - despacha un evento a oyentes registrados.
- envío. **aplicar** - enviar un evento a los oyentes registrados.

Observaciones

El envío es un mecanismo conveniente para separar las inquietudes con un código poco acoplado: registre devoluciones de llamada llamadas y luego llámelas con argumentos arbitrarios. Una variedad de componentes D3, como la solicitud d3, utilizan este mecanismo para emitir eventos a los oyentes. Piense en esto como el EventEmitter de Node, excepto que cada oyente tiene un nombre bien definido, por lo que es fácil eliminarlos o reemplazarlos.

Lecturas relacionadas

- Despachando eventos por Mike Bostock
- d3.dispatch Documentation
- Evento de despacho en NPM

Examples

uso simple

```
var dispatch = d3.dispatch("statechange");

dispatch.on('statechange', function(e){ console.log(e) })

setTimeout(function(){dispatch.statechange('Hello, world!')}, 3000)
```

Lea Despachando eventos con d3.dispatch en línea: https://riptutorial.com/es/d3-js/topic/3399/despachando-eventos-con-d3-dispatch

Capítulo 4: En eventos

Sintaxis

- .on ('mouseover', función)
- .on ('mouseout', función)
- .on ('clic', función)
- .on ('mouseenter', función)
- .on ('mouseleave', función)

Observaciones

Para un ejemplo más detallado donde se definen los eventos personalizados, consulte aquí .

Examples

Adjuntar eventos básicos en selecciones.

Muchas veces querrás tener eventos para tus objetos.

```
function spanOver(d,i){
  var span = d3.select(this);
  span.classed("spanOver",true);
}

function spanOut(d,i){
  var span = d3.select(this);
  span.classed("spanOver", false);
}

var div = d3.select('#divID');

div.selectAll('span')
  .on('mouseover' spanOver)
  .on('mouseout' spanOut)
```

Este ejemplo agregará la clase `spanOver` cuando se `spanOver` sobre un espacio dentro del div con el ID de `divID` y lo eliminará cuando el mouse salga del espacio.

Por defecto, d3 pasará el dato del intervalo actual y el índice. Es muy útil que `this` 's contexto es el objeto actual, así por lo que podemos hacer operaciones en él, como añadir o eliminar clases.

También puede usar una función anónima en el evento.

```
div.selectAll('span')
  .on('click', function(d,i){ console.log(d); });
```

Los elementos de datos también se pueden agregar al objeto seleccionado actual.

```
div.selectAll('path')
  .on('click', clickPath);

function clickPath(d,i) {
  if(!d.active) {
    d.active = true;
    d3.select(this).classed("active", true);
  }
  else {
    d.active = false;
    d3.select(this).classed("active", false);
  }
}
```

En este ejemplo, activo no está definido en la selección antes de que se active el evento de clic. Si tuviera que volver sobre la selección de ruta, todos los objetos en los que se haga clic contendrían la clave `active` .

Eliminar escucha de eventos

d3.js no tiene un método `.off()` para asignar detectores de eventos existentes. Para eliminar un controlador de eventos, debe configurarlo como `null` :

```
d3.select('span').on('click', null)
```

Lea En eventos en línea: https://riptutorial.com/es/d3-js/topic/2722/en-eventos

Capítulo 5: Enfoques para crear gráficos d3.js responsivos

Sintaxis

- var width = document.getElementById ('chartArea'). clientWidth;
- var altura = ancho / 3.236;
- window.onresize = resizeFunctionCall;

Examples

Utilizando bootstrap

Un enfoque que se emplea a menudo es usar el marco cuadriculado de bootstrap para definir el área en la que existirá el gráfico. Usando esto junto con la variable `clientWidth` y el evento `window.onresize` , es muy fácil crear SVG responsivos d3 .

Primero creamos una fila y una columna en la que se construirá nuestro gráfico.

index.html

```
<!DOCTYPE html>
<html>
<head>
<link rel="stylesheet" type="text/css"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css">
</head>
<body>
  <div class="container">
    <div class="row">
      <div class="col-xs-12 col-lg-6" id="chartArea">
      </div>
    </div>
  </div>
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/4.1.1/d3.js"></script>
<script src="chart.js"></script>
</body>
</html>
```

Esto creará una columna que será a pantalla completa en el dispositivo móvil y la mitad en una pantalla grande.

chart.js

```
var width = document.getElementById('chartArea').clientWidth;
//this allows us to collect the width of the div where the SVG will go.
var height = width / 3.236;
```

```
//I like to use the golden rectangle ratio if they work for my charts.

var svg = d3.select('#chartArea').append('svg');
//We add our svg to the div area


//We will build a basic function to handle window resizing.
function resize() {
    width = document.getElementById('chartArea').clientWidth;
    height = width / 3.236;
    d3.select('#chartArea svg')
      .attr('width', width)
      .attr('height', height);
}

window.onresize = resize;
//Call our resize function if the window size is changed.
```

Este es un ejemplo extremadamente simplificado, pero cubre los conceptos básicos de cómo configurar gráficos para ser receptivos. La función de cambio de tamaño deberá realizar una llamada a la función de actualización principal que redibujará todas las rutas, ejes y formas como si los datos subyacentes se hubieran actualizado. La mayoría de los usuarios de d3 que están preocupados con las visualizaciones de respuesta ya sabrán cómo construir sus eventos de actualización en funciones que son fáciles de llamar, como se muestra en el tema de introducción y este tema .

Lea Enfoques para crear gráficos d3.js responsivos en línea: https://riptutorial.com/es/d3-js/topic/4312/enfoques-para-crear-graficos-d3-js-responsivos

Capítulo 6: Gráficos SVG usando D3 js

Examples

Usando D3 js para crear elementos SVG

Aunque D3 no es específico para el manejo de elementos SVG, se usa ampliamente para crear y manipular complejas visualizaciones de datos basadas en SVG. D3 proporciona muchos métodos poderosos que ayudan a crear varias estructuras geométricas SVG con facilidad.

Se recomienda comprender primero los conceptos básicos de las especificaciones SVG, luego usar ejemplos extensos de D3 js para crear visualizaciones.

D3 js ejemplos

Fundamentos de SVG

Lea Gráficos SVG usando D3 js en línea: https://riptutorial.com/es/d3-js/topic/1538/graficos-svg-usando-d3-js

Capítulo 7: Haciendo robusto, responsivo y reutilizable (r3) para d3

Introducción

d3 es una biblioteca poderosa para crear gráficos interactivos; sin embargo, ese poder proviene de los usuarios que tienen que trabajar en un nivel más bajo que otras bibliotecas interactivas. En consecuencia, muchos de los ejemplos para los gráficos d3 están diseñados para demostrar cómo producir una cosa en particular, por ejemplo, bigotes para un cuadro y un gráfico de bigotes, mientras que a menudo se codifican los parámetros para que el código sea inflexible. El propósito de esta documentación es demostrar cómo hacer más código reutilizable para ahorrar tiempo en el futuro.

Examples

Gráfico de dispersión

Este ejemplo contiene más de 1000 líneas de código en total (demasiado para ser incrustado aquí). Por eso todo el código es accesible en http://blockbuilder.org/SumNeuron/956772481d4e625eec9a59fdb9fbe5b2 (alternativamente alojado en https://bl.ocks.org/SumNeuron/956772481d4e625eec9a59fdb9fbe5b2) . Tenga en cuenta que bl.ocks.org usa iframe para ver el cambio de tamaño que tendrá que hacer clic en el botón abierto (esquina inferior derecha del iframe). Dado que hay una gran cantidad de código, se ha dividido en varios archivos y el segmento de código relevante será referencia tanto por nombre de archivo como por número de línea. Por favor abre este ejemplo a medida que avanzamos.

¿Qué hace un gráfico?

Hay varios componentes centrales que entran en cualquier gráfico completo; a saber, estos incluyen:

- título
- hachas
- etiquetas de ejes
- los datos

Hay otros aspectos que pueden incluirse según el gráfico, por ejemplo, una leyenda del gráfico. Sin embargo, muchos de estos elementos pueden ser evitados con información sobre herramientas. Por esa razón, hay elementos específicos de gráficos interactivos, por ejemplo, botones para cambiar entre datos.

Dado que el contenido de nuestro gráfico será interactivo, sería apropiado que el gráfico sea

dinámico, por ejemplo, cambie el tamaño cuando cambie el tamaño de la ventana. SVG es escalable, por lo que podría permitir que su gráfico se amplíe manteniendo la perspectiva actual. Sin embargo, dependiendo de la perspectiva establecida, el gráfico puede ser demasiado pequeño para ser legible incluso si todavía hay suficiente espacio para el gráfico (por ejemplo, si el ancho es mayor que la altura). Por lo tanto, puede ser preferible simplemente volver a dibujar el gráfico en el tamaño restante.

Este ejemplo cubrirá cómo calcular dinámicamente la ubicación de los botones, títulos, ejes, etiquetas de ejes, así como también manejar conjuntos de datos de cantidades variables de datos.

Preparar

Configuración

Dado que nuestro objetivo es reutilizar el código, debemos crear un archivo de configuración que contenga opciones globales para aspectos de nuestro gráfico. Un ejemplo de este archivo de configuración es `charts_configuration.json` .

Si miramos este archivo, podemos ver que he incluido varios elementos que ya deberían tener un uso claro para cuando hagamos nuestro gráfico:

- archivos (almacena la cadena donde se guardan nuestros datos de gráficos)
- document_state (el botón seleccionado actualmente para nuestro gráfico)
- chart_ids (ID de html para los gráficos que haremos)
- svg (opciones para el svg, por ejemplo, tamaño)
- plot_attributes
    - título (establecer varios atributos de fuente)
    - información sobre herramientas (establecer varias propiedades de estilo de información sobre herramientas)
    - ejes (establecer varios atributos de fuente)
    - Botones (establecer varios atributos de fuente y estilo)
- parcelas
    - Dispersión (establezca varios aspectos de nuestro diagrama de dispersión, por ejemplo, radio del punto)
- colores (una paleta de colores específica para usar)

Funciones de ayuda

Además de configurar estos aspectos globales, necesitamos definir algunas funciones de ayuda. Estos se pueden encontrar en `helpers.js`

- `ajax_json` (carga archivos json de forma síncrona o asíncrona)
- `keys` (devuelve claves del objeto dado - equivalente a d3.keys ())
- `parseNumber` (un número general de `parseNumber` en caso de que no sepamos qué tipo o

número es)
- `typeofNumber` (devuelve el tipo de número)

index.html

Por último deberíamos configurar nuestro archivo html. Para el propósito de este ejemplo, pondremos nuestro gráfico en una etiqueta de `section` donde la `id` coincide con la identificación proporcionada en el archivo de configuración (línea 37). Dado que los porcentajes solo funcionan si se pueden calcular a partir de su miembro principal, también incluimos algunos estilos básicos (líneas 19-35)

Haciendo nuestro diagrama de dispersión

Vamos a abrir `make_scatter_chart.js` . Ahora prestemos mucha atención a la línea 2, donde muchas de las variables más importantes están predefinidas:

- svg - d3 selección de svg del gráfico
- chart_group - d3 selección del grupo dentro del svg en el que se colocarán los datos
- lienzo - aspectos centrales del extracto svg para mayor comodidad
- márgenes - los márgenes que debemos tener en cuenta
- maxi_draw_space los valores x e y más grandes en los que podemos dibujar nuestros datos
- doc_state - el estado actual del documento si estamos usando botones (en este ejemplo estamos)

Es posible que haya notado que no incluimos el svg en el html. Por lo tanto, antes de que podamos hacer algo con nuestro gráfico, debemos agregar el svg a `index.html` si aún no existe. Esto se logra en el archivo `make_svg.js` mediante la función `make_chart_svg` . Al ver `make_svg.js` , vemos que usamos la función auxiliar `parseNumber` en la configuración del gráfico para el ancho y la altura de svg. Si el número es un flotante, hace que el ancho y la altura del svg sean proporcionales al ancho y la altura de su sección. Si el número es un entero, simplemente lo establecerá en esos enteros.

Las líneas 6 a 11 prueban, en efecto, si esta es la primera llamada o no y establece el `chart_group` (y el estado del documento si es la primera llamada).

Las líneas 14 a 15 extraen los datos seleccionados actualmente mediante el botón en el que se hizo clic; línea 16 establece `data_extent` . Mientras que d3 tiene una función para extraer la extensión de datos, es *mi* preferencia almacenar la extensión de datos en esta variable.

Las líneas 27 a 38 contienen la magia que configura nuestra gráfica al hacer los márgenes, los botones, el título y los ejes. Todos estos están determinados dinámicamente y pueden parecer un poco complejos, por lo que veremos cada uno a su vez.

make_margins (en make_margins.js)

Podemos ver que el objeto de márgenes toma en cuenta algún espacio a la izquierda, derecha, arriba y abajo del gráfico (x.left, x.right, y.top, y.bottom respectivamente), el título, los botones y los ejes

También vemos que los márgenes de los ejes se actualizan en la línea 21.

¿Por qué hacemos esto? Bueno, a menos que especifiquemos el número de marcas, la marca etiqueta el tamaño de marca y la fuente de la etiqueta, no podríamos calcular el tamaño que necesitan los ejes. Incluso entonces tendríamos que estimar el espacio entre las etiquetas de tick y las ticks. Por lo tanto, es más fácil hacer algunos ejes ficticios utilizando nuestros datos, ver qué tan grandes son los elementos svg correspondientes y luego devolver el tamaño.

En realidad, solo necesitamos el ancho del eje y y la altura del eje x, que es lo que se almacena en axes.y y axes.x.

Con nuestros márgenes predeterminados establecidos, luego calculamos `max_drawing_space` (líneas 29-34 en make_margins.js)

make_buttons (en make_buttons.js)

La función crea un grupo para todos los botones y luego un grupo para cada botón, que a su vez almacena un círculo y un elemento de texto. La línea 37 - 85 calcula la posición de los botones. Para ello, puede ver si el texto a la derecha de la longitud de cada botón es más largo que el espacio permitido para que dibujemos (línea 75). Si es así, coloca el botón en una línea y actualiza los márgenes.

make_title (en make_title.js)

`make_title` es similar a make_buttons en el sentido de que automáticamente dividirá el título de su gráfico en varias líneas, y se dividirá si es necesario. Es un poco pirateado ya que no tiene la sofisticación del esquema de separación de palabras de TeX, pero funciona lo suficientemente bien. Si necesitamos más líneas de una, se actualizan los márgenes.

Con los botones, el título y los márgenes establecidos, podemos hacer nuestros ejes.

make_axes (en make_axes.js)

La lógica de make_axes refleja que para calcular el espacio que necesitan los ejes ficticios. Aquí, sin embargo, añade transiciones para cambiar entre ejes.

Finalmente nuestro diagrama de dispersión

Con toda esa configuración hecha, finalmente podemos hacer nuestro diagrama de dispersión. Dado que nuestros conjuntos de datos pueden tener una cantidad diferente de puntos, debemos tener esto en cuenta y aprovechar los eventos de entrada y salida de d3 en consecuencia. En la línea 40 se obtiene el número de puntos ya existentes. La instrucción if en la línea 45 - 59 agrega más elementos de círculo si tenemos más datos, o transiciona los elementos adicionales a una

esquina y luego los elimina si hay demasiados.

Una vez que sabemos que tenemos el número correcto de elementos, podemos hacer la transición de todos los elementos restantes a su posición correcta (línea 64)

Por último, agregamos información sobre herramientas en la línea 67 y 68. La función de información sobre herramientas está en `make_tooltip.js`

Cuadro de caja y bigotes

Para mostrar el valor de realizar funciones generalizadas como las del ejemplo anterior (make_title, make_axes, make_buttons, etc.), considere este cuadro y el gráfico de bigotes: https://bl.ocks.org/SumNeuron/262e37e2f932cf4b693c24a410ff

Si bien el código para hacer las cajas y los bigotes es más intensivo que solo colocar los puntos, vemos que las mismas funciones funcionan perfectamente.

Gráfico de barras

https://bl.ocks.org/SumNeuron/7989abb1749fc70b39f7b1e8dd192248

Lea Haciendo robusto, responsivo y reutilizable (r3) para d3 en línea: https://riptutorial.com/es/d3-js/topic/9849/haciendo-robusto--responsivo-y-reutilizable--r3--para-d3

Capítulo 8: patrón de actualización

Sintaxis

- selection.enter ()
- selection.exit ()
- selection.merge ()

Examples

Actualización de los datos: un ejemplo básico de selecciones de ingreso, actualización y salida

Crear un gráfico que muestre un conjunto de datos estático es relativamente simple. Por ejemplo, si tenemos esta matriz de objetos como datos:

```
var data = [
    {title: "A", value: 53},
    {title: "B", value: 12},
    {title: "C", value: 91},
    {title: "D", value: 24},
    {title: "E", value: 59}
];
```

Podemos crear un gráfico de barras donde cada barra representa una medida, llamada "título", y su ancho representa el valor de esa medida. Como este conjunto de datos no cambia, nuestro gráfico de barras solo tiene una selección "ingresar":

```
var bars = svg.selectAll(".bars")
    .data(data);

bars.enter()
    .append("rect")
    .attr("class", "bars")
    .attr("x", xScale(0))
    .attr("y", function(d){ return yScale(d.title)})
    .attr("width", 0)
    .attr("height", yScale.bandwidth())
    .transition()
    .duration(1000)
    .delay(function(d,i){ return i*200})
    .attr("width", function(d){ return xScale(d.value) - margin.left});
```

Aquí, estamos configurando el ancho de cada barra a 0 y, después de la transición, a su valor final.

Esta selección de entrada, solo, es suficiente para crear nuestro gráfico, que puede ver en este violín .

¿Pero qué pasa si mis datos cambian?

En este caso, tenemos que cambiar dinámicamente nuestro gráfico. La mejor manera de hacerlo es crear las selecciones "enter", "update" y "exit". Pero, antes de eso, tenemos que hacer algunos cambios en el código.

Primero, moveremos las partes cambiantes dentro de una función llamada `draw()` :

```
function draw(){
    //changing parts
};
```

Estas "partes cambiantes" incluyen:

1. Las selecciones de entrar, actualizar y salir;
2. El dominio de cada escala;
3. La transición del eje;

Dentro de esa función `draw()` , llamamos a otra función, que crea nuestros datos. Aquí, es solo una función que devuelve una matriz de 5 objetos, seleccionando aleatoriamente 5 letras de 10 (ordenando alfabéticamente) y, para cada uno, un valor entre 0 y 99:

```
function getData(){
    var title = "ABCDEFGHIJ".split("");
    var data = [];
    for(var i = 0; i < 5; i++){
        var index = Math.floor(Math.random()*title.length);
        data.push({title: title[index],
            value: Math.floor(Math.random()*100)});
        title.splice(index,1);
    }
    data = data.sort(function(a,b){ return d3.ascending(a.title,b.title)});
    return data;
};
```

Y ahora, pasemos a nuestras selecciones. Pero antes de eso, una advertencia: para mantener lo que llamamos *constancia del objeto* , debemos especificar una función clave como el segundo argumento de selection.data:

```
var bars = svg.selectAll(".bars")
    .data(data, function(d){ return d.title});
```

Sin eso, nuestras barras no realizarán una transición sin problemas, y sería difícil seguir los cambios en el eje (puede ver que se elimina el segundo argumento en el violín a continuación).

Entonces, después de configurar correctamente nuestras `var bars` , podemos tratar con nuestras selecciones. Esta es la selección de salida:

```
bars.exit()
    .transition()
    .duration(1000)
    .attr("width", 0)
```

```
    .remove();
```

Y estas son las selecciones de ingreso y actualización (en D3 v4.x, la selección de actualización se combina con la selección de ingreso mediante `merge`):

```
bars.enter()//this is the enter selection
    .append("rect")
    .attr("class", "bars")
    .attr("x", xScale(0) + 1)
    .attr("y", function(d){ return yScale(d.title)})
    .attr("width", 0)
    .attr("height", yScale.bandwidth())
    .attr("fill", function(d){ return color(letters.indexOf(d.title)+1)})
    .merge(bars)//and from now on, both the enter and the update selections
    .transition()
    .duration(1000)
    .delay(1000)
    .attr("y", function(d){ return yScale(d.title)})
    .attr("width", function(d){ return xScale(d.value) - margin.left});
```

Finalmente, llamamos a la función `draw()` cada vez que se hace clic en el botón:

```
d3.select("#myButton").on("click", draw);
```

Y este es el violín que muestra todas estas 3 selecciones en acción.

Fusionar selecciones

El patrón de actualización en D3 versión 3

Una comprensión correcta de cómo funcionan las selecciones "entrar", "actualizar" y "salir" es fundamental para cambiar correctamente el dataviz usando D3.

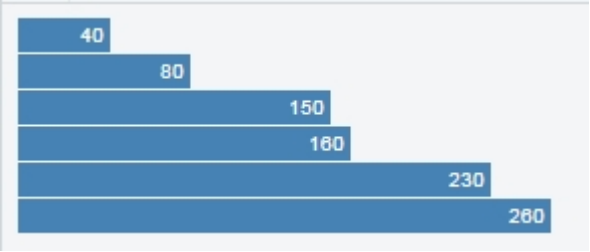
Desde la versión 3 de D3 (en realidad, desde la versión 2), este fragmento podría establecer las transiciones para las selecciones "entrar" y "actualizar" (demostración en vivo aquí):

```
var divs = body.selectAll("div")
    .data(data);//binding the data

divs.enter()//enter selection
    .append("div")
    .style("width", "0px");

divs.transition()
    .duration(1000)
    .style("width", function(d) { return d + "px"; })
    .attr("class", "divchart")
    .text(function(d) { return d; });
```

Dando este resultado después de la transición:



¿Pero qué sucede con el mismo código exactamente si usamos D3 versión 4? Puedes verlo en esta demo en vivo : ¡ *nada* !

¿Por qué?

Cambios en el patrón de actualización en D3 versión 4

Veamos el código. Primero, tenemos `divs` . Esta selección enlaza los datos a `<div>` .

```
var divs = body.selectAll("div")
    .data(data);
```

Luego, tenemos `divs.enter()` , que es una selección que contiene todos los datos con elementos no coincidentes. Esta selección contiene todos los divs en la primera vez que llamamos a la función `draw` , y establecemos sus anchos a cero.

```
divs.enter()
    .append("div")
    .style("width", "0px");
```

Luego tenemos `divs.transition()` , y aquí tenemos el comportamiento interesante: en la versión 3 de D3, `divs.transition()` hace que todo `<div>` en la selección "enter" cambie a su ancho final. ¡Pero eso no tiene sentido! `divs` no contiene la selección "entrar", y no debe modificar ningún elemento DOM.

Hay una razón por la que este extraño comportamiento se introdujo en la versión 2 y 3 de D3 (fuente aquí), y se “corrigió” en la versión 4 de D3. Por lo tanto, en la demostración en vivo de arriba, no sucede nada, ¡y se espera! Además, si hace clic en el botón, aparecerán las seis barras anteriores, porque ahora están en la selección "actualizar", y ya no en la selección "ingresar".

Para la transición que actúa sobre la selección "entrar", tenemos que crear variables separadas o, un enfoque más fácil, fusionando las selecciones :

```
divs.enter()//enter selection
    .append("div")
    .style("width", "0px")
    .merge(divs)//from now on, enter + update selections
    .transition().duration(1000).style("width", function(d) { return d + "px"; })
    .attr("class", "divchart")
    .text(function(d) { return d; });
```

Ahora, fusionamos las selecciones "enter" y "update". Vea cómo funciona en esta demostración en vivo .

Lea patrón de actualización en línea: https://riptutorial.com/es/d3-js/topic/5749/patron-de-actualizacion

Capítulo 9: Proyecciones D3

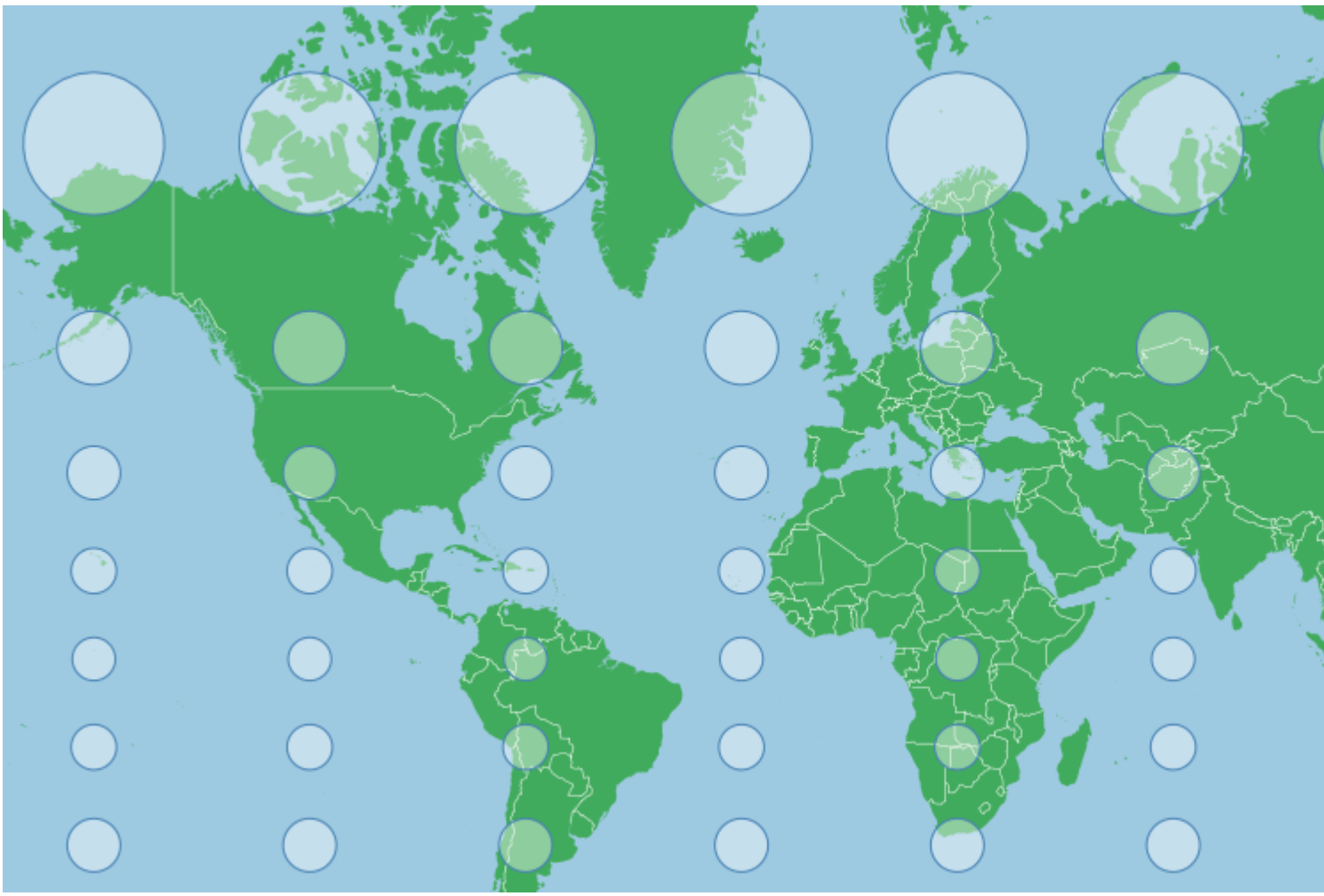
Examples

Proyecciones de Mercator

Una proyección de Mercator es una de las proyecciones más reconocibles utilizadas en los mapas. Como todas las proyecciones de mapas, tiene distorsión y, para un Mercator, la proyección es más notable en las regiones polares. Es una proyección cilíndrica, los meridianos se ejecutan verticalmente y las latitudes se ejecutan horizontalmente.

La escala depende del tamaño de su svg, para este ejemplo, todas las escalas utilizadas tienen un svg de 960 píxeles de ancho por 450 píxeles de alto.

El siguiente mapa muestra una Indicatriz de Tissot para una proyección de Mercator, cada círculo es en realidad del mismo tamaño, pero la proyección obviamente muestra que algunos son más grandes que otros:



Esta distorsión se debe a que la proyección intenta evitar un estiramiento unidimensional del mapa. A medida que los meridianos comienzan a fusionarse en los polos norte y sur, la distancia entre ellos comienza a acercarse a cero, pero la superficie de la proyección es rectangular (no el mapa, aunque también es rectangular) y no permite un cambio en la distancia. Entre los

meridianos en la proyección. Esto estiraría las características a lo largo del eje x cerca de los polos, distorsionando su forma. Para contrarrestar esto, un Mercator estira el eje y, así como uno se acerca a los polos, lo que hace que los indicadores sean circulares.

La proyección para el mapa de arriba es esencialmente la proyección predeterminada de Mercator modificada ligeramente:

```
var projection = d3.geoMercator()
    .scale(155)
    .center([0,40]) // Pan north 40 degrees
    .translate([width/2,height/2]);
```

Para centrar la proyección en un punto dado con una latitud conocida y una longitud conocida, puede desplazarse fácilmente a ese punto especificando el centro:

```
var projection = d3.geoMercator()
    .center([longitude,latitude])
```

Esto se desplazará a esa función (pero no a zoom) en la superficie proyectada (que se parece al mapa de arriba).

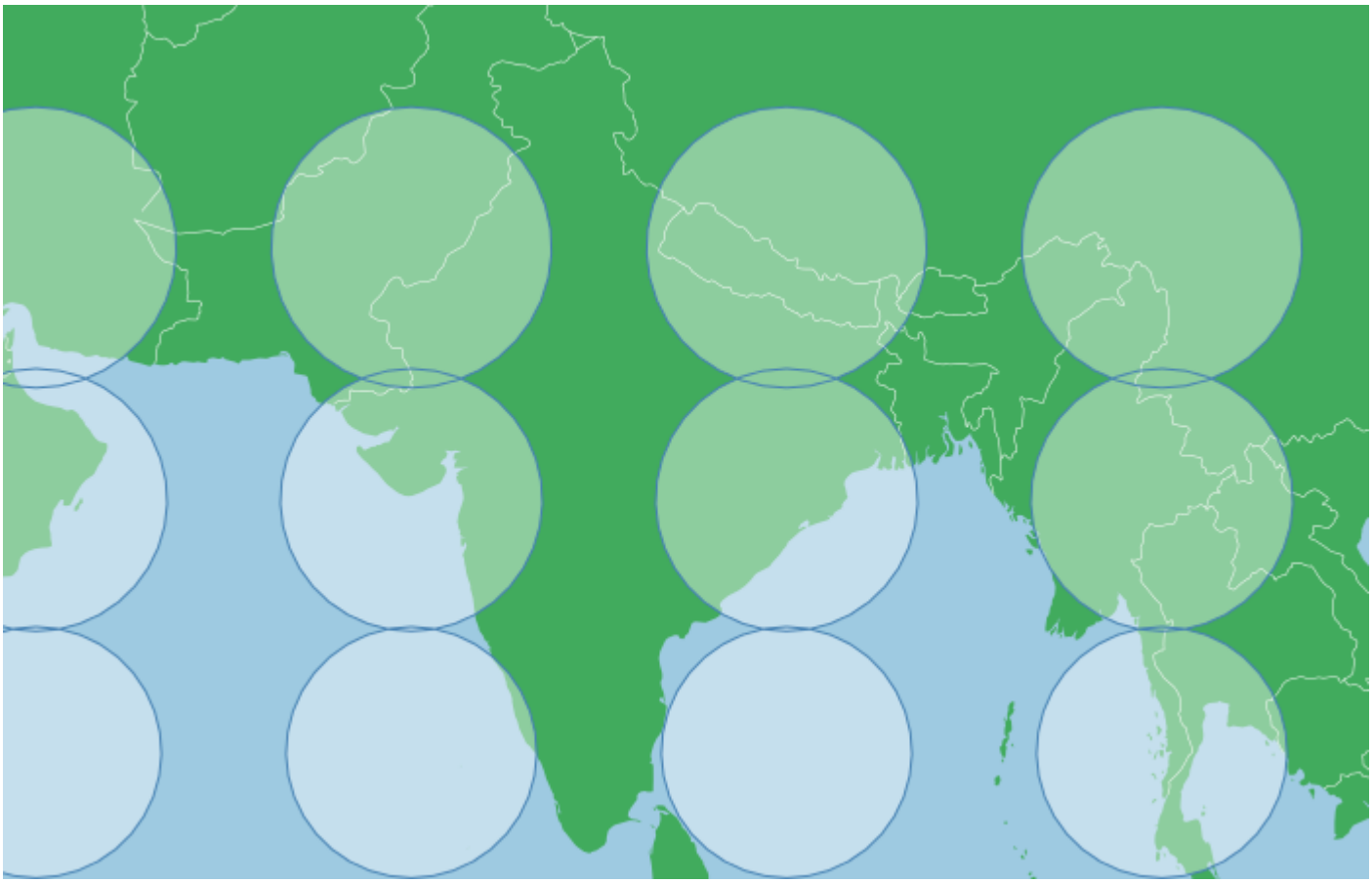
Las escalas deberán adaptarse al área de interés, los números más grandes son iguales a las funciones más grandes (mayor grado de acercamiento), los números más pequeños al contrario. Reducir el zoom puede ser una buena forma de orientarse para ver dónde se encuentran sus características en relación con el punto en el que se ha centrado, si tiene problemas para encontrarlas.

Debido a la naturaleza de una proyección de Mercator, las áreas cercanas al ecuador o en latitudes bajas funcionarán mejor con este tipo de proyección, mientras que las áreas polares pueden estar altamente distorsionadas. La distorsión es uniforme a lo largo de cualquier línea horizontal, por lo que las áreas que son amplias pero no altas también pueden ser buenas, mientras que las áreas que tienen una gran diferencia entre sus extremos norte y sur tienen más distorsión visual.

Para India, por ejemplo, podríamos usar:

```
var projection = d3.geoMercator()
    .scale(800)
    .center([77,21])
    .translate([width/2,height/2]);
```

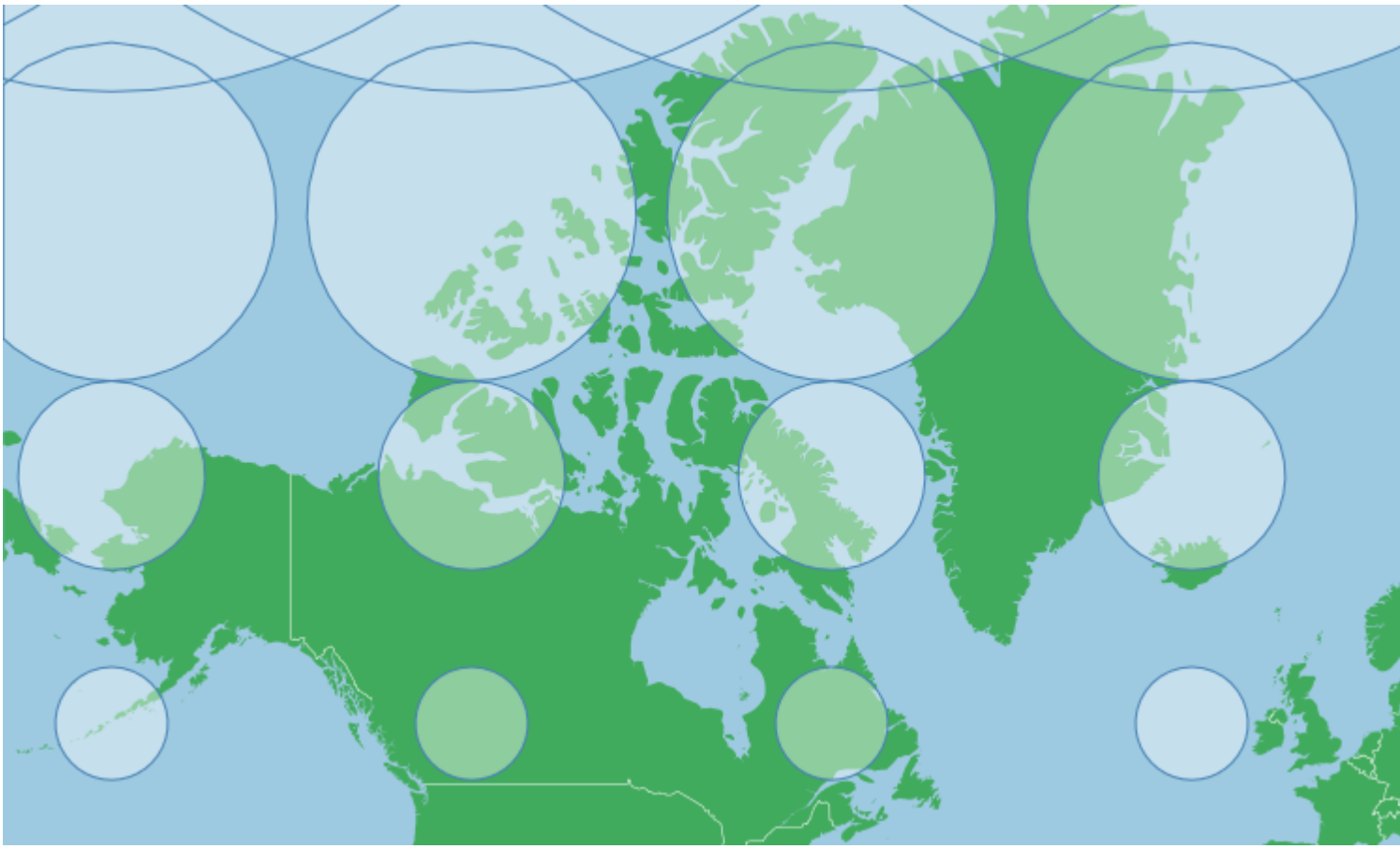
Lo que nos da (de nuevo con una indicación de Tissot para mostrar la distorsión):



Esto tiene un bajo nivel de distorsión, pero los círculos son en gran medida del mismo tamaño (puede ver una mayor superposición entre las dos filas superiores que las dos filas inferiores, por lo que la distorsión es visible). En general, sin embargo, el mapa muestra una forma familiar para la India.

La distorsión en el área no es lineal, se exagera en gran medida hacia los polos, por lo que Canadá con extremos norte y sur bastante alejados y una posición bastante cerca de los polos significa que la distorsión puede ser insostenible:

```
var projection = d3.geoMercator()
    .scale(225)
    .center([-95,69.75])
    .translate([width/2,height/2]);
```

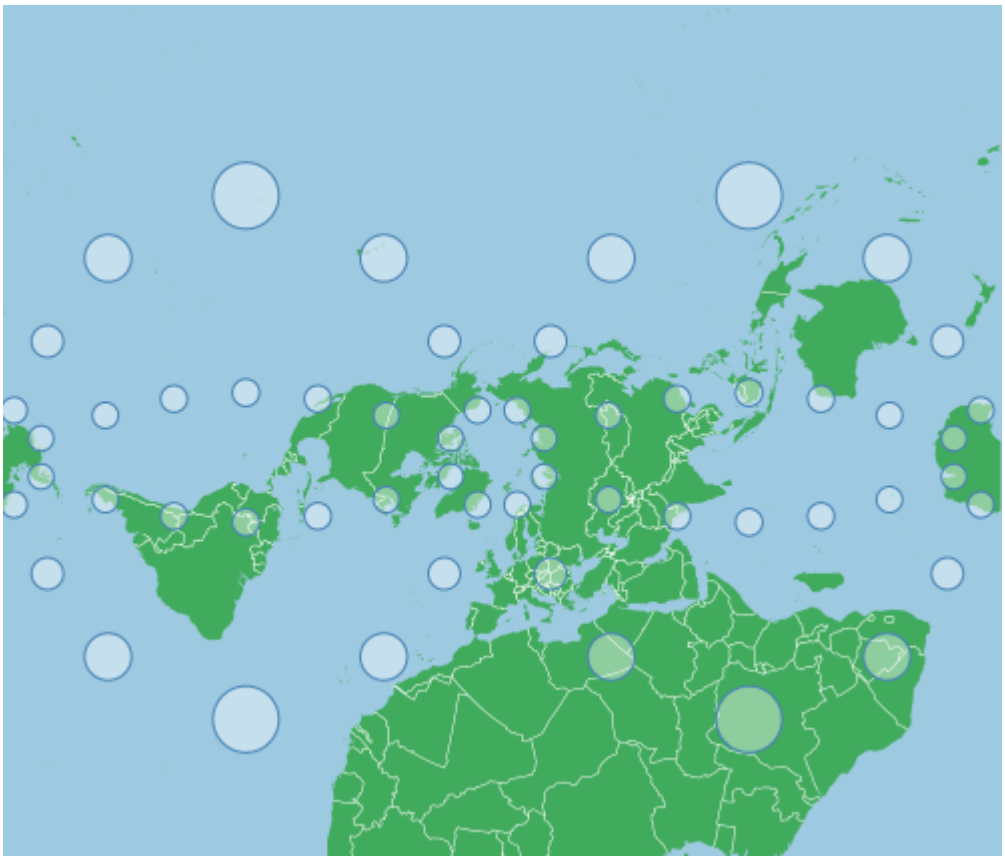


Esta proyección hace que Groenlandia se vea tan grande como Canadá, cuando en realidad Canadá es casi cinco veces más grande que Groenlandia. Esto se debe simplemente a que Groenlandia está más al norte que la mayor parte de Canadá (lo siento, Ontario, parece haber cortado parte de su extremo sur).

Debido a que el eje y se estira considerablemente cerca de los polos en un Mercator, esta proyección usa un punto considerablemente al norte del centro geográfico de Canadá. Si se trata de áreas de latitud alta, es posible que deba adaptar su punto de centrado para tener en cuenta este estiramiento.

Si necesita una proyección de Mercator para áreas polares, hay una manera de minimizar la distorsión y seguir utilizando una proyección de Mercator. Puedes lograr esto girando el globo. Si gira el eje x en el Mercator predeterminado, parecerá que gira hacia la izquierda o hacia la derecha (simplemente gire el globo en el cilindro que está proyectando), si, sin embargo, cambia el eje y del Mercator predeterminado, puede Gira la tierra de lado o hacia cualquier otro ángulo. Aquí hay un Mercator con una rotación ay de -90 grados:

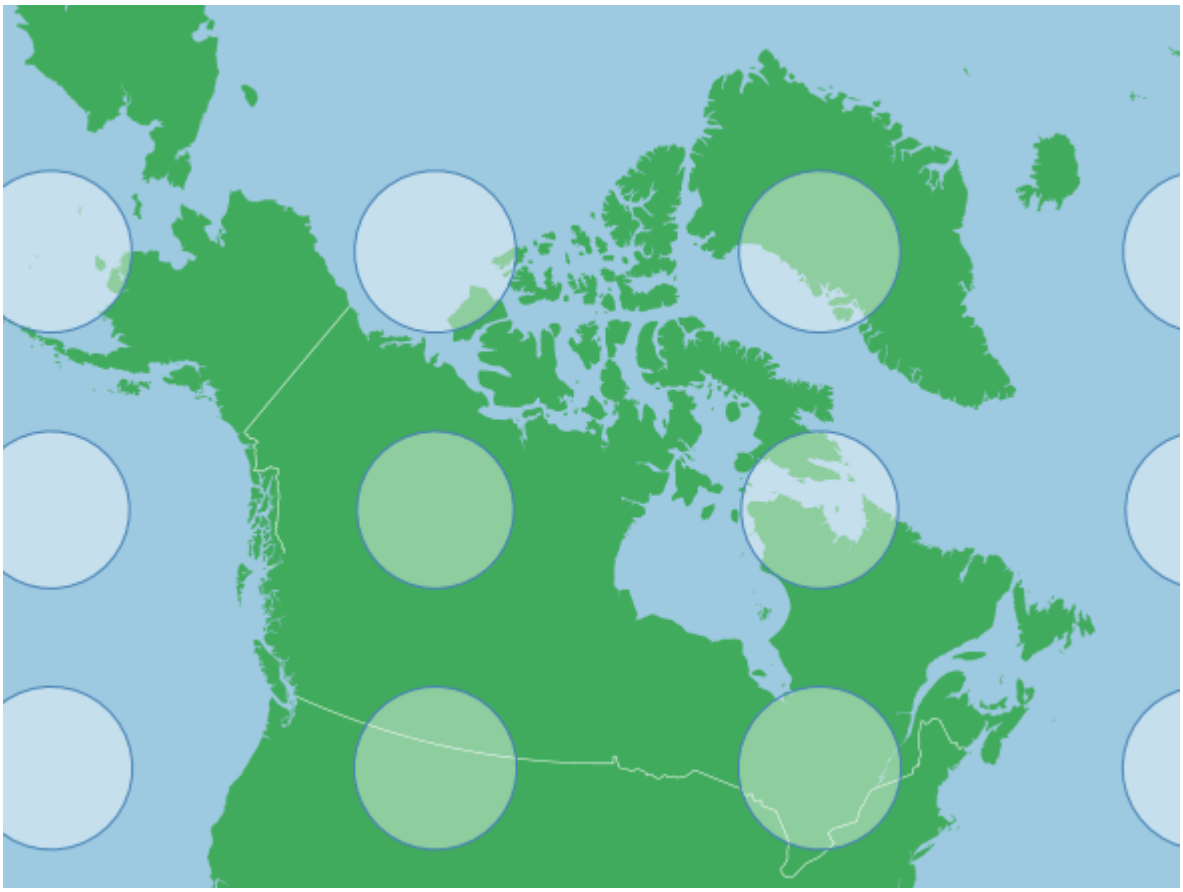
```
var projection = d3.geoMercator()
.scale(155)
.rotate([0,-90]);
.translate([width/2,height/2]);
```



Los puntos indicatrix están en las mismas ubicaciones que el primer mapa de arriba. La distorsión sigue aumentando a medida que se llega a la parte superior o inferior del mapa. Así es como aparecería un mapa de Mercator predeterminado si la Tierra girara alrededor de un Polo Norte en [0,0] y un Polo Sur en [180,0], la rotación ha girado el cilindro que estamos proyectando a 90 grados con respecto a los polos. Tenga en cuenta que los polos ya no tienen una distorsión insostenible, esto presenta un método alternativo para proyectar áreas cercanas a los polos sin demasiada distorsión en el área.

Usando Canadá como ejemplo nuevamente, podemos rotar el mapa a una coordenada central, y esto minimizará la distorsión en el área. Para hacerlo, podemos rotar nuevamente hacia un punto de centrado, pero esto requiere un paso adicional. Con el centrado nos movemos hacia una característica, con la rotación movemos la tierra debajo de nosotros, por lo que necesitamos el negativo de nuestra coordenada de centrado:

```
var projection = d3.geoMercator()
    .scale(500)
    .rotate([96,-64.15])
    .translate([width/2,height/2]);
```



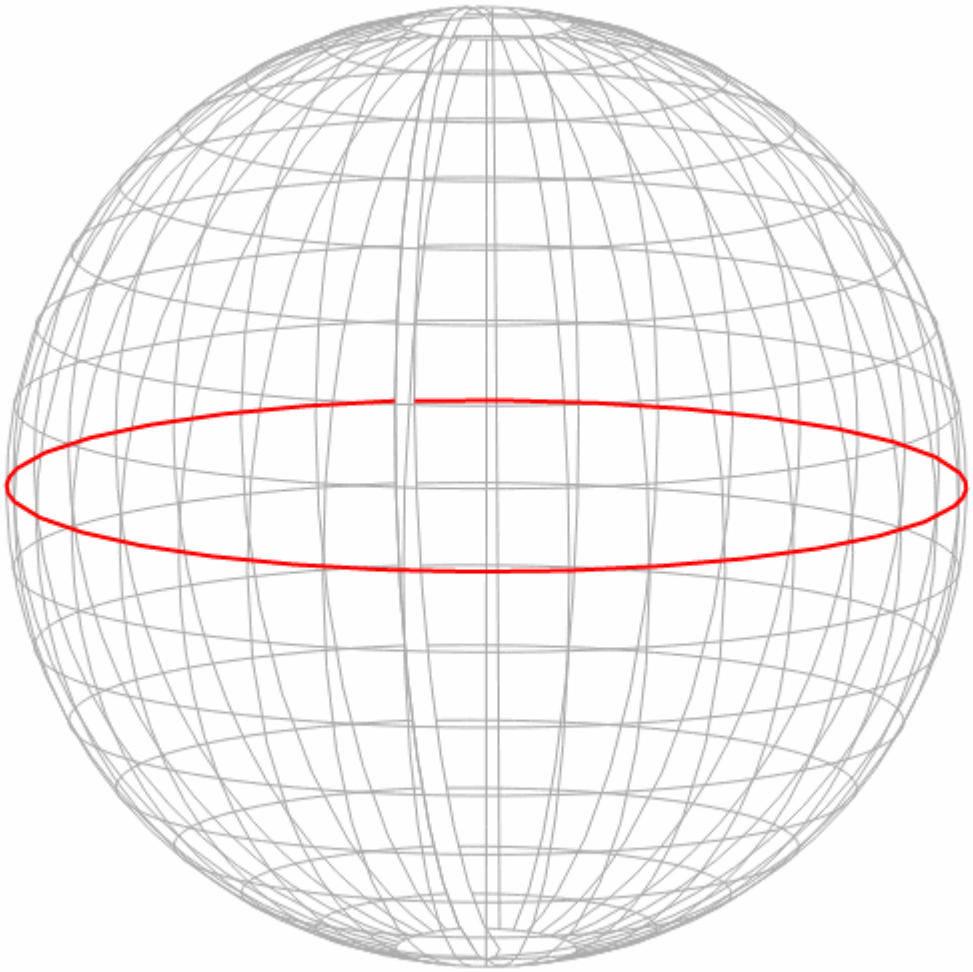
Tenga en cuenta que la indicación de Tissot está mostrando una distorsión baja en el área ahora. El factor de escala también es mucho más grande que antes, ya que este Canadá se encuentra ahora en el origen de la proyección, y en la mitad del mapa las características son más pequeñas que en la parte superior o inferior (consulte la primera indicación). No necesitamos `[-96,64.15]` porque el punto central u origen de esta proyección está en `[-96,64.15]` , el centrado nos alejaría de este punto.

Proyecciones Albers

Una proyección de Albers, o más bien, una proyección cónica de área uniforme de Albers, es una proyección cónica común y un proyecto oficial de varias jurisdicciones y organizaciones como la oficina de censos de EE. UU. Y la provincia de Columbia Británica en Canadá. Conserva el área a expensas de otros aspectos del mapa como la forma, el ángulo y la distancia.

Propiedades generales

La transformación general se captura en el siguiente gif:



(Basado en el bloque de Mike Bostock)

La proyección de Albers minimiza la distorsión en torno a dos paralelos estándar. Estos paralelos representan donde la proyección cónica cruza la superficie de la tierra.

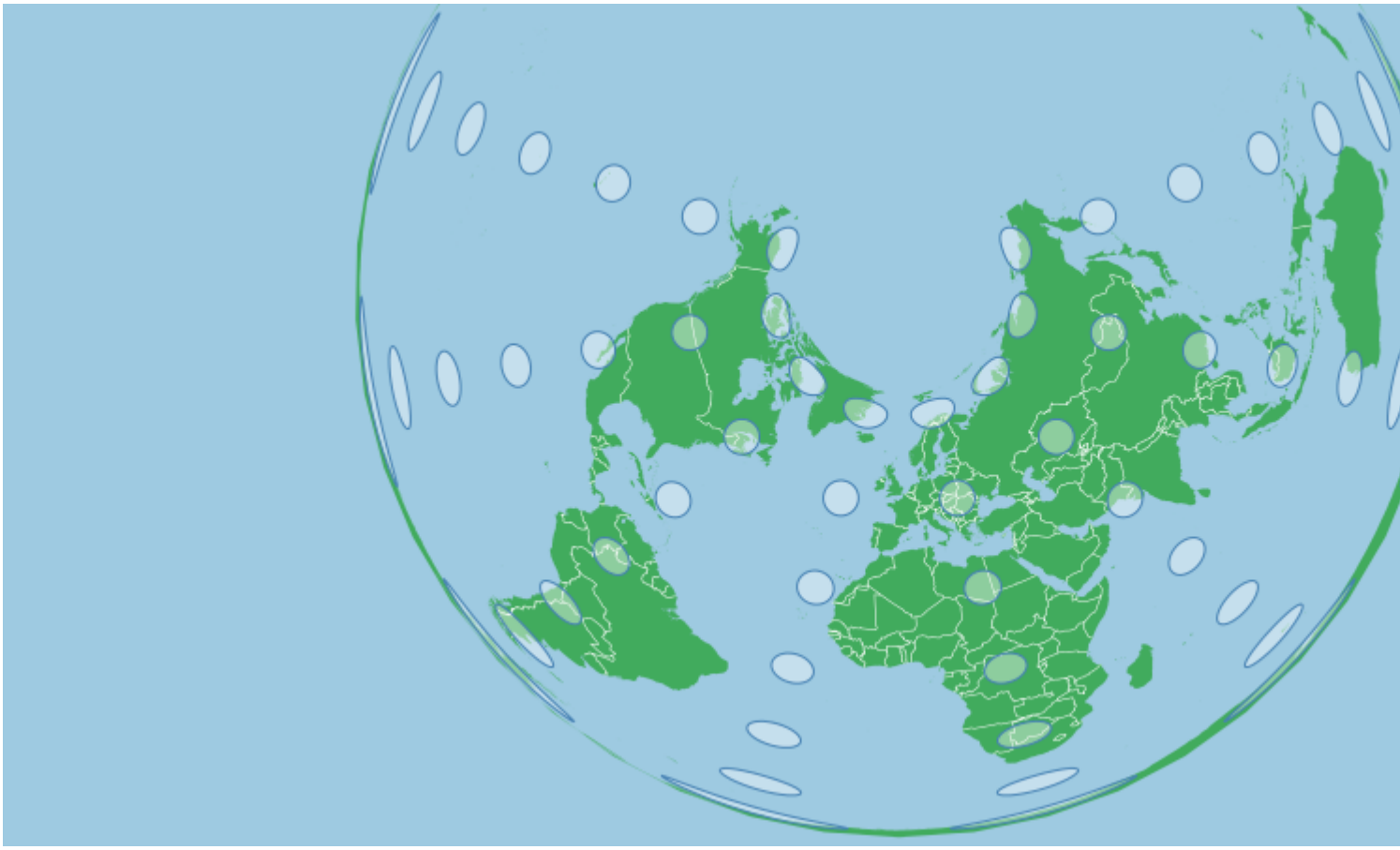
Para este ejemplo, todas las escalas se utilizan con dimensiones de svg de 960 píxeles de ancho por 450 píxeles de alto, la escala cambiará con estas dimensiones

El siguiente mapa muestra una Indicatriz de Tissot para una proyección de Albers con paralelos estándar de 10 y 20 grados norte. Cada círculo es en realidad el mismo tamaño y forma, pero la proyección del mapa los distorsionará en forma (no área). Observe cómo, aproximadamente a unos 10 a 20 grados norte, los indicadores son más redondos que en otros lugares:

Esto fue creado con la siguiente proyección:

```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,0])
    .rotate([0,0])
    .parallels([10,20])
    .translate([width/2,height/2]);
```

Si utilizamos paralelos que en las altitudes más altas, el grado de formación de arco en la proyección aumenta. Las siguientes imágenes utilizan los paralelos de 50 y 60 grados al norte:



```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,70]) // shifted up so that the projection is more visible
    .rotate([0,0])
    .parallels([40,50])
    .translate([width/2,height/2]);
```

Si tuviéramos paralelos negativos (del sur), el mapa sería cóncavo en lugar de arriba. Si un paralelo es norte y otro sur, el mapa será cóncavo hacia el paralelo más alto / más extremo, si están a la misma distancia del ecuador, entonces el mapa no será cóncavo en ninguna dirección.

Eligiendo paralelos

Como los paralelos marcan las áreas con la menor distorsión, deben elegirse en función de su área de interés. Si su área de interés se extiende de 10 grados al norte a 20 grados al norte, entonces la elección de paralelos de 13 y 17 minimizará la distorsión en todo el mapa (ya que la distorsión se minimiza a ambos lados de estos paralelos).

Los paralelos no deben ser los límites extremos norte y sur de su área de interés. *Los paralelos pueden tener el mismo valor si solo desea que la proyección intersecte la superficie de la tierra una vez.*

Las referencias y definiciones de proyección incluyen datos paralelos que puede utilizar para recrear proyecciones estandarizadas.

Centrado y Rotación

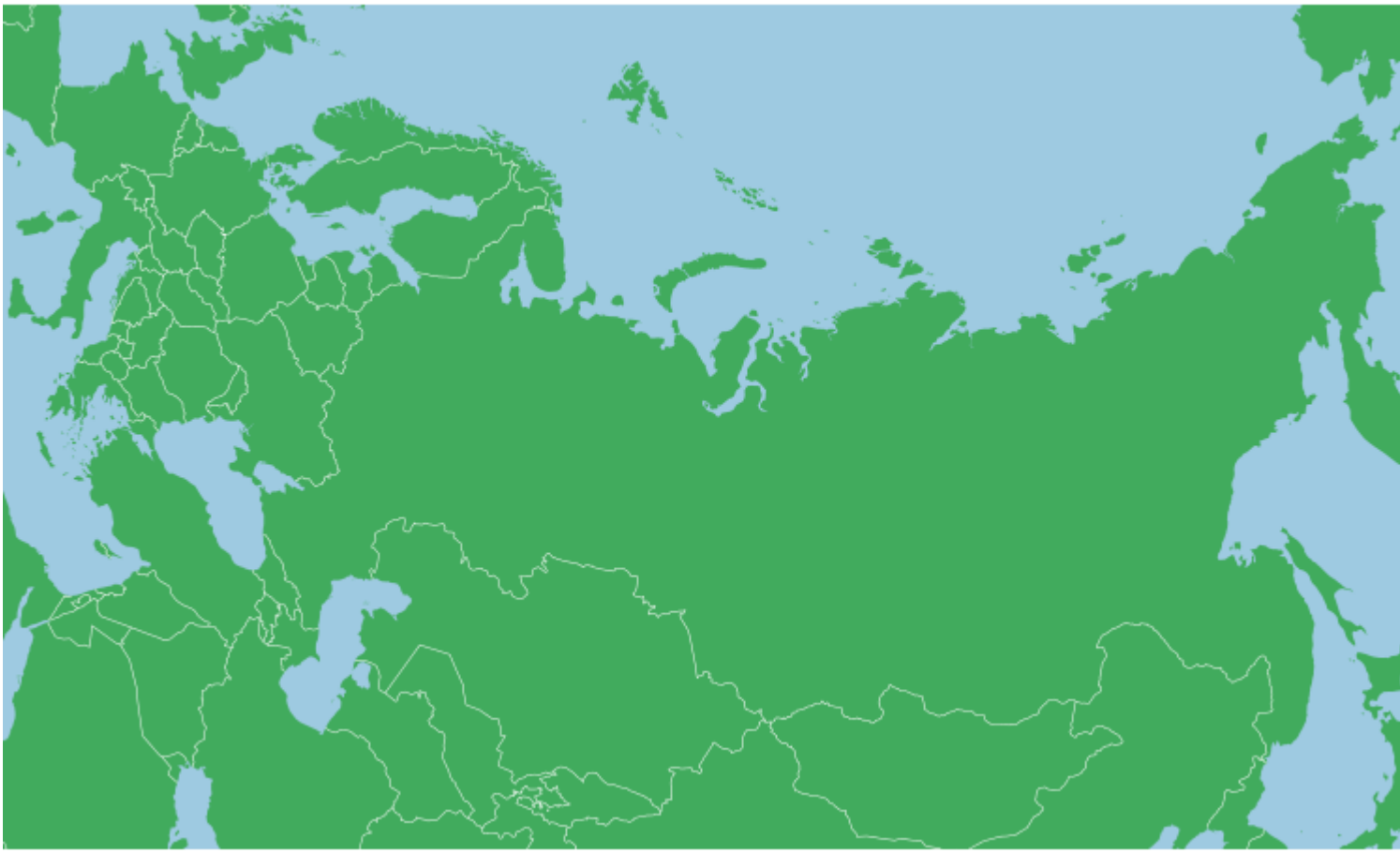
Una vez que se seleccionan los paralelos, el mapa debe posicionarse de manera que el área de interés se alinee correctamente. Si usa solo `projection.center([x,y])` , el mapa simplemente se desplazará al punto seleccionado y no tendrá lugar ninguna otra transformación. Si el área objetivo es Rusia, la panorámica podría no ser ideal:

```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,50]) // Shifted up so the projection is more visible
    .rotate([0,0])
    .parallels([50,60])
    .translate([width/2,height/2]);
```

El meridiano central de una proyección de Albers es vertical, y necesitamos rotar la tierra debajo de la proyección para cambiar el meridiano central. La rotación para una proyección de Alber es el método para centrar una proyección en el eje x (o por longitud). Y a medida que la tierra gira por debajo de la proyección, utilizamos el negativo de la longitud que queremos centrar. Para Rusia, esto podría ser de unos 100 grados al este, por lo que giraremos el globo 100 grados hacia el otro lado.

```
var projection = d3.geoAlbers()
    .scale(120)
    .center([0,60])
    .rotate([-100,0])
    .parallels([50,60])
```

Ahora podemos desplazarnos hacia arriba y hacia abajo y las características a lo largo y cerca del meridiano central estarán en posición vertical. *Si usted* `.center()` *en el eje x, su centrado será relativo al meridiano central establecido por la rotación* . Para Rusia es posible que desee desplazarse un poco hacia el Norte y acercar un poco:



```
var projection = d3.geoAlbers()
    .scale(500)
    .center([0,65])
    .rotate([-100,0])
    .parallels([50,60])
```

Para una función como Rusia, el arco del mapa significa que los bordes más alejados del país se extenderán alrededor del polo, lo que significa que el punto central no puede ser el centroide de la función, ya que es posible que deba desplazarse más hacia el centro. Norte o sur de lo habitual.

Con la Tissots Indicatrix, podemos ver algo de aplanamiento cerca del polo en sí, pero esa forma es bastante cierta en toda la zona de interés (recuerde que para el tamaño de Rusia, la distorsión es bastante mínima, sería mucho menor para las funciones más pequeñas):

Parámetros predeterminados

A diferencia de la mayoría de las demás proyecciones, la proyección d3.geoAlbers viene con parámetros predeterminados que no son .rotate ([0,0]) y .center ([0,0]), la proyección predeterminada se centra y gira para los Estados Unidos. Esto también es válido para `.parallels()` . Por lo tanto, si alguno de estos no está configurado, se establecerán de forma predeterminada valores no cero.

Resumen

Una proyección de Albers se establece generalmente con los siguientes parámetros:

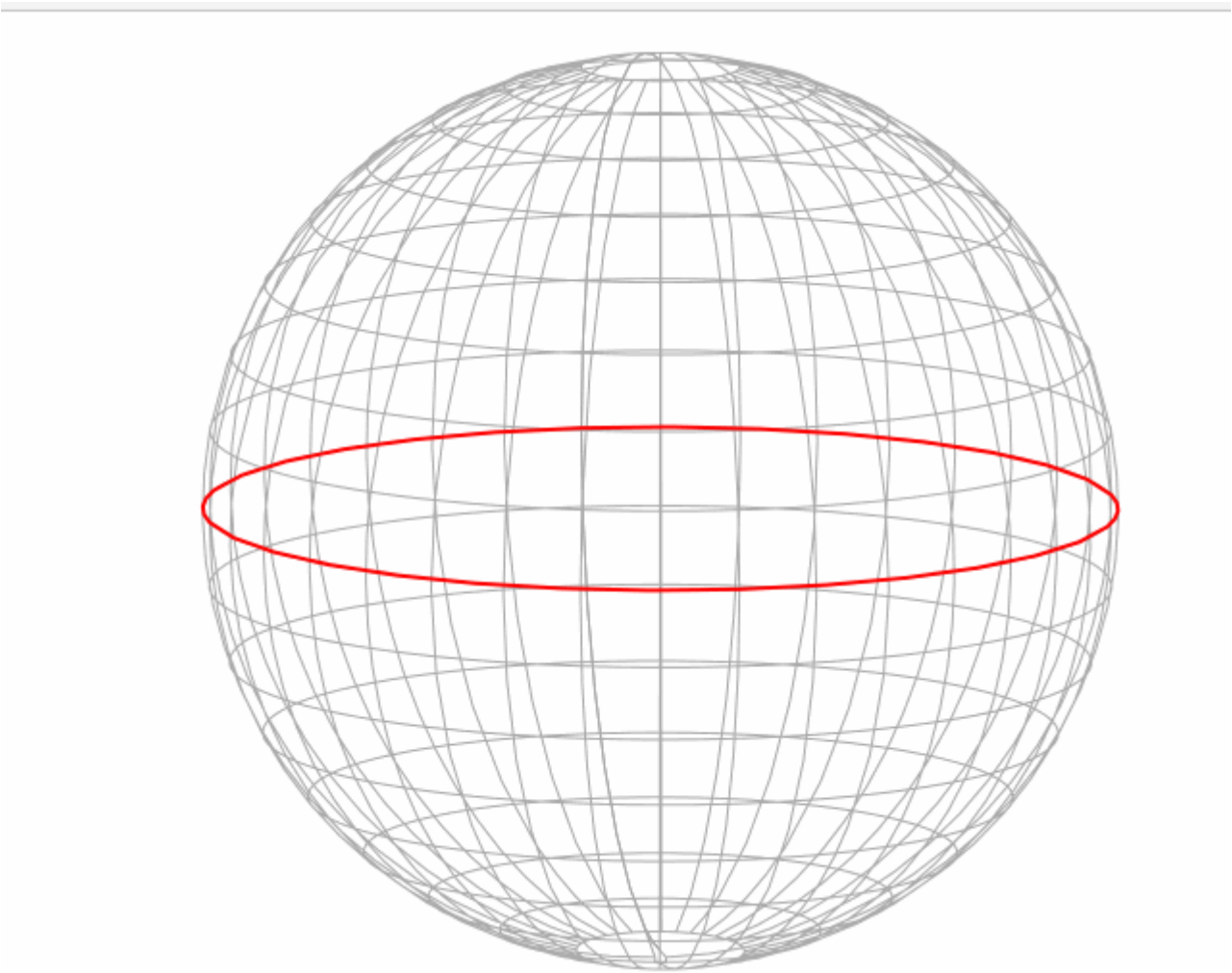
```
var projection = d3.geoAlbers()
    .rotate([-x,0])
    .center([0,y])
    .parallels([a,b]);
```

Donde a y b son iguales a los dos paralelos.

Proyecciones equidistantes azimutales

Propiedades generales:

Una proyección equidistante azimutal se reconoce mejor cuando se usa en áreas polares. Se utiliza en el emblema de la ONU . Desde el punto central, se conservan el ángulo y la distancia. Pero la proyección distorsionará la forma y el área para lograr esto, especialmente a medida que uno se aleja del centro. Del mismo modo, la distancia y el ángulo no son ciertos en lugares que no sean el centro. La proyección cae dentro de la categoría azimutal (en lugar de cilíndrica (Mercator) o cónica (Albers). Esta proyección proyecta la tierra como un disco:



(Basado en el bloque de Mike Bostock. *Centrado en el Polo Norte, ignorar el artefacto triangular en la parte superior de la imagen una vez desplegado*)

La escala depende del tamaño de su svg, para este ejemplo, todas las escalas utilizadas están dentro de un svg de 960 píxeles de ancho por 450 píxeles de alto (y la pantalla está recortada para un cuadrado donde sea necesario), a menos que se especifique lo contrario.

El siguiente mapa muestra una Indicatriz de Tissot para una proyección equidistante azimutal:

Esto fue creado con la siguiente proyección:

```
var projection = d3.geoAzimuthalEquidistant()
   .scale(70)
   .center([0,0])
   .rotate([0,0])
   .translate([width/2,height/2]);
```

Centrado y Rotación:

El centrado simplemente desplazará un mapa pero no cambiará su composición general. Centrar un equidistante azimutal en el Polo Norte mientras no se cambian otros parámetros o en cero moverá el Polo Norte al centro de la pantalla, pero de lo contrario no cambiará el mapa de arriba.

Para centrar adecuadamente un área, necesitas girarla. Al igual que con cualquier rotación en d3, es mejor pensar que se mueve la tierra debajo de la proyección, por lo que al rotar la tierra -90 grados debajo del mapa en el eje y, en realidad, se ubicará el polo norte en el centro:

```
var projection = d3.geoAzimuthalEquidistant()
   .scale(70)
   .center([0,0])
   .rotate([0,-90])
   .translate([width/2,height/2]);
```

Del mismo modo, la rotación en el eje x se comporta de manera similar. Por ejemplo, para rotar el mapa de manera que el ártico canadiense esté en posición vertical, mientras se centra en el polo norte, podemos usar una proyección como esta:

```
var projection = d3.geoAzimuthalEquidistant()
    .scale(400)
    .center([0,0])
    .rotate([100,-90])
    .translate([width/2,height/2]);
```

Este mapa utiliza un svg 600x600

En general, esta simplicidad hace que una proyección equidistante azimutal sea más fácil de configurar, simplemente use la rotación. Una proyección típica se verá así:

```
var projection = d3.geoProjection()
   .center([0,0])
   .rotate([-x,-y])
   .scale(k)
   .translate([width/2,height/2]);
```

Lea Proyecciones D3 en línea: https://riptutorial.com/es/d3-js/topic/9001/proyecciones-d3

Capítulo 10: Trozos escogidos

Sintaxis

- d3 **seleccionar** (selector)
- d3 **seleccionarAll** (selector)
- *seleccion* **seleccionar** (selector)
- *seleccion* **seleccionarAll** (selector)
- *seleccion* **filtro** (filtro)
- *seleccion* **fusionar** (otro)

Observaciones

Lecturas relacionadas:

- Cómo funcionan las selecciones - Mike Bostock
- d3-selection README

Examples

Selección básica y modificaciones.

Si está familiarizado con la sintaxis de jQuery y Sizzle, las selecciones de d3 no deberían ser muy diferentes. d3 imita la API de selectores de W3C para facilitar la interacción con los elementos.

Para un ejemplo básico, seleccionar todos `<p>` y agregar un cambio a cada uno de ellos:

```
d3.selectAll('p')
  .attr('class','textClass')
  .style('color', 'white');
```

En pocas palabras, esto es relativamente lo mismo que hacer en jQuery

```
$('p')
  .attr('class','textClass')
  .css('color, 'white')
```

En general, comenzará con una sola selección en su contenedor div para agregar un elemento SVG que se asignará a una variable (comúnmente llamada svg).

```
var svg = d3.select('#divID').append('svg');
```

Desde aquí podemos solicitar a `svg` que realice nuestras subselecciones de múltiples objetos (incluso si aún no existen).

```
svg.selectAll('path')
```

Diferentes selectores

Puede seleccionar elementos con diferentes selectores:

- por etiqueta: `"div"`
- por clase: `".class"`
- por id: `"#id"`
- por atributo: `"[color=blue]"`
- Selectores múltiples (OR): `"div1, div2, class1"`
- selectores múltiples (AND): `"div1 div2 class1"`

Selección simple de datos limitados

```
var myData = [
    { name: "test1", value: "ok" },
    { name: "test2", value: "nok" }
]

// We have to select elements (here div.samples)
// and assign data. The second parameter of data() is really important,
// it will determine the "key" to identify part of data (datum) attached to an
// element.
var mySelection = d3.select(document.body).selectAll("div.samples") // <- a selection
                    .data(myData, function(d){ return d.name; }); // <- data binding

// The "update" state is when a datum of data array has already
// an existing element associated.
mySelection.attr("class", "samples update")

// A datum is in the "enter" state when it's not assigned
// to an existing element (based on the key param of data())
// i.e. new elements in data array with a new key (here "name")
mySelection.enter().append("div")
    .attr("class", "samples enter")
    .text(function(d){ return d.name; });

// The "exit" state is when the bounded datum of an element
// is not in the data array
// i.e. removal of a row (or modifying "name" attribute)
// if we change "test1" to "test3", "test1" bounded
//          element will figure in exit() selection
// "test3" bounded element will be created in the enter() selection
mySelection.exit().attr("class", "samples remove");
```

El papel de los marcadores de posición en las selecciones "entrar"

¿Qué es una selección de entrada?

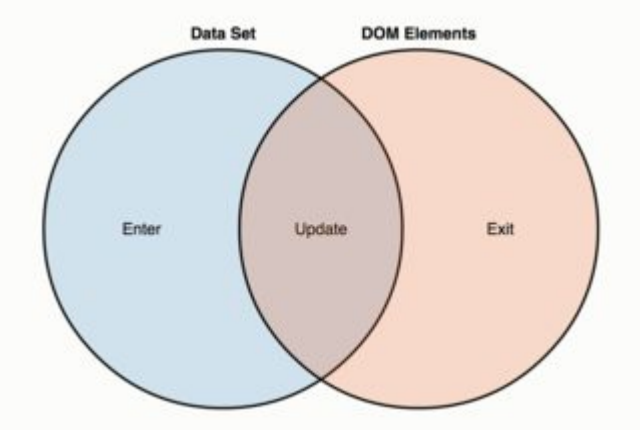
En D3.js, cuando uno enlaza datos a elementos DOM, son posibles tres situaciones:

1. El número de elementos y el número de puntos de datos son los mismos;
2. Hay más elementos que puntos de datos;

3. Hay más puntos de datos que elementos;

En la situación # 3, todos los puntos de datos sin un elemento DOM correspondiente pertenecen a la selección de *entrada* . Por lo tanto, en D3.js, *ingrese* selecciones son selecciones que, después de unir elementos a los datos, contienen todos los datos que no coinciden con ningún elemento DOM. Si usamos una función de `append` en una selección de *ingreso* , D3 creará nuevos elementos que vincularán esos datos para nosotros.

Este es un diagrama de Venn que explica las posibles situaciones con respecto a la cantidad de puntos de datos / cantidad de elementos DOM:



Como podemos ver, la selección de *ingreso* es el área azul a la izquierda: puntos de datos sin los elementos DOM correspondientes.

La estructura de la selección enter.

Normalmente, una selección de *entrada* tiene estos 4 pasos:

1. `selectAll` : Seleccionar elementos en el DOM;
2. `data` : cuenta y analiza los datos;
3. `enter` : comparando la selección con los datos, crea nuevos elementos;
4. `append` : `append` los elementos reales en el DOM;

Este es un ejemplo muy básico (mire los 4 pasos en los `var divs`):

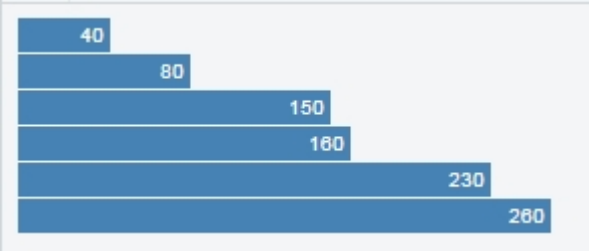
```
var data = [40, 80, 150, 160, 230, 260];

var body = d3.select("body");

var divs = body.selectAll("div")
    .data(data)
    .enter()
    .append("div");

divs.style("width", function(d) { return d + "px"; })
    .attr("class", "divchart")
    .text(function(d) { return d; });
```

Y este es el resultado (jsfiddle aquí):



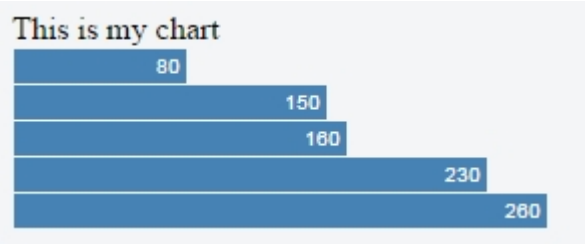
Observe que, en este caso, usamos `selectAll("div")` como la primera línea en nuestra variable de selección "enter". Tenemos un conjunto de datos con 6 valores, y D3 creó 6 divs para nosotros.

El papel de los marcadores de posición

Pero supongamos que ya tenemos un div en nuestro documento, algo como `<div>This is my chart</div>` en la parte superior. En ese caso, cuando escribamos:

```
body.selectAll("div")
```

Estamos seleccionando ese div existente. Por lo tanto, nuestra selección de entrada tendrá solo 5 datos sin elementos coincidentes. Por ejemplo, en este jsfiddle , donde ya existe una división en el HTML ("Este es mi gráfico"), este será el resultado:



Ya no vemos el valor "40": nuestra primera "barra" desapareció, y el motivo es que nuestra selección "enter" ahora tiene solo 5 elementos.

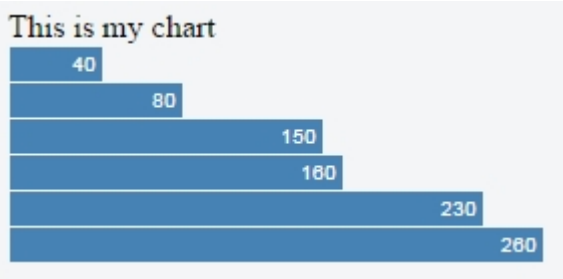
Lo que tenemos que entender aquí es que en la primera línea de nuestra variable de selección de ingreso, `selectAll("div")` , esos divs son solo *marcadores de posición* . No tenemos que seleccionar todos los `divs` si estamos agregando `divs` , o todo el `circle` si estamos agregando `circle` . Podemos seleccionar diferentes cosas. Y, si no planeamos tener una "actualización" o una selección de "salida", podemos seleccionar *cualquier cosa* :

```
var divs = body.selectAll(".foo")//this class doesn't exist, and never will!
    .data(data)
    .enter()
    .append("div");
```

Haciendo esto, estamos seleccionando todo el ".foo". Aquí, "foo" es una clase que no solo no existe, sino que nunca se creó en ninguna otra parte del código. Pero no importa, esto es sólo un marcador de posición. La lógica es esta:

> Si en su selección "enter" selecciona algo que no existe, su selección "enter" *siempre* contendrá todos sus datos.

Ahora, seleccionando `.foo` , nuestra selección "enter" tiene 6 elementos, incluso si ya tenemos un div en el documento:



Y aquí está el jsfiddle correspondiente .

Seleccionando `null`

Por mucho, la mejor manera de garantizar que no está seleccionando nada es seleccionando `null` . No solo eso, sino que esta alternativa es mucho más rápida que cualquier otra.

Por lo tanto, para una selección de entrada, simplemente haga:

```
selection.selectAll(null)
    .data(data)
    .enter()
    .append(element);
```

Aquí hay un violín de demostración: https://jsfiddle.net/gerardofurtado/th6s160p/

Conclusión

Cuando se trata de selecciones de "ingreso", tenga mucho cuidado de no seleccionar algo que ya existe. Puede usar cualquier cosa en su selección de `selectAll` , incluso cosas que no existen y que nunca existirán (si no planea tener una selección de "actualización" o "salida").

El código de los ejemplos se basa en este código de Mike Bostock:
https://bl.ocks.org/mbostock/7322386

Usando "esto" con una función de flecha

La mayoría de las funciones en D3.js aceptan una función anónima como argumento. Los ejemplos comunes son `.attr` , `.style` , `.text` , `.on` y `.data` , pero la lista es mucho más grande que eso.

En tales casos, la función anónima se evalúa para cada elemento seleccionado, en orden, que se pasa:

1. El dato actual (`d`)
2. El índice actual (`i`)
3. El grupo actual (`nodes`)
4. `this` como el elemento DOM actual.

El dato, el índice y el grupo actual se pasan como argumentos, el famoso primer, segundo y tercer argumento en D3.js (cuyos parámetros son tradicionalmente llamado `d` , `i` y `p` en D3 v3.x). Para usar `this` , sin embargo, uno no necesita usar ningún argumento:

```
.on("mouseover", function(){
    d3.select(this);
});
```

El código anterior seleccionará `this` cuando el mouse esté sobre el elemento. Compruébalo funcionando en este fiddle: https://jsfiddle.net/y5fwgopx/

La función de flecha

Como una nueva sintaxis de ES6, una función de flecha tiene una sintaxis más corta en comparación con la expresión de función. Sin embargo, para un programador D3 que utiliza `this` constante, hay una trampa: una función de la flecha no crea su propia `this` contexto. Eso significa que, en una función de flecha, `this` tiene su significado original del contexto adjunto.

Esto puede ser útil en varias circunstancias, pero es un problema para un codificador acostumbrado a usar `this` en D3. Por ejemplo, usando el mismo ejemplo en el violín anterior, esto no funcionará:

```
.on("mouseover", ()=>{
    d3.select(this);
});
```

Si tiene dudas, aquí está el violín: https://jsfiddle.net/tfxLsv9u/

Bueno, eso no es un gran problema: uno simplemente puede usar una expresión de función tradicional y antigua cuando sea necesario. ¿Pero qué pasa si quieres escribir todo tu código usando las funciones de flecha? ¿Es posible tener un código con funciones de dirección **y** seguir utilizando adecuadamente `this` en D3?

Los argumentos segundo y tercero combinados

La respuesta es **sí** , porque `this` es lo mismo de los `nodes[i]` . La sugerencia está realmente presente en toda la API D3, cuando describe esto:

> ... con `this` como el elemento DOM actual (`nodes[i]`)

La explicación es simple: dado que los `nodes` son el grupo actual de elementos en el DOM `i` es el índice de cada elemento, los `nodes[i]` refieren al elemento DOM actual. Es decir, `this` .

Por lo tanto, uno puede utilizar:

```
.on("mouseover", (d, i, nodes) => {
    d3.select(nodes[i]);
});
```

Y aquí está el violín correspondiente: https://jsfiddle.net/2p2ux38s/

Lea Trozos escogidos en línea: https://riptutorial.com/es/d3-js/topic/2135/trozos-escogidos

Capítulo 11: Usando D3 con JSON y CSV

Sintaxis

- d3.csv (url [[, fila], devolución de llamada])
- d3.tsv (url [[, fila], devolución de llamada])
- d3.html (url [, devolución de llamada])
- d3.json (url [, devolución de llamada])
- d3.text (url [, devolución de llamada])
- d3.xml (url [, devolución de llamada])

Examples

Cargando datos de archivos CSV

Hay varias formas de obtener los datos que enlazará con los elementos DOM. La más simple es tener tus datos en tu script como una matriz ...

```
var data = [ ... ];
```

Pero **D3.js** nos permite cargar datos desde un archivo externo. En este ejemplo, veremos cómo cargar y tratar adecuadamente los datos de un archivo CSV.

Los archivos CSV son *valores separados por comas* . En este tipo de archivo, cada línea es un registro de datos, cada registro consta de uno o más campos, separados por comas. Es importante saber que la función que estamos a punto de usar, `d3.csv` , usa la primera línea del CSV como **encabezado** , es decir, la línea que contiene los nombres de los campos.

Entonces, considera este CSV, llamado "data.csv":

```
city,population,area
New York,3400,210
Melbourne,1200,350
Tokyo,5200,125
Paris,800,70
```

Para cargar "data.csv", usamos la función `d3.csv` . Para hacerlo más fácil, suponga que "data.csv" está en el mismo directorio de nuestro script, y su ruta relativa es simplemente "data.csv". Entonces, escribimos:

```
d3.csv("data.csv", function(data){
    //code dealing with data here
});
```

Observe que, en la devolución de llamada, usamos los `data` como un argumento. Esa es una práctica común en D3, pero puedes usar cualquier otro nombre.

¿ `d3.csv` hace `d3.csv` con nuestro CSV? Convierte el CSV en una matriz de objetos. Si, por ejemplo, `console.log` nuestros datos:

```
d3.csv("data.csv", function(data){
    console.log(data)
});
```

Esto es lo que vamos a ver:

```
[
    {
        "city": "New York",
        "population": "3400",
        "area": "210"
    },{
        "city": "Melbourne",
        "population": "1200",
        "area": "350"
    },{
        "city": "Tokyo",
        "population": "5200",
        "area": "125"
    },{
        "city": "Paris",
        "population": "800",
        "area": "70"
    }
]
```

Ahora podemos enlazar esos datos a nuestros elementos DOM.

Observe que, en este ejemplo, la `population` y el `area` son cadenas. Pero, probablemente, quieras tratarlos como números. Puede cambiarlos en una función dentro de la devolución de llamada (como `forEach`), pero en `d3.csv` puede usar una función de "acceso":

```
d3.csv("data.csv", conversor, function(data){
    //code here
});

function conversor(d){
    d.population = +d.population;
    d.area = +d.area;
    return d;
}
```

También puede usar los `d3.tsv` en `d3.tsv` , pero no en `d3.json` .

Nota: `d3.csv` es una función asíncrona, lo que significa que el código después de que se ejecute inmediatamente, incluso antes de que se cargue el archivo CSV. Por lo tanto, una atención especial para el uso de sus `data` dentro de la devolución de llamada.

Uno o dos parámetros en la devolución de llamada: manejo de errores en d3.request ()

Cuando se usa `d3.request()` o uno de los constructores de conveniencia (d3.json , d3.csv , d3.tsv , d3.html y d3.xml) hay muchas fuentes de error. Puede haber problemas con la solicitud emitida o su respuesta debido a errores de red, o el análisis podría fallar si el contenido no está bien formado.

Dentro de las devoluciones de llamadas pasadas a cualquiera de los métodos mencionados anteriormente, es conveniente, por lo tanto, implementar algún manejo de errores. Para este propósito, las devoluciones de llamada pueden aceptar dos argumentos, el primero es el error, si hay alguno, el segundo son los datos. Si se produjo algún error durante la carga o el análisis, la información sobre el error se pasará como primer `error` argumento con los `data` `null` .

```
d3.json{"some_file.json", function(error, data) {
  if (error) throw error;   // Exceptions handling
  // Further processing if successful
});
```

Usted, en todo caso, no está obligado a proporcionar dos parámetros. Está perfectamente bien usar los métodos de solicitud con una devolución de llamada que presenta solo un parámetro. Para manejar este tipo de devoluciones de llamada, hay una función privada `fixCallback()` en request.js, que ajusta la forma en que se pasa la información al único argumento del método.

```
function fixCallback(callback) {
  return function(error, xhr) {
    callback(error == null ? xhr : null);
  };
}
```

Esto será invocado por D3 para todas las devoluciones de llamada que tengan un solo parámetro, que por definición son los datos.

No importa cuántos parámetros se proporcionen a la devolución de llamada del método de solicitud, la regla para el parámetro de `data` es:

- Si la solicitud falla, los `data` serán `null`
- Si la solicitud tiene éxito, los `data` contendrán los contenidos cargados (y analizados)

La única diferencia entre la versión de un parámetro y la versión de dos parámetros es la forma en que se proporciona la información sobre el error que puede ocurrir. Si se omite el parámetro de `error` , el método fallará silenciosamente dejando los `data` como `null` . Si, por otro lado, la devolución de llamada está definida para tener dos argumentos, la información sobre un error durante la carga o el análisis se pasará al primer parámetro que le permitirá manejarlo adecuadamente.

Las siguientes cuatro llamadas a `d3.json` demuestran los escenarios posibles para archivos existentes / no existentes frente a un parámetro / dos devoluciones de llamada de parámetros:

```
// FAIL: resource not available or content not parsable
// error contains information about the error
// data will be null because of the error
d3.json("non_existing_file.json", function(error, data) {
```

```
  console.log("Fail, 2 parameters: ", error, data);
});

// FAIL: resource not available or content not parsable
// no information about the error
// data will be null because of the error
d3.csv("non_existing_file.json", function(data) {
  console.log("Fail, 1 parameter: ", data);
});

// OK: resource loaded successfully
// error is null
// data contains the JSON loaded from the resource
d3.json("existing_file.json", function(error, data) {
  console.log("OK, 2 parameters: ", error, data);
});

// OK: resource loaded successfully
// no information about the error; this fails silently on error
// data contains the JSON loaded from the resource
d3.json("existing_file.json", function(data) {
  console.log("OK, 1 parameter: ", data);
});
```

Lea Usando D3 con JSON y CSV en línea: https://riptutorial.com/es/d3-js/topic/5201/usando-d3-con-json-y-csv

Capítulo 12: Usando D3 con otros frameworks

Examples

Componente D3.js con ReactJS

Este ejemplo se basa en una publicación de blog de Nicolas Hery . Utiliza clases de ES6 y los métodos de ciclo de vida de ReactJS para mantener actualizado el componente D3

---

d3_react.html

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <title>Hello, d3React!</title>
  <style>
    .d3Component {
      width: 720px;
      height: 120px;
    }
  </style>
</head>
<script src="https://fb.me/react-15.2.1.min.js"></script>
<script src="https://fb.me/react-dom-15.2.1.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
<script src="https://d3js.org/d3.v4.min.js"></script>

<body>
  <div id="app" />
  <script type="text/babel" src="d3_react.js"></script>
</body>

</html>
```

d3_react.js

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      d3React: new d3React()
    };
    this.getd3ReactState = this.getd3ReactState.bind(this);
  }
```

```
  getd3ReactState() {
    // Using props and state, calculate the d3React state
    return ({
      data: {
        x: 0,
        y: 0,
        width: 42,
        height: 17,
        fill: 'red'
      }
    });
  }

  componentDidMount() {
    var props = {
      width: this._d3Div.clientWidth,
      height: this._d3Div.clientHeight
    };
    var state = this.getd3ReactState();
    this.state.d3React.create(this._d3Div, props, state);
  }

  componentDidUpdate(prevProps, prevState) {
    var state = this.getd3ReactState();
    this.state.d3React.update(this._d3Div, state);
  }

  componentWillUnmount() {
    this.state.d3React.destroy(this._d3Div);
  }

  render() {
    return (
      <div>
        <h1>{this.props.message}</h1>
        <div className="d3Component" ref={(component) => { this._d3Div = component; } } />
      </div>
    );
  }
}

class d3React {
  constructor() {
    this.create = this.create.bind(this);
    this.update = this.update.bind(this);
    this.destroy  = this.destroy.bind(this);
    this._drawComponent = this._drawComponent.bind(this);
  }

  create(element, props, state) {
    console.log('d3React create');
    var svg = d3.select(element).append('svg')
      .attr('width', props.width)
      .attr('height', props.height);

    this.update(element, state);
  }

  update(element, state) {
    console.log('d3React update');
    this._drawComponent(element, state.data);
```

```
  }

  destroy(element) {
    console.log('d3React destroy');
  }

  _drawComponent(element, data) {
    // perform all drawing on the element here
    var svg = d3.select(element).select('svg');

    svg.append('rect')
      .attr('x', data.x)
      .attr('y', data.y)
      .attr('width', data.width)
      .attr('height', data.height)
      .attr('fill', data.fill);
  }
}

ReactDOM.render(<App message="Hello, D3.js and React!"/>, document.getElementById('app'));
```

Coloque el contenido de `d3_react.html` y `d3_react.js` en el mismo directorio y navegue por un navegador web hasta el archivo d3React.html. Si todo va bien, verás un encabezado que dice `Hello, D3.js and React!` renderizado desde el componente React y un rectángulo rojo debajo del componente D3 personalizado.

React usa refs para "llegar" a la instancia del componente. Los métodos de ciclo de vida de la clase `d3React` requieren que esta referencia agregue, modifique y elimine elementos DOM. La clase `d3React` se puede ampliar para crear más componentes personalizados e insertarse en cualquier lugar `div.d3Component` React `div.d3Component` un `div.d3Component` .

D3js con Angular

El uso de D3js con Angular puede abrir nuevos frentes de posibilidades, como la actualización en vivo de gráficos, tan pronto como se actualicen los datos. Podemos encapsular la funcionalidad completa del gráfico dentro de una directiva angular, lo que lo hace fácilmente reutilizable.

index.html >>

```
<!DOCTYPE html>
<html ng-app="myApp">
<head>
  <script src="https://d3js.org/d3.v4.min.js"></script>
  <script data-require="angular.js@1.4.1" data-semver="1.4.1"
src="https://code.angularjs.org/1.4.1/angular.js"></script>
  <script src="app.js"></script>
  <script src="bar-chart.js"></script>
</head>

<body>

  <div ng-controller="MyCtrl">
    <!-- reusable d3js bar-chart directive, data is sent using isolated scope -->
    <bar-chart data="data"></bar-chart>
  </div>
```

```
</body>
</html>
```

Podemos pasar los datos a la tabla utilizando el controlador, y observar cualquier cambio en los datos para permitir la actualización en vivo de la tabla en la directiva:

app.js >>

```
angular.module('myApp', [])
  .controller('MyCtrl', function($scope) {
    $scope.data = [50, 40, 30];
    $scope.$watch('data', function(newVal, oldVal) {
      $scope.data = newVal;
    }, true);
  });
```

Finalmente, la definición de la directiva. El código que escribimos para crear y manipular el gráfico se ubicará en la función de enlace de la directiva.

Tenga en cuenta que también hemos puesto un alcance. $ Watch en la directiva, para actualizar tan pronto como el controlador pase nuevos datos. Estamos reasignando nuevos datos a nuestra variable de datos si hay algún cambio en los datos y luego llamamos a la función repaintChart (), que realiza la representación del gráfico.

bar-chart.js >>

```
angular.module('myApp').directive('barChart', function($window) {
  return {
    restrict: 'E',
    replace: true,
    scope: {
      data: '='
    },
    template: '<div id="bar-chart"></div>',
    link: function(scope, element, attrs, fn) {

      var data = scope.data;
      var d3 = $window.d3;
      var rawSvg = element;

      var colors = d3.scale.category10();

      var canvas = d3.select(rawSvg[0])
        .append('svg')
        .attr("width", 300)
        .attr("height", 150);

      // watching for any changes in the data
      // if new data is detected, the chart repaint code is run
      scope.$watch('data', function(newVal, oldVal) {
        data = newVal;
        repaintChart();
      }, true);

      var xscale = d3.scale.linear()
```

```
        .domain([0, 100])
        .range([0, 240]);

      var yscale = d3.scale.linear()
        .domain([0, data.length])
        .range([0, 120]);

      var bar = canvas.append('g')
        .attr("id", "bar-group")
        .attr("transform", "translate(10,20)")
        .selectAll('rect')
        .data(data)
        .enter()
        .append('rect')
        .attr("class", "bar")
        .attr("height", 15)
        .attr("x", 0)
        .attr("y", function(d, i) {
          return yscale(i);
        })
        .style("fill", function(d, i) {
          return colors(i);
        })
        .attr("width", function(d) {
          return xscale(d);
        });

      // changing the bar widths according to the changes in data
      function repaintChart() {
        canvas.selectAll('rect')
          .data(data)
          .transition()
          .duration(800)
          .attr("width", function(d) {
            return xscale(d);
          })
      }
    }
  }
});
```

Aquí está el trabajo de JSFiddle.

Gráfico de D3.js con Angular v1

HTML:

```
<div ng-app="myApp" ng-controller="Controller">
    <some-chart data="data"></some-chart>
</div>
```

Javascript:

```
angular.module('myApp', [])
.directive('someChart', function() {
    return {
        restrict: 'E',
```

```
        scope: {data: '=data'},
        link: function (scope, element, attrs) {
        var chartElement = d3.select(element[0]);
            // here you have scope.data and chartElement
            // so you may do what you want
        }
    };
});

function Controller($scope) {
    $scope.data = [1,2,3,4,5]; // useful data
}
```

Lea Usando D3 con otros frameworks en línea: https://riptutorial.com/es/d3-js/topic/3733/usando-d3-con-otros-frameworks

Creditos

S. No	Capítulos	Contributors
1	Empezando con d3.js	4444, Adam, altocumulus, Arthur Tarasov, Community, davinov, Fawzan, Gerardo Furtado, Ian H, jmdarling, kashesandr, Marek Skiba, Maximilian Kohl, Rachel Gallen, Ram Visagan, RamenChef, Ruben Helsloot, TheMcMurder, Tim B, winseybash
2	Conceptos básicos de SVG utilizados en la visualización de D3.js	fengshuo, Gerardo Furtado
3	Despachando eventos con d3.dispatch	aug, kashesandr, Ram Visagan
4	En eventos	aug, Ian H, Kemi, Tim B
5	Enfoques para crear gráficos d3.js responsivos	Ian H
6	Gráficos SVG usando D3 js	rajarshig
7	Haciendo robusto, responsivo y reutilizable (r3) para d3	SumNeuron
8	patrón de actualización	Gerardo Furtado
9	Proyecciones D3	Andrew Reid
10	Trozos escogidos	Ashitaka, aug, Carl Manaster, Gerardo Furtado, Ian, Ian H, JulCh, Tim B
11	Usando D3 con JSON y CSV	altocumulus, Gerardo Furtado
12	Usando D3 con otros frameworks	Adam, kashesandr, Rishabh