Tuto3: Developing UI with React and D3.js

Nicolas Médoc Luxembourg Inst. of Science and Technology

Outline

1. Introduction to D3.js
2. Selection and data binding
3. Using scales
4. Interactions

What is D3.js?

`https://d3js.org/`

- **An open source library for data visualization** built on Web standards (HTML, CSS, SVG, Canvas, javascript).
- A **low-level toolbox** with modules to prepare and transform the data, to draw data-driven graphics, and to interact with them.
- **Dynamic, interactive and animated visualizations:** thanks to the data binding and join mechanisms, D3.js facilitates interactivity with animated transitions in response to user inputs or external events.
- **Not a high-level chart library:** but a collection of low-level primitives to tailor the visualizations to fit specific requirements (for quick and basic visualizations, libraries with built-in charts are prefect: Observable Plot in javascript or MathPlotLib in python).

First example: a matrix visualization in d3.js

- https://github.com/nicolasmedoc/TD3-D3js
- declarative approach by calling a chain of methods
- to append DOM elements: .append("g")
- to update their attributes: .attr("class","matSvgG")
- to remove DOM elements: .remove()

```
matSvg=d3.create("svg")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
;
matSvgG = matSvg.append("g")
    .attr("class","matSvgG")
    .attr("transform","translate("+margin.left+","+margin.top+")");
;
```

Selection and data binding

- Call **selection.selectAll(selector)** to search in the DOM structure all the child elements corresponding to the criteria given in selector (a class, an id, a DOM element)
- Call **selection.select(selector)** return the first element in the DOM structure of the selection children
- Call **selection.data(array, idGetter)** to bind an array of data items to the selection. It returns a selection based on the array items matching the previous data binding (comparison with the attribute returned by the getter function, or by default with object references).
- Call **enter()** chained after data() to select each new data item that didn't exist in the previous data binding (they are not yet associated to elements in the DOM). At the first call, it returns a selection of items with placeholder elements (they don't exist yet), to then append new elements in the DOM.
- Call **exit()** after data() or enter() to select old DOM elements related to the data items that don't exist anymore in the new array, and then to remove them.

See illustrations at `https://bost.ocks.org/mike/selection/#enter-update-exit`.

Example of selection, data binding and enter() to add new elements

In the function renderMatrix(), create <g> elements and store the selection in cellG

```
const cellG = matSvgG.selectAll(".cellG")
    // all elements with the class .cellG (empty the first time)
    .data(genData,(cellData)=>cellData.index)
    .enter()
    // all data items to add:
    // didn't exist in the previous data binding but exist now in the new array
    .append("g")
    .attr("class","cellG")
    .attr("transform",(cellData)=>{
        return "translate("+(cellData.colPos*cellSize)+","+(cellData.rowPos*cellSize)+")"
    })
;
```

Check the DOM structure with 16 <g> elements

Append child elements (rect and circle) to each <g>

```
// render rect as child of each element "g"
cellG.append("rect")
    .attr("class","CellRect")
    .attr("width",cellSize-1)
    .attr("height",cellSize-1)
    .attr("fill","lightGray")
;
// render circle as child of each element "g"
cellG.append("circle")
    .attr("class","CellCircle")
    .attr("cx",radius)
    .attr("cy",radius)
    .attr("r",radius)
    .attr("fill","steelblue")
;
```

The scales

- the scales allows to apply scale transformations from the data space (the domain values) to a range of values corresponding to any chosen visual encoding
- different types of scale are available for different types of transformation: linear, pow, log, ordinal, quantile, ..
- `https://d3js.org/d3-scale`

Size and color of cells

```
// build the size scale
const minNbProductSold = d3.min(genData.map(cellData=>cellData.nbProductSold));
const maxNbProductSold = d3.max(genData.map(cellData=>cellData.nbProductSold));
const cellSizeScale = d3.scaleLinear()
    .domain([minNbProductSold, maxNbProductSold])
    .range([2, radius-1])
;
// build the color scale
const colorScheme = d3.schemeYlGnBu[9];
const cellColorScale = d3.scaleQuantile()
    .domain(genData.map(cellData=>cellData.salesGrowth))
    .range(colorScheme)
;
```

Example: using scales for encoding the size and color of circles

And we use the scales to compute the circle attributes by using a function with the data item in parameter:

```
...
    .attr("r",(cellData)=>cellSizeScale(cellData.nbProductSold))
    .attr("fill",(cellData) =>{
        const color =  cellColorScale(cellData.salesGrowth);
        return color;
    })
...
```

Adding interactions

In D3js you can declare events with .on() function:

```
.on("click",(event, cellData)=>{// do something with event and/or cellData})
.on("mouseenter",(event, cellData)=>{// do something with event/cellData})
.on("mouseleave",(event, cellData)=>{// do something with event/cellData})
```

Example: highlighting the cell border on user click

```
function renderMatrix(genData)
...
      .append("g")
      .attr("class","cellG")
      .attr("transform",(cellData)=>{
          return "translate("+(cellData.colPos*cellSize)+ ...
      })
      .on("click", (event,cellData)=>{
          handleOnClickCell(cellData);
      })
    ;
...
    cellG.append("circle")
      .attr("class","CellCircle")
      .attr("stroke", "red")
      .attr("stroke-width", (cellData)=>cellData.selected?2:0)
```

Example: highlighting the cell border on user click

Since genData is updated and no new item is added, enter() selection is empty and nothing happens when calling renderMatrix() on click envent. A quick-and-dirty solution consists in removing all elements and rebuild with a new data binding to select the updated genData in enter() function, but it is not optimal.

```
...
function removeMatrix(){
    matSvgG.selectAll('*').remove();
}
function renderMatrix(genData) {
    removeMatrix();
...
```

Using the update pattern is preferable.

General update pattern with join()

The general update pattern of D3js allows declaring different behaviors after binding new/updated data to a selection. The **join()** function called just after the data binding takes in parameter 3 functions to declare these behaviors:

- **enter function** to define what to do with new items;
- **update function** to handle the items matching with the previous data binding;
- **exit function** for old items that does not exist anymore.

See illustrations at `https://bost.ocks.org/mike/selection/#enter-update-exit` and `https://observablehq.com/@d3/selection-join`.

Exercise 1: implement the update pattern

In renderMatrix avoid calling removeMatrix() and call join() function following the example below:

```
const cellG = this.matSvg.selectAll(".cellG")
    .data(matrixData.genData,(cellData)=>cellData.index)
    .join(
        enter =>{// appends elements with fixed atributes
            // append cellG
            // append CellRect with the color
            // append CellCircle at center position
        },
        update =>{ // select elements and declare changing attributes
            // the cell position (<g> translation)
            // the circle size
            // the circle color
        },
        exit =>{ // declare what to do with items that don't exist anymore
            exit.remove();
        }
    )
;
```

Exercise 2: Optimizing updates

In certain cases we want to re-render only a few number of items, e.g. highlighting clicked or hovered cell(s). In that case we don't need to re-render all the cells. So we create a specific function to declare this behavior with update pattern:

```
function handleOnClickCell(cellData){
    const cellsToUpdate=[{...cellData,selected:!cellData.selected}]
    updateCellHighlighting(cellsToUpdate)
}
...
function updateCellHighlighting(cellsToUpdate){
    matSvgG.selectAll(".cellG")
        .data(cellsToUpdate, cellData=>cellData.index)
        // no need to call join() because we don't need enter or exit
        // update selection is already returned by data()
        .select(".CellCircle")
        .attr("stroke-width",cellData=>cellData.selected?2:0);
    ;
}
```

Exercise 3: using animated transitions

Before removing elements or updating attributes we can declare an animated transition with a specific duration to smoothly observe the transitions between updated positions or colors:

```
cellG.transition()
    .duration(transitionDuration)
    .attr("transform",(cellData)=>{
        return "translate("+(cellData.colPos*this.cellSize)...
    })
;
```

Add transitions before update and exit of the join

Exercise 4 (optional)

Use mouseenter and mouseleave events to implement the mouse hover interaction on matrix cells

```
.on("mouseenter",(event, cellData)=>{// do something with event/cellData})
.on("mouseleave",(event, cellData)=>{// do something with event/cellData})
```

Exercise 5 (optional)

In renderMatrix(), add labels in top and left margin.

Exercise 6 (optional)

- When the user clicks on country labels, sort the products by their decreasing nbProducSold
- When the user clicks on company labels, sort the countries by their decreasing nbProducSold