



## TUTO5: DEVELOPING DATA VISUALIZATIONS FOR MULTI-DIMENSIONAL DATA

NICOLAS MÉDOC    LUXEMBOURG INST. OF SCIENCE AND TECHNOLOGY



# Outline

1. Recall: React + D3Js
  - 1.1 Separation of concerns
  - 1.2 Ready-to-use templates
  - 1.3 React useEffect() hook
  - 1.4 React useRef() hook
2. Visualization for multi-dimensional data: scatterplot
  - 2.1 Building a scatterplot
  - 2.2 Synchronization of multiple scatterplots

# Combining D3.js and React: Separation of concerns



- **D3.js visualizations** are implemented in **self-contained class**, without any dependencies to React library
- For each visualization, **one React component** implements the **container of the visualization** and **makes the connection** between the D3js class and the React application.
- The React component:
  - **receives the data** from the Redux store;
  - **instantiates the D3.js class** and call render/update methods to build/update the visualization,
  - **provides the event handler methods to D3.js** class to update the store.

# Combining D3.js and React: ready-to-use templates



- <https://github.com/nicolasmedoc/Tuto5-MultiDim-Redux.git>
- Vis-d3.js
  - is a **javascript class** which will renders the visualization. It is self-contained and independent of React
  - declares a method **create()** to initialize the SVG element
  - declares a method **clear()** to remove the SVG from the DOM
  - declares **one or several update methods** (e.g. `renderVis()`, `updateFunction1()`, `updateFunction2()`) to change the view or a part of the visualization when the data changes, with the **global update pattern**.



## Combining D3.js and React: first example

- VisContainer.js (React component)
  - is the container of the visualization. It uses the React state (useState() hook) or the Redux store (useSelector() hook) and dispatch the actions of reducers to handle the events.
  - controls the component lifecycle with **useEffect() hook** when the component **did mount, did update** (when data changes) or **did unmount** (when removed from the page)
  - **instantiates the d3 class** and store it in a **Ref** (with useRef() hook) to keep the d3 instance when the component re-render.
  - **renders the <div>** element containing the SVG and **stores it in a Ref** (with useRef() hook) to avoid re-render it.

# React useEffect() hook to control the component life cycle



3 functions are declared in 2 different profiles of the useEffect() hook to declare additional behavior in the React component life cycle:

```
import { useEffect } from 'react';  
...  
useEffect(()=>{  
  // the behavior after the component creation (did mount)  
  return ()=>{  
    // the behavior after the component deletion (did unmount)  
    // is declared in the return function  
  }  
}, []) // empty array  
useEffect(()=>{  
  // behavior after update of dependency1 or dependency2 only  
},[dependency1,dependency2]) // array of dependency variables  
useEffect(()=>{  
  // behavior after update of dependency3 only  
},[dependency3]) // array of dependency variables
```

**dependency1, dependency2 and dependency3** are variables derived from the state or propagated in props by the parent

# React useRef() hook to persist a value in the component



Used to persist the instance object of the D3 class in a React component:

```
import { useEffect, useRef } from 'react';  
...  
const divContainerRef=useRef(null);  
const visD3Ref = useRef(null)  
useEffect(()=>{  
    const visD3 = new VisD3(divContainerRef.current);  
    visD3Ref.current = visD3;  
},[])
```

# React useRef() hook to persist a value in the component



Used to persist the previous value in the sub part of a dataSlice propagated in props:

```
import { useEffect, useRef } from 'react';  
...  
const dataSliceAttributeRef = useRef(null)  
useEffect(()=>{  
  // behavior after update of dataSlice  
  if(dataSliceAttributeRef.current!==dataSlice.attribute){  
    // attribute has been updated => do something  
    // e.g. call specific update method in D3 class  
    dataSliceAttributeRef.current = dataSlice.attribute  
  }  
},[dataSlice]) // array of dependency variables
```



# Building a scatterplot: getting the data set



See in `redux/DataSetSlice.js` the use of `createAsyncThunk()` function to retrieve data from `data/Housing.csv` with `async` fetch function.

`extraReducers` in second parameter of `createSlice()` allows to declare behaviours when the `async` action is pending, fulfilled or rejected

```
export const dataSetSlice = createSlice({
  ...
  extraReducers: builder => {
    builder.addCase(getDataSet.pending, (state, action) => {
      console.log("extraReducer getDataSet.pending");
      // do something with state, e.g. to change a status
    })
    builder.addCase(getDataSet.fulfilled, (state, action) => {
      return action.payload
    })
    builder.addCase(getDataSet.rejected, (state, action) => {
      // Add any fetched house to the array
      const error = action.payload
      console.log("extraReducer getDataSet.rejected with error" + error);
    })
  }
})
```

# Building a scatterplot: getting the data set



**Exercise1:** In App.js, create a `useEffect()` function to load the dataset when the App component did mount.

# Building a scatterplot: render the visualization component



**Exercise2:** In ScatterplotContainer.js call the scatterplotD3.renderScatterplot() method in the useEffect hook reacting to visData updates:

```
// get the current instance of scatterplotD3 from the Ref object...  
// call renderScatterplot method ...  
// controllerMethods being already defined in the useEffect() method.  
// with empty handleClick, handleOnMouseEnter and handleOnMouseLeave
```

# Building a scatterplot: render the visualization component



Then, add the component in the rendered JSX of App.js (x="population" y="ViolentCrimesPerPop")

```
return (  
  <div className="App">  
    <div id={"MultiviewContainer"} className={"row"}>  
      // call the scatterplot component here  
    </div>  
  </div>  
>);
```

# Building a scatterplot: creation of scales and X/Y axis



In components/scatterplot/Scatterplot-d3.js, see in create() how the xScale and yScale are initialized, and the creation of groups 'g' for x and y axes.

**Exercice3:** in updateAxis(), using d3.min(mylist) and d3.max(mylist), set the domain values of this.xScale.domain(...) and this.yScale.domain(...) to put **"area" in X Axis and "price" in Y axis.**

In "updateAxis" function, use these scales to build X axis and Y axis:

```
// .xAxisG and .yAxisG are initialized in create() function
this.svg.select(".xAxisG")
  .transition().duration(500)
  .call(d3.axisBottom(this.xScale))
;
this.svg.select(".yAxisG")
  .transition().duration(500)
  .call(d3.axisLeft(this.yScale))
;
```

# Building a scatterplot: update circle positions with scales



in components/scatterplot/Scatterplot-d3.js, in the method `updateMarkers()`:

**Exercise4:** using X/Y scales, apply a translation to `.markerG` to update the circle positions. `updateMarkers(selection)` is called from `renderScatterplot()`. The "selection" parameter comes from the update pattern using `join()`. It corresponds to a selection of `".markerG"`.

```
selection
  .transition().duration(500)
  .attr("transform", (itemData)=>{
    // use scales to return shape position from data values
  })
```

# Building a scatterplot: reuse the scatterplot component



**Exercise 5:** display two scatterplot side by side to show different pair of dimensions:

- scatterplot 1: (x="population" y="ViolentCrimesPerPop")
- scatterplot 2: (x="medIncome" y="ViolentCrimesPerPop")



# Coordinated multiple views: principles

- **Exercise 6:** synchronize the two scatterplots on click interaction. The purpose is to highlight the clicked objects simultaneously in the two scatterplots (e.g. make the red border visible).
- **In `ScatterplotContainer.js` use `useEffect()`:** the object `controllerMethods` is propagated to the `ScatterplotD3` class and contains all the methods defining the behaviour of interactions (`handleOnClick`, `handleOnMouseEnter`, `handleOnMouseLeave`)
- These methods have to **update a data slice in the redux store** to synchronize the two scatterplots by calling a dispatch with the corresponding reducer actions.



## Coordinated multiple views: prepare the actions int the reducer



Create a new redux slice (`ItemInteractionSlice.js`) that will contain an object with an array `selectedItems` and an object `hoveredItem`  
`selectedItems: [], hoveredItem: .` Create the reducer actions `setSelectedItems` and `setHoveredItem`. Don't forget to declare the reducer in the store.

## Coordinated multiple views: use dispatch to call the action functions



In ScatterplotContainer.js call the reducer action with dispatch in handleClick, which is propagated to the D3 class.

# Coordinated multiple views: highlight an object in the views



Propagate the synchronization data in ScatterplotContainer-js:

- with `useSelector()`, retrieve `selectedItems`
- create a specific `useEffect` with `selectedItems` as dependency to call the function in `D3` class that will highlight the selected marker:  
`highlightSelectedItems()`

# Coordinated multiple views: highlight an object in the views



In Scatterplot-d3.js `highlightSelectedItems` method, use the update pattern to declare the right behaviour:

```
highlightSelectedItems(selectedItems){  
  // use pattern update to change the border and opacity of objects:  
  //   - call this.changeBorderAndOpacity(selection,true)  
  //   for markers that match selectedItems  
  //   - this.changeBorderAndOpacity(selection,false) for  
  //   markers that do not match selectedItems  
}
```

# Coordinated multiple views: highlight an object in the views



By looking at the logs when clicking an item, the scatterplot renders again while it is not necessary. This is due to the various dependencies declared in the `useEffect`, including the object `"scatterplotControllerMethods"` having a new instance at each React life cycle. **To render only when the `scatterplotData` changes, we use Ref object** to keep the previous state of `scatterplotData` and compare with the new state:

```
const scatterplotDataRef = useRef(scatterplotData);
// did update, called each time dependencies change, dispatch remain stable over component cycles
useEffect(()=>{
  ...
  if(scatterplotDataRef.current !== scatterplotData) {
    console.log("ScatterplotContainer useEffect when scatterplotData changes...");
    // call renderScatterplot of ScatterplotD3...;
    scatterplotDataRef.current = scatterplotData;
  }
  ...
}, [scatterplotData]);
```