Visualization TD1 Developing UI with react

Installation and configuration setup (10 min)

Getting the TD1 tutorial

Download: https://github.com/nicolasmedoc/TD1-react1/archive/refs/heads/main.zip

Development environment

Install Node.js: https://nodejs.org/en/download/package-manager

- Choose your preferred installation mode for your OS
- For windows without admin rights:
  - https://nodejs.org/dist/v20.17.0/node-v20.17.0-win-x64.zip
  - Declare the extracted folder in PATH environment variable

Test the installed version in a terminal:

```
node -v # should print `v20.17.0`
npm -v # should print `v10.8.2`
```

IDE:

- VS Code : https://code.visualstudio.com/
- Or your preferred javascript IDE

Install project dependencies with npm, the package manager system for javascript embedded in Node.js. All dependencies are declared at the root of the project in the file package.json. After having installed Node.js open a terminal and call:

```
npm install # from your project folder
```

Understanding the project structure

Open the project in VS code or your preferred IDE:

- File>Open Folder and choose TD1-react1-main

public/index.html (do not change): the web page with a <div> element in which React will inject html.

```
...
<body>
  <div id="root"></div>
</body>
...
```

src/index.js (do not change): render the main React javascript code that produces html in the root <div>

```
// import dependencies
...
const root = createRoot(document.getElementById("root"));
root.render(
  <StrictMode>
    <App />
  </StrictMode>
);
...
```

App.js + App.css: the main component from which you will develop your application.

For a better readability of your code create one css per component.

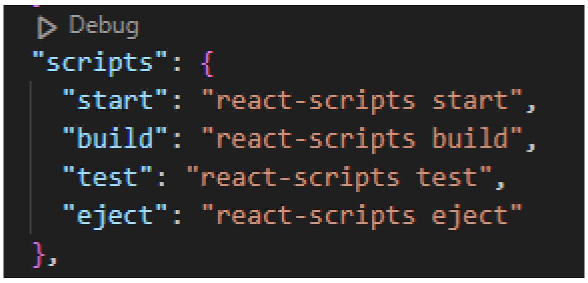
```
import './App.css';
// here import other dependencies

// a component is a piece of code which render a part of the user interface
function App() {
   return (
      // JSX code:
      // Combination of JavaScript code and HTML tags that describe what is displayed
   )
}
```

Run the React application:

Open a terminal and call:

```
npm start
```

or in VS code, open package.json and click on script: Debug button on top of "scripts":



```
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
```

In your browser, a web page will open on http://localhost:3000/

You will see "Hello world!"

Some links to understand html, css and javascript:

- https://www.w3schools.com/html/html_intro.asp
- https://www.w3schools.com/css/css_intro.asp
- https://www.w3schools.com/js/js_syntax.asp
- https://www.w3schools.com/js/js_datatypes.asp
- https://www.w3schools.com/js/js_function_closures.asp

Links with the doc of React

- https://react.dev/learn

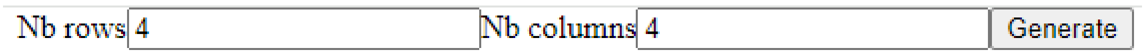
Visualize a randomly generated data in a matrix

The purpose of this tutorial will be to build a matrix visualization with generated data. You will progressively learn different notions used by react to build a web application.

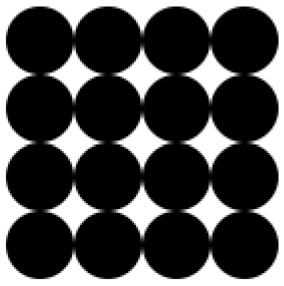
Create the page layout

We will create 2 containers:

- one with a control bar to configure the generation of the data, i.e. the number of rows and columns we want:



- one with the matrix visualization in SVG:



In App.js, we will replace "Hello world!" by two containers with <div> element:

```
function App() {
  return (
    <div className="App">
        <div id="control-bar-container">
            Control Bar
        </div>
        <div id="view-container">
            View
        </div>
    </div>
  );
}
```

In App.css, define the height for the two containers. We divide the height in two area, 10% for the control bar and 80% for the visualization.

```
#control-bar-container{
    height: 10%;
}
#view-container{
    height: 80%;
}
```

At this stage, you will not have the right division of space because the parent <div> of the control bar and the view containers does not declare a height. The height percentage of the two children cannot be calculated, so you need to declare a height (100%) in the class App in App.css:

```
.App {
    text-align: center;
    height:100%
}
```

Now in your browser you should see the two areas with the corresponding texts: "control bar" and "view" with the correct partition of the height.

Click right on the "Control bar" text and choose inspect in the contextualized menu to inspect the generated html elements.

Create a component that draws a SVG with circles in a grid.

Create the folder src/components which will contain all your components.

Create the folder src/components/matrix for the matrix component

Create the two files Matrix.js and Matrix.css

In Matrix.js create the component template and return an empty <svg>

```
import './Matrix.css'

function Matrix(){
    return(
        <svg width="100%" height="100%" >

        </svg>
    )
}

export default Matrix;
```

Call the component in App.js:

```
...
import Matrix from './components/matrix/Matrix'
...
<div id="view-container">
    <Matrix/>
</div>
...
```

In the browser, inspect the html to see the generated <svg> element.

In the SVG, we will add 4 rows of 4 circles of 34 px of diameter.

We first declare a function renderMatrix() in Matrix.js:

```
...
function Matrix(){
    const renderMatrix = function(){
        const nbRows= 4;
        const nbColumns = 4;
        const cellSize= 34;
        const radius = cellSize / 2;
        const cells=[];
        for (let rowPos=0; rowPos<nbRows; rowPos++){
            for(let colPos = 0; colPos<nbColumns; colPos++){
                cells.push(<circle r={radius} cx={colPos*cellSize + radius}
cy={rowPos*cellSize + radius}/>)
            }
        }
        return cells;
    }

    return(
        <svg width="100%" height="100%" >
            {renderMatrix()}
        </svg>
    )
}
...
```

Note: in the html code of the return(...) we can insert javascript code between {} to generate html code. Here we call a function which returns a list of circles. Based on this list, React will generate multiple <circle> elements in the resulting html (see in your browser with inspect).

It is preferable to wrap the shapes in <g> (groups in SVG) so that you can group multiple shapes at a same position and apply geometrical or colour transformations to all members of the group.

To do so, we will create in Matrix.js a new component (Cell) to draw the <circle> wrapped in a <g> element. We will introduce here the notion of component properties that declare any data to be passed from the parent component to the child components. Two properties will be declared in the Cell component: rowPos and colPos.

```
function Cell({rowPos,colPos}){
    const cellSize= 34;
    const radius = cellSize / 2;
    const transformStr="translate("+(colPos*cellSize + radius)+", "+(rowPos*cellSize +
radius)+")"
    return(
        <g transform={transformStr}>
            <circle r={radius}/>
        </g>
    )
}
```

As you can see in the code, instead of changing the position of the circles, we apply a translation on the <g> element. So the relative positions of circles to <g> is 0.

And the renderMatrix is simplified by calling the new Cell component:

```
 const renderMatrix = function(){
    const nbRows= 4;
    const nbColumns = 4;
    const cells=[];
    for (let rowPos=0; rowPos<nbRows; rowPos++){
        for(let colPos = 0; colPos<nbColumns; colPos++){
            cells.push(<Cell key={(rowPos*colPos)} rowPos={rowPos} colPos={colPos}/>);
        }
    }
    return cells;
}
```

Note: we added a 'key' property in the Cell component. This property must be unique in the list and is used by React to differentiate the cell components of the list. This allows to optimize the rendering by evaluating which components have been removed, added or updated.

Now we have a better separation of responsibilities between the Matrix component which builds the cell components with the rows and columns positions, and the Cell component which computes the individual coordinates x and y in the 2D space.

Use generated data and bind it to the React components

In src/utils/helper.js, you will find a function named genGridData, that generates a list of objects that will be visually represented in the matrix cells. The generated objects have the following attributes:

```
{index:integer, rowPos:integer, colPos:integer, nbProductSold:integer,
salesGrowth:float, rowLabel:string, colLabel:string}
```

The index is the position of the object in the list, the rowPos and colPos are respectively the positions of the cell in rows and columns. The rows represent the companies selling their products in different countries in columns. The value nbProductSold is the number of products sold by each company in each country and the salesGrowth is the sales growth of the previous year of each company in each country.

We will first generate a 4x4 matrix dataset in App.js component. We import the function and call it in the App component function:

```
import {genGridData} from "./utils/helper";
...
function App() {
    const nbRows=4, nbCols=4;
    const initGenData = genGridData(nbRows,nbCols);
    return (
    ...
    )
```

React provides a Hook named useState that maintains the state of a piece of data and notifies all dependent components to be refreshed when updated.

We first import the React hook function useState and then we store the dataset in a state object:

```
import {useState} from 'react'
...
function App() {
    ...
    const initGenData = genGridData(nbRows,nbCols);
    const [genData,setGenData] = useState(initGenData)
    ...
}
```

useState returns two objects: one containing the stored data (genData), and one containing a setter method (setGenData) to update the data in the store. When the setter method is called, the React life cycle notifies automatically all the components using genData (declared in their properties) to be informed the data has changed and they need to render themself again.

In our example we pass the data through a property of the Matrix component ({matrixData} in parameter) and use it to render the cells in renderMatrix:

```
...
function Matrix({matrixData}){
    const renderMatrix = function(){
        return matrixData.map(cellData=>{
            return <Cell key={cellData.index} rowPos={cellData.rowPos} 
colPos={cellData.colPos}/>
        })
    }
...
```

In the return of App function in App.js, add the property matrixData={genData} to declare the stored genData state as a property of the Matrix component:

```
...
<div id="view-container">
    <Matrix matrixData={genData}/>
</div>
...
```

Control the data generation (nbRows x nbColumns) in the control panel

Exercice1: create a component ControlBar which render the following form:

```
<form>
    <label>
        Nb rows
        <input name="nbRows" defaultValue = "4"/>
    </label>

    <label>
        Nb columns
        <input name="nbCols" defaultValue = "4"/>
    </label>
    <button type="submit">Generate</button>
</form>
```

- create a new folder in src/components/ControlBar
- Create ControlBar.js with the component function
- Create ControlBar.css
- Call it in the App Component in the <div id="control-bar-container">

Sharing data from child to parent component:

The App component is responsible to generate the data and send it to the Matrix component (a child component) through a property. We want now to communicate the configuration data from the ControlBar component to its parent (App.js) to generate a new matrix data. To do so, we transmit a function in ControlBar property which will take the new configuration data as parameter and generate a new matrix data. Saving the state of the configuration data in the parent component is a good approach to display it or to send this data to other child components of the application. Let's do it in exercice2.

Exercice2:

- Add a new state in App to store {nbRows:4, nbCols:4} in an object named genConfig with a setter method named setGenConfig.
- Add a property to ControlBar to send genConfig data to it.

In the ControlBar you can use the genConfig object as default value instead of a number:

```
<form>
    <label>
        Nb rows
        <input name="nbRows" defaultValue ={genConfig.nbRows}/>
    </label>

    <label>
        Nb columns
        <input name="nbCols" defaultValue ={genConfig.nbCols}/>
    </label>
    <button type="submit">Generate</button>
</form>
```

Declare a function in ControlBar to handle onSubmit event when you click on the form button. We will now only display a message "Data generation with nbRows=... nbCols=...":

```
const handleOnSubmit = function(event){
    // Prevent the browser from reloading the page
    event.preventDefault();

    // get the form data and transform it in JSON format
    const form = event.target;
    const formData = new FormData(form);
    const formJSON = Object.fromEntries(formData.entries());

    alert("Data generation with nbRows="+formJSON.nbRows+" nbCols="+formJSON.nbCols);
}
return (
    <form onSubmit={handleOnSubmit}>
    ...
```

Now we will create a function updateGenConfigAndGenerate in App.js to update the store:

```
function App() {
    const [genConfig,setGenConfig] = useState({nbRows:4,nbCols:4});
    const initGenData = genGridData(genConfig.nbRows,genConfig.nbCols);
    const [genData,setGenData] = useState(initGenData);
    const updateGenConfigAndGenerate = function(newGenConfig){
        setGenConfig(newGenConfig);
        setGenData(genGridData(newGenConfig.nbRows,newGenConfig.nbCols))
    }
    ...
```

Add a property to send it to the ControlBar component:

```
function ControlBar({genConfig,onSubmitGenAction}){
```

Call it in handleOnSubmit function:

```
const handleOnSubmit = function(event){
    // Prevent the browser from reloading the page
    event.preventDefault();

    // // Read the form data
    const form = event.target;
    const formData = new FormData(form);
    const formJSON = Object.fromEntries(formData.entries());
    formJSON.nbRows = parseInt(formJSON.nbRows);
    formJSON.nbCols = parseInt(formJSON.nbCols);
    alert("Data generation with nbRows="+formJSON.nbRows+" nbCols="+formJSON.nbCols);
    onSubmitGenAction(formJSON);
}
```

send the function updateGenConfigAndGenerate in the ControlBar component property:

```
<div id="control-bar-container">
    <ControlBar genConfig={genConfig} onSubmitGenAction={updateGenConfigAndGenerate}/>
</div>
```

Now if you change the value of nbCols or nbRows in the UI, the size of the matrix will change.

We can also send the genConfig data to Matrix data to display the number of rows and columns in the margin.

Exercice3: add a property in Matrix component to send genConfig data.

In Matrix.js, we can now add a margin at the left and the top of the svg to display the numbers of columns and rows:

```
import { getDefaultFontSize } from '../../utils/helper';
...
function Matrix({matrixData,genConfig}){
...
    const margin={left:100,top:100}
    const fontSize=getDefaultFontSize()
    const marginLabelsToMatrix=5;
    return(
        <svg width="100%" height="100%" >
            <g transform={"translate("+(margin.left-
marginLabelsToMatrix)+","+(margin.top+fontSize)+")"}>
                <text className="ColLabels" text-anchor="end">Nb rows=?</text>
            </g>
            <g transform={"translate("+(margin.left)+","+(margin.top-
marginLabelsToMatrix)+")"}>
                <text className="RowLabels">Nb cols=?</text>
            </g>
            <g transform={"translate("+margin.left+","+margin.top+")"}>
                {renderMatrix()}
            </g>
        </svg>
    )
}
```

Exercice4: use the genConfig property to replace '?' by the nb of rows and columns in <text> elements.

Visual encoding of cell values

This part concerns the Matrix component.

Exercice5: choose an appropriate visual encoding to represent the numerical data information in nbProductSold and salesGrowth of data items (numerical values for quantity and rate in [0,1] respectively).

Change the Matrix Component to visualize: the name of 10 companies in rows, the name of 10 countries in columns, the number of products sold and the sales growth for each company in each country.