Utiliser d3.js pour afficher des graphiques avec Vue.js

Exemple simple

d3.js est une librairie javascript très populaire pour faire des visualisations de données. Elle permet notamment de faire des visualisations réactives et dynamiques : quand les données changent, les visualisations évoluent de manière animée. C'est notamment cet aspect qui l'a rendue très populaire, car permettant de réaliser un **effet waouh**. Cependant, les principaux framework javascript permettant de réaliser des applications web utilisent également le data-binding pour leur coté réactif.
Comment concilier les 2 pour réaliser une visualisation dans une application web sans faire une usine à gaz ? Nous allons voir un cas pratique avec le framework Vue.js et une visualisation de type **force graph**.

Chacun ses responsabilités

Pour réaliser cette visualisation dynamique, nous allons faire le rendu et la gestion des évènements avec Vue.js et les calculs avec d3.js.

Data-binding et réactivité : Vue.js pour manipuler le DOM

La grande force de Vue.js, c'est de pouvoir manipuler le DOM et gérer les évènements facilement. Les visualisations interactives sont très souvent faites en SVG. C'est un langage à balise basé sur XML, ce qui signifie qu'un élément SVG et ses sous-éléments font partie du DOM.
On peut donc générer une représentation SVG à partir d'un template Vue.js. Dès que les données seront mises à jour, le binding de Vue.js va faire en sorte de mettre à jour la représentation SVG. Cela permet aussi un meilleur découplage entre la préparation des données et leur affichage.

Le fait de pouvoir utiliser un template écrit en pseudo HTML est un avantage pour ceux qui préfèrent manipuler le DOM de manière déclarative. Le fait d'utiliser Vue.js n'est pas un problème pour les autres car le framework permet aussi de manipuler le DOM de manière fonctionnelle.

d3.js pour les calculs

Depuis la version 4.0, d3.js est modulaire, et c'est une excellente chose car cela permet une meilleure intégration avec les frameworks javascript récents. On peut donc importer uniquement les parties qui nous intéressent, ce qui permet de produire un livrable plus léger. Pour plus de détails sur les différents modules disponibles, vous pouvez vous référer à cet article. Dans cette liste, nous n'utiliserons pas les modules relatifs à la manipulation du DOM et la gestion des évènements.

Le code

Nous avons adapté cette démo qui utilise des **web workers** pour réaliser le calcul des forces. Les librairies utilisées sont les suivantes :

- https://cdnjs.cloudflare.com/ajax/libs/vue/2.2.6/vue.min.js
- https://d3js.org/d3-collection.v1.min.js
- https://d3js.org/d3-dispatch.v1.min.js
- https://d3js.org/d3-quadtree.v1.min.js
- https://d3js.org/d3-timer.v1.min.js
- https://d3js.org/d3-force.v1.min.js
- https://d3js.org/d3-array.v1.min.js (utilisée uniquement pour la génération du graphe de démo via d3.range)

Index.html

```
   <script src="https://cdnjs.cloudflare.com/ajax/libs/vue/2.2.6/vue.min.js">
      </script>
    <script src="https://d3js.org/d3-collection.v1.min.js"></script>
    <script src="https://d3js.org/d3-dispatch.v1.min.js"></script>
    <script src="https://d3js.org/d3-quadtree.v1.min.js"></script>
    <script src="https://d3js.org/d3-timer.v1.min.js"></script>
    <script src="https://d3js.org/d3-force.v1.min.js"></script>
    <script src="https://d3js.org/d3-array.v1.min.js"></script>
    <script src="./script.js"></script>
```

Ce code peut être facilement adapté pour faire un composant Vue.js qui prendrait quelques propriétés en entrée, comme par exemple le graphe ou les données servant à le générer.

Le template HTML

Le template est assez simple : nous avons un élément **svg** qui contient un élément **line** pour chaque lien du graphe et un élément **circle** pour chaque nœud.

Index.html

```
<div id="app">
        <svg xmlns="http://www.w3.org/2000/svg" :width="width+'px'" :height="h
eight+'px'" @mousemove="drag($event)"
            @mouseup="drop()" v-if="bounds.minX">
            <line v-
for="link in graph.links" :x1="coords[link.source.index].x" :y1="coords[link.s
ource.index].y"
                :x2="coords[link.target.index].x" :y2="coords[link.target.inde
x].y" stroke="black" stroke-width="2" />

            <circle v-
for="(node, i) in graph.nodes" :cx="coords[i].x" :cy="coords[i].y" :r="20"
                :fill="colors[Math.ceil(Math.sqrt(node.index))]" stroke="white
" stroke-width="1"
                @mousedown="currentMove = {x: $event.screenX, y: $event.screen
Y, node: node}" />
        </svg>
    </div>
```

Le code javascript

Les données avec leurs valeurs initiales sont définies dans le champ **data**. Nous avons notre **graph** généré de manière procédurale. **width** et **height** permettent de régler la taille de la visualisation qui peut n'occuper qu'une partie de l'écran. **padding** et **colors** sont la pour l'esthétique. **currentMove** permet des interactions avec les nœuds. **simulation** permet de géré le coté dynamique de la visualisation, nous en reparlerons après.
D'autres propriétés sont automatiquement calculées de manière réactive dans le champ **computed**. La propriété **bounds** permet de calculer un cadre pour que le graphe prenne toute la place possible. La propriété **coords** permet de faire en sorte que les coordonnées calculées par la simulation soient bien rendues à l'écran : les nœuds aux limites ne sont pas tronqués grâce au **padding** et tout l'espace est occupé en utilisant **width**, **height** et **bounds**.
Au moment de la création de l'application, la simulation est lancée. C'est notamment à l'appel de la méthode **d3.forceSimulation()** que l'on va régler la gravité, les forces de repoussement, la longueur des liens, ... Pour plus de détails sur les réglages possibles, vous pouvez vous référer à cette page.
Enfin, il y a 2 méthodes **drag** et **drop** qui permettent de pouvoir interagir avec les nœuds en les déplaçant.

script.js

```
new Vue({
    el: '#app',
    data: {
      graph: {
        nodes: d3.range(100).map(i => ({ index: i, x: null, y: null })),
        links: d3.range(99).map(i => ({ source: Math.floor(Math.sqrt(i)), targ
et: i + 1 }))
      },
      width: Math.max(document.documentElement.clientWidth, window.innerWidth
|| 0),
      height: Math.max(document.documentElement.clientHeight, window.innerHeig
ht || 0) - 40,
      padding: 20,
      colors: ['#2196F3', '#E91E63', '#7E57C2', '#009688', '#00BCD4', '#EF6C00
', '#4CAF50', '#FF9800', '#F44336', '#CDDC39', '#9C27B0'],
      simulation: null,
      currentMove: null
    },
    computed: {
      bounds() {
        return {
          minX: Math.min(...this.graph.nodes.map(n => n.x)),
          maxX: Math.max(...this.graph.nodes.map(n => n.x)),
          minY: Math.min(...this.graph.nodes.map(n => n.y)),
          maxY: Math.max(...this.graph.nodes.map(n => n.y))
        }
      },
      coords() {
        return this.graph.nodes.map(node => {
          return {
            x: this.padding + (node.x - this.bounds.minX) * (this.width - 2*th
is.padding) / (this.bounds.maxX - this.bounds.minX),
```

```
            y: this.padding + (node.y - this.bounds.minY) * (this.height - 2*t
his.padding) / (this.bounds.maxY - this.bounds.minY)
          }
        })
      }
    },
    created(){
       this.simulation = d3.forceSimulation(this.graph.nodes)
          .force('charge', d3.forceManyBody().strength(d => -100))
          .force('link', d3.forceLink(this.graph.links))
          .force('x', d3.forceX())
          .force('y', d3.forceY())
    },
    methods: {
      drag(e) {
        if (this.currentMove) {
          this.currentMove.node.fx = this.currentMove.node.x - (this.currentMo
ve.x - e.screenX) * (this.bounds.maxX - this.bounds.minX) / (this.width - 2 *
this.padding)
          this.currentMove.node.fy = this.currentMove.node.y - (this.currentMo
ve.y - e.screenY) * (this.bounds.maxY - this.bounds.minY) / (this.height - 2 *
 this.padding)
          this.currentMove.x = e.screenX
          this.currentMove.y = e.screenY
        }
      },
      drop(){
        delete this.currentMove.node.fx
        delete this.currentMove.node.fy
        this.currentMove = null
        this.simulation.alpha(1)
        this.simulation.restart()
      }
    }
  })
```

Comparaison avec l'approche de tout faire avec d3.js

Nous avons présenté dans cette article une approche pour intégrer un framework réactif comme **Vue.js** avec **d3.js** en laissant le premier faire les manipulations de **DOM** et la gestion des évènements. Il y a une autre manière de faire : l'idée est de faire un composant qui va créer un élément **svg** avec un identifiant. On utilise ensuite classiquement **d3.js** pour faire les manipulations de **DOM** et la gestion d'évènements dans cet élément. Notre composant fonctionne alors comme une sorte de boîte noire pour le reste du framework réactif.
L'avantage est de pouvoir intégrer facilement une large base d'exemples **d3.js** sans avoir à les réadapter. Cependant le code est hétérogène : pour ce composant de visualisation, les évènements et la construction du **DOM** ne seront pas réalisés de la même manière que dans les autres composants de l'application.

Nous pensons que l'approche décrite dans cet exercice est plus adaptée si vous connaissez bien **Vue.js** et si vous débutez, ou avez une base de code faible avec **d3.js**. Finalement, il n'y a pas de méthode meilleure que les autres : tout dépend du problème à résoudre et de vos contraintes. Le plus important est de bien cerner les forces et les faiblesses de chaque approche pour vous aider à faire le meilleur choix possible.