Initiation à VueJS

1 Introduction

Un *framework frontend* est un ensemble de bibliothèques permettant la réalisation et la mise en place d'interfaces utilisateurs pour le web. Actuellement, trois grands acteurs se partagent le podium : Angular, React et VueJS. D'après plusieurs sources sur Internet, le plus simple à prendre en main reste VueJS, et semble donc le plus adapté pour une première approche du développement *frontend* avec un *framework*.

Les langages utilisés sont le HTML5 et le Javascript. Ce framework respecte le standard MVVM, dans lequel le modèle (les données) influence la vue (l'interface graphique) et inversement. Dans un projet de taille moyenne, nous verrons comment décomposer les différents éléments de l'application en composants, capables de communiquer entre eux. Nous verrons également comment utiliser une bibliothèque de composants (Vuetify) afin de produire une application Web responsive, au design actuel, sans toutefois avoir besoin de s'attarder sur le CSS.

2 Création d'une première application

Dans un premier temps, nous verrons comment incorporer VueJS directement dans les fichiers HTML, afin d'en saisir le fonctionnement. Puis, nous aborderons des outils en ligne de commande utilisés par la communauté afin de créer des projets de taille moyenne.

```
1  <!doctype HTML>
2  <html>
3    <head>
4      <title>Initiation VueJS</title>
5      <meta charset="utf8"/>
6    </head>
7    <body>
8      <div id="app">Hello, world !</div>
9      <script language="javascript">
10       //TODO
11     </script>
12   </body>
13 </html>
```

Le code ci-dessus représente un fichier HTML basique, sur lequel nous ajouterons progressivement des fonctionnalités de Vue. Commençons par importer le CDN nécessaire au fonctionnement du *framework*, via la balise suivante, au sein de notre code (par exemple dans le header de votre page) :

```
<script src="https://cdn.jsdelivr.net/npm/vue@2.6.12"></script>
```

2.1 Instance de vue

Notre application sera représentée par la balise `<div id="app">...</div>`, dans laquelle on placera les divers éléments de la vue. Le modèle sera quant à lui défini dans les balises de script. On déclare une nouvelle instance de Vue prenant sur notre division via le code suivant :

```
new Vue({
  el: '#app',
  data:{
    message:"Hello, world!"
  }
});
```

On utilise le constructeur de la classe Vue pour une nouvelle instance, et celui pour paramètre un objet JSON ayant divers attributs optionnels. Ici, l'attribut `el` désigne l'élément HTML qui portera l'application, alors que l'attribut `data` contiendra les divers données qui évolueront au fil de l'exécution de l'application. Voyons comment afficher le message au sein du HTML :

```
<div id="app"> {{ message }}</div>
<script language="javascript">
  new Vue({
    el: '#app',
    data:{
      message:"Hello, world!"
    }
  });
</script>
```

Les différents attributs de `data` sont accessibles via la notation {{ element }}. Ici, le framework remplace directement {{ message }} par la chaîne "Hello, world!". Si celle-ci est modifiée au cours de l'exécution, affichage sera automatiquement actualisé.

En plus des données, il est possible de déclarer des méthodes et des attributs calculés au sein de notre instance. Ici, nous déclarons une méthode augmentant le message à chaque clic sur un bouton. On remarquera que la capture du clic sur le bouton se fait via la primitive `v-on:click`, qui permet le lancement d'une exécution en rapport avec la Vue, ici l'appel d'une méthode. On verra par la suite qu'il est possible de capturer un grand nombre de type d'évènements (changement, survol, ...), tout comme en HTML et JS standard.

```
<div id="app">
  <p>{{ message }}</p>
  <input type="button" v-on:click="augmentation()"
   ↪  value="+"/>
</div>
<script language="javascript">
new Vue({
  el: '#app',
  data:{
    message:"Hello, world!"
  },
  methods:{
    augmentation: function(){
      this.message = this.message+"!";
    }
  }
});
</script>
```

Voyons maintenant la prise en charge des attributs calculés. Pour faire simple, ceux-ci s'utilisent comme des attributs classiques, mais se déclarent comme des méthodes. Ils permettent par exemple d'accéder rapidement à la moyenne d'une liste de valeurs, à la concaténation de deux attributs, ... En théorie, ils n'apportent rien de plus que les méthodes, cependant, un attribut calculé n'est recalculé que lors de la mise à jour des données du modèle. Par conséquent et d'un point de vue complexité algorithmique, il est plus intéressant de faire appel à ces attributs calculés plutôt qu'aux méthodes lorsque l'on a le choix.

```
<div id="app">{{ somme }}</div>
<script language="javascript">
new Vue({
  el: '#app',
  data:{
    valeurs:[1,2,3,4,5]
  },
  computed:{
    somme: function(){
      return this.valeurs.reduce(function(total, value){
        return total+value;
      });
    }
  }
});
</script>
```

Ici, l'attribut calculé `somme` ne sera recalculé que lors de la modification du tableau `valeurs`. Par conséquent, même si l'attribut calculé `somme` était appelé plusieurs fois, le calcul est lui-même ne serait effectué qu'une seule fois.

2.2 Communication vue-modèle

On l'a vu, il est possible d'afficher les données du modèle via la syntaxe donnée plus haut. Il est également possible de conditionner l'affichage en fonction des données du modèle. Ainsi, il est possible de boucler sur un tableau, afficher ou non un élément en fonction d'une donnée, ... Commençons par l'utilisation de `v-for` : cela permettra la création d'un élément html pour chacun des éléments d'un tableau de données.

```
<div id="app">
  <ul>
    <li v-for="fruit in fruits" :key="fruit.nom">
      <strong>{{ fruit.nom }}</strong>
      {{ fruit.prix }} euros
    </li>
  </ul>
</div>
<script language="javascript">
new Vue({
  el: '#app',
  data:{
    fruits:[
      {nom:"oranges", prix:1.5},
      {nom:"fraises", prix:3},
      {nom:"pommes", prix:0.5},
      {nom:"cerises", prix:2}
    ]
  }
});
</script>
```

Dans notre exemple, le modèle contient un tableau d'objets, chacun des éléments représentant un fruit avec un nom et un prix. Dans le code HTML, on définit une liste qui contiendra un élément `<li>..</li>` par élément contenu dans le tableau. La primitive `v-for="element in tableau"` est déclaré à l'intérieur de la balise à répéter. De plus, il est nécessaire d'ajouter l'attribut `:key` afin de fournir une clé primaire unique à chacun des éléments générés, et ceci pour permettre à Vue d'actualiser l'affichage lors de la modification des données. De la même manière, il est possible de filtrer les affichages avec la primitive `v-if="expression booléenne"`.

```
<li v-for="fruit in fruits" :key="fruit.nom"
  ↪ v-if="fruit.prix>=2">
```

Si on modifie notre code comme ci-dessus, seuls les fraises et les cerises seront affichées. Comme on peut le voir ici, `v-for` a une plus forte priorité que `v-if`, ce dernier sera appliqué à chaque tour de boucle. Il existe également `v-else` et `v-else-if` qui s'utilisent de la manière suivante :

```
<li v-for="fruit in fruits" :key="fruit.nom">
  <strong>{{ fruit.nom }}</strong>
  <span v-if="fruit.prix<=1" style="color:green">
    {{ fruit.prix }} euros
  </span>
  <span v-else-if="fruit.prix<=2" style="color:orange">
    {{ fruit.prix }} euros
  </span>
  <span v-else style="color:red">
  {{ fruit.prix }} euros
  </span>
</li>
```

Si l'on veut que l'attribut d'une balise comprenne une donnée du modèle, il suffit de préfixer celui-ci par `v-bind:`, ou tout simplement " :". Modifions notre exemple précédent pour ajouter la couleur d'affichage à chaque objet représentant un fruit :

```
<script language="javascript">
new Vue({
  el: '#app',
  data:{
    fruits:[
    {nom:"oranges", prix:1.5, color:"orange"},
    {nom:"fraises", prix:3, color:"rouge"},
    {nom:"pommes", prix:0.5, color:"vert"},
    {nom:"cerises", prix:2, color:"orange"}
    ]
  }
});
</script>
```

Ici, on choisira de définir des classes css modifiant la couleur du texte, nommées vert, orange et rouge. On ajoute donc à notre code la portion css suivante :

```
<style>
.orange{color:orange;}
.rouge{color:red;}
.vert{color:green;}
</style>
```

Il ne suffit plus qu'à attribuer la classe au `span` contenant le prix, en lui attribuant la valeur contenu dans l'objet correspondant :

```
<span :class="fruit.color">
{{ fruit.prix }} euros
</span>
```

De la même manière, il est possible de procéder dans l'autre sens, en liant la valeur

saisie dans un champ de formulaire au modèle. Ainsi, un élément de formulaire ayant la propriété `v-model="x"` régira la valeur de `data.x`. Dans l'exemple suivant, on affiche la chaîne `message` inversée via l'attribut calculé `inverse`. De plus, on met en place une boîte texte régissant la valeur de `message`. Ainsi, l'affichage se met automatiquement à jour, affichant l'inverse de ce que l'on tape :

```
<div id="app">
  <p>{{ inverse }}</p>
  <input type="text" v-model="message"/>
</div>
<script language="javascript">
new Vue({
  el: '#app',
  data:{
    message:""
  },
  computed:{
    inverse:function(){return
      ↪ this.message.split("").reverse().join("");}
  }

});
</script>
```

3 Première mise en application

Cela fait beaucoup de notions d'un coup, on va donc procéder à un premier exercice. Vous devez créer une liste de courses :

- vous affichez tous les éléments de la liste, avec la possibilité d'augmenter ou réduire les quantités via deux boutons sur chaque ligne ;
- vous mettez à disposition un champ texte et un bouton permettant d'ajouter un objet dans la liste.

Pour l'instant, ne vous préoccupez pas du design, on verra par la suite qu'il existe des bibliothèques de composants permettant de réaliser des visuels aboutis sans effort.

4 Les composants

Dès lors que vous travaillez sur un projet de taille conséquente, votre code risque de devenir difficile à maintenir sans une organisation adaptée. Dans notre cas, le fait de travailler uniquement sur l'instance de la Vue est problématique, car le code deviendra trop dense pour être facilement compréhensible. De plus, le manque de modularité nécessiterait de reprendre le travail à zéro à chaque nouveau projet. Pour pallier à cette difficulté, Vue utilise un système de composants. Chaque composant comprend une partie HTML, une partie style CSS et une partie script. Cela permet de décomposer proprement son code, réutiliser le travail effectuer précédemment, mais également la création de librairies de composants.

4.1 Déclaration

Commençons par définir note composant : on veut créer un bouton enregistrant et affichant le nombre de clics. Pour bien faire, on voudrait également déclarer celui-ci dans un fichier séparé, afin de ne pas polluer notre fichier principal. On créé donc un fichier "bouton.js", dans un répertoire "composants", à côté de notre fichier "index.html", et avec le contenu suivant :

```
Vue.component('monBouton', {
  data: function () {
    return {
      count: 0
    }
  },
  template: '<button v-on:click="count++">Clics : {{ count }}</button>'
})
```

Ici, notre composant dispose :

- d'un nom, qui est 'monBouton' en camelCase, mais également 'mon-bouton' en kebab-case : pour la déclaration du nom, vous pouvez utiliser l'un ou l'autre ;
- d'un champ data, qui est une **fonction** retournant un objet, et qui fonctionnera comme l'attribut data de l'instance de Vue ;
- d'un champ template fournissant le code HTML du bouton, sous la forme d'une chaîne de caractères, permettant l'utilisation des données du composants, des méthodes, ...

Un composant peut également disposer de méthodes, d'attributs calculés, ... Voyons maintenant comment les utiliser au sein de notre fichier principal. L'inclusion se fait simplement comme n'importe quel fichier javascript auxiliaire, et l'appel du composant se fait via un balise comportant son nom en **kebab-case**, même si vous avez déclaré celui-ci en camelCase :

```
  <div id="app">
    <mon-bouton></mon-bouton>
    <mon-bouton></mon-bouton>
  </div>
  <script src="composants/bouton.js"></script>
  <script language="javascript">
  new Vue({
    el: '#app',
    data:{
      message:"nombre de clics"
    }

  });
  </script>
```

Deux instances de notre bouton sont créées et insérées dans la page. Chacune gère ses données indépendamment de l'autre, et le nombre de clics affiché est donc différent

d'un bouton à l'autre. On remarque que l'instanciation de la Vue dans le fichier principal est toujours présente, et qu'un message est inclus dans l'attribut `data`. On verra dans la partie suivante comment communiquer ce message aux composants.

4.2 Props

Il est possible de passer des informations aux composants via les *props*. Pour cela, il suffit de modifier la déclaration de notre composant :

```
1  Vue.component('monBouton', {
2    props:['prefixe'],
3    ...
```

le champ `props` de notre composant est un tableau de chaînes de caractères, chacune de ces chaînes étant un clef permettant d'utiliser l'objet correspondant, passé lors de l'instanciation du composant. Ainsi, on peut utiliser `prefixe` dans le template de notre composant :

```
1  Vue.component('monBouton', {
2    ...
3    template: '<button v-on:click="count++">{{ prefixe+" "+count
     ↪  }}</button>'
4  })
```

Il est désormais nécessaire de passer le préfixe à utiliser lors de l'instanciation du composant. Pour cela, on procède de la manière suivante :

```
1  <mon-bouton prefixe="bonjour"></mon-bouton>
2  <mon-bouton :prefixe="message"></mon-bouton>
```

Dans le premier cas, on passe la chaîne "bonjour", alors que dans le deuxième cas, l'attribut "prefixe" est précédé du symbole " :", et l'objet passé est alors `data.message`.

4.3 Signaux

Voyons maintenant comment remonter une information d'un composant vers notre instance de Vue. Pour cela, il faut utiliser les signaux : lors d'un évènement, on peut choisir d'émettre un signal qui pourra être capturer par l'application, et déclencher ainsi une réaction de l'instance de Vue. On va modifier notre application courante afin de compter le nombre de clics total, sans distinction du bouton appuyé. Pour cela, on ajoute un paragraphe affichant la valeur d'une nouvelle donnée :

```
<div id="app">
  <p>{{ clics }}</p>
  <mon-bouton prefixe="bonjour"></mon-bouton>
  <mon-bouton :prefixe="message"></mon-bouton>
</div>
<script src="bouton.js"></script>
<script language="javascript">
new Vue({
  el: '#app',
  data:{
    message:"nombre de clics",
    clics:0
  }
});
</script>
```

On modifie maintenant notre fichier "bouton.js" : l'appui sur le bouton appellera la méthode `clic`, qui, en plus d'incrémenter le compteur du bouton, émettra l'évènement nommé "clic", via la méthode `this.$emit("nomDeMonEvenement")` :

```
methods:{
  clic:function(){
    this.count++;
    this.$emit("clic");
  }
},
template: '<button v-on:click="clic()">{{ prefixe+" "+count
 ↪ }}</button>'
```

De nouveau dans le fichier "index.html", on ajoute maintenant les listeners sur les deux boutons. A chaque fois que l'un d'entre eux émettra "clic", cela provoquera l'appel de la méthode `onClic`, qui incrémentera le compteur :

```
<mon-bouton prefixe="bonjour" v-on:clic="onClic"></mon-bouton>
<mon-bouton :prefixe="message" v-on:clic="onClic"></mon-bouton>
...
methods:{
  onClic:function(){
    this.clics++;
  }
}
```

Ainsi, on peut relayer des informations depuis les composants vers le modèle. De plus, il est possible d'ajouter un objet en paramètre à la fonction d'émission. Par exemple, si l'on veut passer le nombre de clics du bouton émettant le signal, on peut écrire `this.$emit("clic", this.clics)`. Il faut alors modifier le fichier "index.html" de la manière suivante :

```
<mon-bouton prefixe="bonjour" v-on:clic="onClic"></mon-bouton>
<mon-bouton :prefixe="message" v-on:clic="onClic"></mon-bouton>
...
methods:{
  onClic:function(params){
    console.log("Reçu : "+params);
    this.clics++;
  }
}
```

Ici, le comportement de notre application n'est pas modifié, mis à part que celle-ci affiche dans la console javascript le compteur du bouton sur lequel on clique.

4.4 Déclaration locale et imbrication de composants

Jusqu'à présent, nos composants étaient déclarés en global. Cependant, il est possible de faire une déclaration locale des composants afin d'éviter d'éventuels conflits avec d'autres bibliothèques. Modifions notre fichier "bouton.js" de la manière suivante :

```
var monBouton =  {
  props:['prefixe'],
  data: function () {
    return {
      count: 0
    }
  },
  methods:{
    clic:function(){
      this.count++;
      this.$emit("clic");
    }
  },
  template: '<button v-on:click="clic()">{{ prefixe+" "+count
   ↪  }}</button>'
};
```

Ici, la méthode `Vue.component` n'est plus appelée, on déclare l'objet qu'on lui passait en paramètre, qu l'on stocke dans une variable, ici `monBouton`. Ensuite, on modifie la déclaration de l'instance de Vue de la manière suivante :

```
new Vue({
  el: '#app',
  components:{
    'mon-bouton':monBouton
  },
  data:{
    message:"nombre de clics",
    clics:0
  },
  methods:{
    onClic:function(){
      this.clics++;
    }
  }

});
```

On ajoute un champ `components` sous la forme d'un objet énumérant des couples (chaîne de caractère, composant). Ici, on associe la variable `monBouton`, déclarée dans le fichier "bouton.js", à la clef "mon-bouton" qui servira de nom de balise HTML. L'intérêt est pour l'instant limité, mais cela permettra d'avoir des composants différents ayant la même clef, dans des scopes différents : le bouton d'un formulaire d'inscription sera un composant différent du bouton d'envoi d'un message sur un forum, mais utilisera le même nom de balise.

Ici, on a vu que l'on pouvait déclarer l'utilisation de composants au sein de l'instance de Vue. On peut également faire de même au sein de la déclaration d'un composant. Par exemple, vous pouvez créer un composant formulaire, qui lui même utilise des composants champs texte, boutons, listes déroulantes, ... Cependant, le javascript "vanilla" ne connaît pas de primitives d'inclusion de code. Pour l'instant, vous devrez soit travailler dans un seul fichier js, soit gérer les inclusions dans le fichier html utilisant vos composants. On verra plus loin dans ce document comment utiliser des outils plus avancés permettant une meilleure modularité de votre travail.

5 Deuxième mise en application

On va créer un composant formulaire respectant le cahier des charges suivant :

- le formulaire peut comporter plusieurs sections, chacune ayant un titre et pouvant accueillir des champs texte et des listes déroulantes ;
- le nombre de sections ainsi que le nombre d'éléments de chaque catégorie au sein des sections est variable, et sera défini au sein des props passées au formulaire ;
- lors du clic sur le bouton d'envoi du formulaire, celui-ci recueille toutes les données et les émet.

Vous partirez pour cela du fichier "formulaire.html" fourni sur Moodle, qui ne devra pas être modifié. Encore une fois, le style graphique n'est pas important ici, on verra comment rendre ça agréable dans la section suivante.

6 Bibliothèque Vuetify

Vuetify est l'une des bibliothèques de composants disponibles pour le *framework* Vue. Elle intègre un grand nombre de composants génériques, documentés sur le site suivant :

Lien vers la documentation

Vous trouverez, notamment via la section *UI Components* dans le menu de gauche, le détail d'utilisation de chaque composant de la bibliothèque. L'importation de la bibliothèque peut se faire de plusieurs manières, nous utiliserons dans un premier temps les CDN suivants, chacun d'entre eux à importer via une balise de script :

```
https://cdn.jsdelivr.net/npm/vue@2.x/dist/vue.js
https://cdn.jsdelivr.net/npm/vuetify@2.x/dist/vuetify.js
```

De la même manière, il va être nécessaire de placer quelques balises en entête de votre fichier, pour importer des feuilles de styles, des icônes, des polices, ... Vous trouverez celles-ci dans le fichier "template.html" fourni sur moodle. Lors de l'utilisation de Vuetify, il est nécessaire de travailler à l'intérieur d'une balise `v-app` afin de permettre le bon fonctionnement des composants. Celle-ci sera l'unique enfant direct de la balise hébergeant l'application. De plus, il faudra renseigner un nouvel attribut au sein de notre instance de Vue, qui lui-même instanciera Vuetify :

```
new Vue({
  el: '#app',
  vuetify: new Vuetify(),
})
```

6.1 Tour de quelques composants disponibles

On commence par `v-card`, qui correspond à un bloc structuré avec un titre `v-card-title`, un sous-titre `v-card-subtitle`, un texte `v-card-text` et un emplacement pour une barre d'action `v-card-actions`, qui seront placés dans la carte de haut en bas. Chaque sous-élément est optionnel : pas besoin d'intégrer la barre d'actions si vous ne souhaitez pas ajouter de boutons à votre carte.

```
  Vue.component("ma-carte", {
    data:function() {
      return {
        titre:"Ma super carte",
        texte:"Hello, world!"
      };
    },
    template:"<v-card> <v-card-title> {{ titre }} </v-card-title>
     ↪ <v-card-text> {{ texte }} </v-card-text> <v-card-actions>
     ↪ <v-btn> OK </v-btn> <v-btn> Annuler </v-btn>
     ↪ </v-card-actions> </v-card>"
  });
```

On voit ci-dessus une manière d'utiliser cet élément via la création d'un composant. Vous pourrez bien entendu ajouter du contenu plus riche qu'un simple texte, notamment des images, des icônes, et travailler avec une architecture sous la forme d'une grille. Cet élément sera probablement le plus utile dans le design de votre application. La bibliothèque intègre un grand nombre de composants, on procédera donc à une présentation de quelques-un d'entre eux, via le tableau suivant regroupant, pour chaque composant, le nom, la balise, une description succincte ainsi que le lien vers la documentation.

Balise	**Description**	**Lien**
Structure		
`v-container`	Contenant classique	Documentation
`v-row`	Contenant disposant les enfants en ligne	Documentation
`v-col`	Contenant disposant les enfants en colonne	Documentation
`v-spacer`	Composant prenant toute la place disponible	Documentation
Widgets		
`v-card`	Bloc de base avec titre, contenu et contrôles	Documentation
`v-expansion-panels`	Succession de blocs déroulants	Documentation
`v-carousel`	Bloc défilant des slides	Documentation
`v-img`	Affichage simple d'une image	Documentation
`v-steppers`	Affichage un processus par étape	Documentation
`v-chip`	Bloc de petite taille	Documentation
`v-dialog`	Affichage d'une boite de dialogue	Documentation
`v-divider`	Ligne de séparation	Documentation
`v-footer`	Bloc de pied de page	Documentation
`v-icon`	Affichage d'une icône Material Design	Documentation
`v-list`	Affichage de texte sous forme de liste	Documentation
`v-menu`	Affichage d'un menu déroulant	Documentation
`v-navigation-drawer`	Affichage d'un menu de navigation	Documentation
`v-pagination`	Numérotation de pages	Documentation
`v-tabs`	Gestion d'onglets	Documentation
`v-timeline`	Affichage stylisé d'une chronologie	Documentation
`v-treeview`	Affichage d'une arborescence	Documentation
`v-sheet`	Bloc simple	Documentation
Éléments de formulaire		
`v-btn`	Bouton	Documentation
`v-checkbox`	Case à cocher	Documentation
`v-text-field`	Champ texte	Documentation
`v-textarea`	Zone de texte	Documentation
`v-switch`	Switch permettant d'alterner entre deux valeurs	Documentation
`v-select`	Liste déroulante	Documentation
`v-file-input`	Ouvre une fenêtre de sélection de fichier	Documentation

6.2 Changement de style graphique

Il est bien sûr possible de modifier le style graphique des composants proposés par la bibliothèque. Dans cette section, nous travaillerons sur le code suivant :

```
<div id="app">
  <v-app>
    <v-btn>Mon bouton</v-btn>
  </v-app>
</div>
```

Cela affiche un bouton, avec le design par défaut, avec un texte en noir sur un fond gris. Si l'on souhaite changer ces couleurs, il faudra lui associer les classes correspondantes. Plusieurs palettes sont disponibles, dont celle de Material Design. En plus du noir et du blanc, on pourra utiliser les couleurs suivantes :

red	pink	purple	dark-purple	indigo
blue	light-blue	cyan	teal	green
light-green	lime	yellow	amber	orange
deep-orange	brown	blue-grey	grey	shades

TABLE 1 – Couleurs de base

Ces 20 couleurs sont dérivables via les sous classes lighten-1, ..., lighten-5, darken-1, ..., darken-5, accent-1, ... accent-5. Modifions notre code de la manière suivante :

```
<div id="app">
  <v-app>
    <v-btn class="light-blue lighten-1 white--text">Mon
    ↪ bouton</v-btn>
  </v-app>
</div>
```

La première classe définit la couleur de fond, dérivée en une variante plus claire par la deuxième classe. On remarquera que le survol et l'activation du bouton provoque une modification mineure de la couleur, en restant en cohérence avec la couleur définie par la première classe : on reste sur des variantes du bleu clair. La dernière classe définit la couleur du texte.

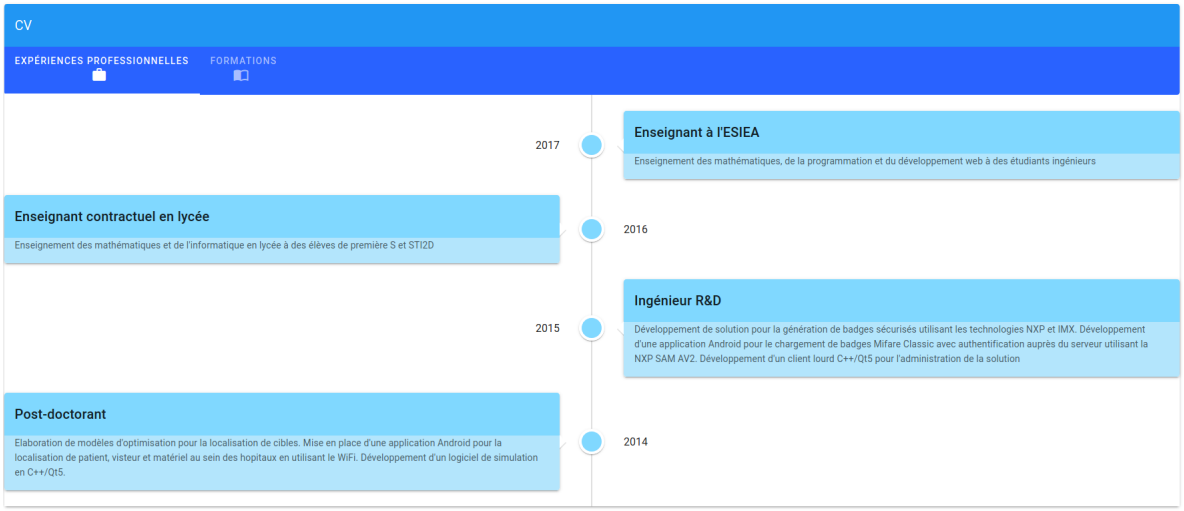
La bibliothèque Vuetify ne dispose pour l'instant pas de modificateurs quant aux bordures elles-même (épaisseur, style et couleur), mais il est cependant possible d'agir sur le rayon de bordure délimitant le composant, via les classes rounded-0, rounded-xs, rounded, rounded-lg, rounded-xl, rounded-pill et enfin rounded-circle. Chacune d'entres elles définit un arrondi de plus en plus accentué. Enfin, l'attribut elevation permet de définir le relief d'un élément, et prend une valeur en 0 et 24.

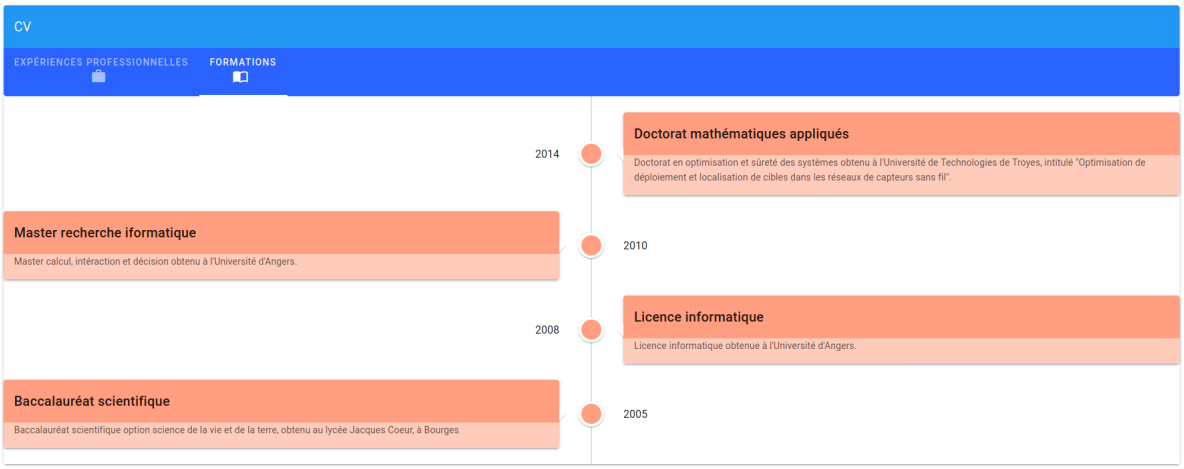
6.3 Création d'un super CV

Ici, nous allons tenter de créer un composant CV utilisant la librairie Vuetify. Les composants impliqués seront les suivants :

- `v-card` ;
- `v-timeline` ;
- `v-tabs` ;
- `v-img` ;
- `v-icon`.

Le résultat attendu est visible dans les deux figures ci-après. Une barre d'onglets permet de passer de la partie "expériences professionnelles" à la partie "formations". On remarquera que les titres des onglets sont accompagnés d'une icône Material Design (voir liste et documentation). Chaque partie est représentée avec une *timeline*, dont les couleurs varient selon la section affichée.





Le composant créé prendra deux props : une liste d'expériences et une liste de formations. Chaque élément de liste sera un objet avec les attributs "annee", "titre" et "texte".

```
1  <div id="app">
2    <v-app>
3      <cv :experiences="experiences" :formations="formations"></cv>
4    </v-app>
5  </div>
```

On pourra alors fournir les différentes informations à afficher au composant. Ici, on choisit de les insérer directement dans notre modèle.

```
new Vue({
  el: '#app',
  vuetify: new Vuetify(),
  data:{
    experiences:[
      {annee:2017, titre:'Enseignant à l\'ESIEA', texte:'...'},
      ...
    ],
    formations:[
      {annee:2014, titre:'Doctorat mathématiques appliqués',
       ↪ texte:'...'},
      ...
    ],
  }
})
```

6.4 Positionnement des éléments

6.4.a Marges et Padding

Des classes ont été mises en place pour faciliter l'application de marges intérieures (padding) et extérieures (margin). Les classes concernées sont de la forme "px-n" et "mx-k", avec x une lettre parmi les suivantes : l, r, t, b, a représentant respectivement gauche, droite, haut, bas, tous, et k un entier entre 0 et 16, définissant une taille de marges de k*4 pixels. Ainsi, si on veut attribuer une marge intérieure gauche de 16 pixels et une marge extérieure de 24 pixels dans toutes les directions, on utilise les deux classes "pl-4" et "ma-6".

6.4.b Système de grille

Comme pour le *framework* Bootstrap, il est possible d'utiliser un système de grille pour disposer les différents éléments d'une page. Pour cela, on utilise les éléments `v-container`, `v-row`, `v-col` et `v-spacer`. Le premier est un conteneur classique, que l'on utilisera pour gérer les alignements de contenu. Les deux suivants permettent de définir une ligne l'élément et une colonne appartenant à une ligne. Enfin, le dernier permet d'insérer un élément invisible prenant l'espace disponible dans la ligne ou la colonne.

La logique est la suivante : le parent dispose d'un espace découpé en douze colonnes, et chaque enfant direct peut occuper de une à douze colonnes. Dans l'exemple suivant, une ligne à quatre enfants, dont trois blocs prenant respectivement 1, 2 et 3 colonnes sur les douze. Entre le deuxième et le troisième enfant est placé un v-spacer, qui prendra l'espace restant (6 colonnes) :

```
<v-row>
  <v-col class="col-1 blue">E1</v-col>
  <v-col class="col-2 red">E2</v-col>
  <v-spacer></v-spacer>
  <v-col class="col-3 green">E3</v-col>
</v-row>
```

Il est possible de prendre en compte différentes tailles de fenêtres afin d'adapter l'affichage du contenu de la page. Vuetify définit les tokens suivants :

- "xs" pour très petit : la page s'affiche sur une largeur inférieure à 600px ;
- "sm" pour petit : la page s'affiche sur une largeur inférieure à 960px ;
- "md" pour moyen : la page s'affiche sur une largeur inférieure à 1264px ;
- "lg" pour large : la page s'affiche sur une largeur inférieure à 1904px ;
- "xl" pour très large : la page s'affiche sur une largeur supérieure à 1904px.

Il est ainsi possible d'utiliser ces tokens pour attribuer des règles d'occupation de l'espace différentes en fonction de la taille de la fenêtre :

```
<v-row>
  <v-col class="col-xl-2 col-lg-3 col-md-4 col-sm-6 col-xs-12
  ↪ blue">E1</v-col>
  <v-col class="col-xl-2 col-lg-3 col-md-4 col-sm-6 col-xs-12
  ↪ red">E2</v-coliv>
  <v-col class="col-xl-2 col-lg-3 col-md-4 col-sm-6 col-xs-12
  ↪ green">E3</v-col>
  <v-col class="col-xl-2 col-lg-3 col-md-4 col-sm-6 col-xs-12
  ↪ blue">E4</v-col>
  <v-col class="col-xl-2 col-lg-3 col-md-4 col-sm-6 col-xs-12
  ↪ red">E5</v-col>
</v-row>
```

Ici, on dispose 5 éléments dans une ligne. Il est important de savoir que si les douze colonnes d'une ligne sont consommées, les éléments suivants passent à la ligne suivante. Ainsi, notre affichage réagira de la façon suivante :

- si la fenêtre fait au moins 1904px de large, alors les cinq éléments prennent 10 colonnes au total, et sont donc affichés sur une même ligne,
- si la fenêtre fait au moins 1264px de large, alors les cinq éléments prennent 10 colonnes au total, les quatre premiers sont sur une même ligne, et le dernier sur la ligne suivante,
- si la fenêtre fait au moins 960px de large, alors les cinq éléments prennent 20 colonnes au total, les trois premiers sont sur une même ligne, et les deux derniers sur la ligne suivante,
- si la fenêtre fait au moins 600px de large, alors les cinq éléments prennent 30 colonnes au total, et prennent trois lignes,
- si la fenêtre fait moins de 600px pixels de large, alors les cinq éléments prennent 60 colonnes au total, et prennent cinq lignes.

Ainsi, si on exécute cet exemple et que l'on fait varier la taille de la fenêtre, on verra les éléments se réorganiser de manière dynamique.

6.4.c Flexbox

Vuetify intègre diverses fonctionnalités du système Flexbox étudié l'année dernière. Pour rappel, Flexbox permet de définir des conteneurs qui aligneront les enfants en ligne ou en colonne. On travaille donc sur deux axes (horizontal et vertical) qui seront l'axe primaire ou secondaire selon l'orientation du conteneur : l'axe primaire et l'axe secondaire d'un conteneur en ligne seront respectivement l'axe horizontal et l'axe vertical.

```
<v-card class="d-flex">
  <div class="col-1 blue">E1</div>
  <div class="col-1 red">E2</div>
  <div class="col-1 green">E3</div>
</v-card>
```

Dans le code présenté ci-dessus, on déclare un bloc de la classe "d-flex", cette division sera donc un conteneur Flexbox disposant ses enfants en ligne par défaut. Si on veut inverser les axes, on procède de la manière suivante :

```
<v-card class="d-flex flex-column fill-height">
  <div class="col-1 blue">E1</div>
  <div class="col-1 red">E2</div>
  <div class="col-1 green">E3</div>
</v-card>
```

Ici, on force l'orientation en colonne, tout en obligeant la carte à occuper toute la hauteur disponible. Les alignements sur l'axe principal se définissent via les classes du conteneur parent :

- "justify-start" : les enfants seront alignés à droite (ligne) ou en haut (colonne) ;
- "justify-end" : les enfants seront alignés à gauche (ligne) ou en bas (colonne) ;
- "justify-center" : les enfants seront centrés
- "justify-space-between" : les enfants seront centrés, tout en les écartant au maximum ;
- "justify-space-around" : les enfants seront centrés, tout en les écartant au maximum mais en gardant un espace avant le premier enfant et après le dernier enfant ;

Les alignements sur l'axe secondaire se définissent via les classes du conteneur parent "align-start", "align-end", "align-center". Il est également possible qu'un enfant est une politique d'alignement sur l'axe secondaire du parent différente. Pour cela, il suffit d'attribuer à l'enfant en question une des classes "align-self-start", "align-self-end", "align-self-center" :

```
<v-card class="d-flex justify-center flex-column fill-height">
  <div class="col-1 blue">E1</div>
  <div class="col-1 red align-self-center">E2</div>
  <div class="col-1 green align-self-end">E3</div>
</v-card>
```

Il ne s'agit là que d'une introduction, vous trouverez plus de détails en cliquant sur le lien suivant : Documentation. Vous pouvez également étudier la documentation du

composant `v-layout`, qui permet d'établir un conteneur Flexbox plus rapidement, en ne passant pas par les classes, mais directement par les attributs de l'objet : Documentation.

7 Communication avec le backend

La communication avec le serveur *backend* peut se faire avec Axios, qui vous permettra de faire des requêtes GET et POST au serveur, et de traiter la réponse (par exemple un JSON) via un callback. Le CDN est le suivant :

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

Dans l'exemple qui suivra, on partira du principe que votre fichier html est fourni par le serveur gérant également le *backend*. Vous devrez donc mettre en place un serveur NodeJS permettant de servir les fichiers :

```
var express = require('express');
var serve_static = require('serve-static');
var http = require('http');

var app = express();
app.use(serve_static(__dirname+"/public"));

var serveur = http.Server(app);
serveur.listen(8080, function(){});
```

Ajoutons maintenant un service permettant l'obtention de `nb` nombres aléatoires, via une requête GET :

```
app.get('/generation/:nb', function (req, res) {
  let obj = {status:'KO', liste:[]};
  if ('nb' in req.params){
    let nb = 0;
    try{
      nb = parseInt(req.params.nb);
    }
    catch(e){
      nb = 0;
    }

    obj.status = 'OK';
    for (let i=0; i<nb; i++){
      obj.liste.push(Math.random());
    }
  }
  res.send(obj);
});
```

Ainsi, en demandant au serveur la page "generation/10", celui-ci devrait nous renvoyer

pour réponse un objet avec un statut (OK ou KO), et une liste contenant 10 nombres générés aléatoirement entre 0 et 1. Passons maintenant au client : on va créer une simple page avec un un champ texte et un bouton. Le champ texte permettra de saisir le nombre de nombres à générer, et le bouton déclenchera la requête :

```
<div id="app">
  <ul>
    <li v-for="(elt, index) in liste" :key="index">{{ elt }}</li>
  </ul>
  <input type="text" v-model="nb"/>
  <button v-on:click="generer()">Générer</button>
</div>
```

Les données du modèle seront `nb` et `liste`, respectivement liées au champ texte et à la liste "ul". Côté javascript, on définit notre instance de Vue avec les données précédemment citées, mais également avec la définition de la méthode `generer`, appelée lors du clic sur le bouton :

```
new Vue({
  el: '#app',
  data:{
    nb:0,
    liste:[]
  },
  methods:{
    generer:function(){
      let self = this;
      axios.get("generation/"+this.nb).then(function(reponse){
        if (reponse.data.status=="OK") self.liste =
        ↪  reponse.data.liste;
      });
    }
  }
});
```

Celle-ci procède aux actions suivantes :

1. elle stocke le pointeur sur l'objet courant (ici l'instance de Vue) dans une variable `self`, dont l'utilité sera discutée après cette liste ;
2. elle appelle la méthode `axios.get`, avec pour paramètre le service demandé (generation avec la valeur de `nb`) ;
3. une fois la réponse obtenue, un callback est déclenché, et, en fonction du statut de la réponse, la liste est récupérée.

Ici, on remarquera que la référence la notre instance de Vue `this` est stockée dans une variable `self`. On aura donc accès aux data, methods, ..., via cette variable. Il est utile de faire ça ici car, lors de l'exécution du callback, on perdra le référentiel. En pratique, à l'intérieur du callback, on aura accès uniquement aux variables globales et aux variables déclarées à l'intérieur de la fonction appelant le callback. Dans notre cas, à l'intérieur du callback, on aura accès à :

- `reponse` le paramètre du callback ;
- `self` la variable déclarée dans la fonction, et permettant de retrouver tous les attributs et méthodes de notre instance de Vue.