Formation VueJS

Antoine DIEUDONNE/ UTOPIOS

@Utopios Consulting

Sommaire

- Introduction
- Lier nos variables avec Vue
- Un peu de théorie
- La Vue CLI
- Les Composants
- Communiquer entre les composants
- Les Formulaires
- Les Requêtes HTTP
- Le Routing
- La Composition API
- Les Mixins & Composables

Introduction

Qu’est ce que VueJS ?

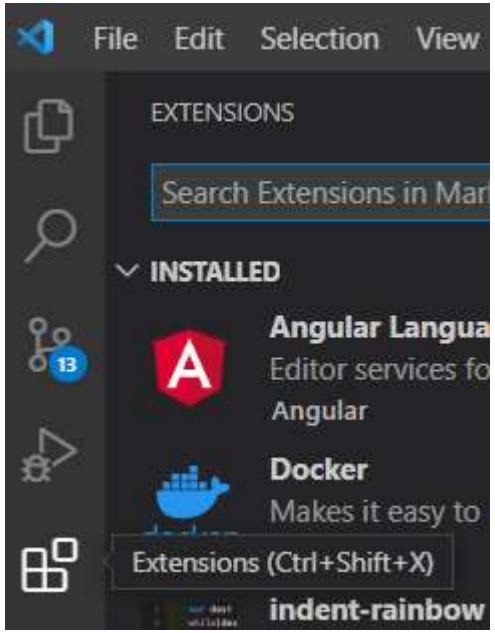
VueJs, ou tout simplement **Vue**, est un **framework front-end** permettant aux développeurs de réaliser des applications front-end (dans le navigateur, du type Google Chrome ou Opera) en se servant de **l’HTML**, du **CSS** et du **Javascript**.

Vue est, à l’instar **d’Angular** et de **React**, l’un des framework phare du web. Contrairement à Angular, il est cependant plus simple d’utilisation. De plus, il utilise le Javascript et non le Typescript. Il permet également de réaliser soit des **application mono-page (SPA)**, soit des **widgets** se trouvant dans des site-webs classiques.

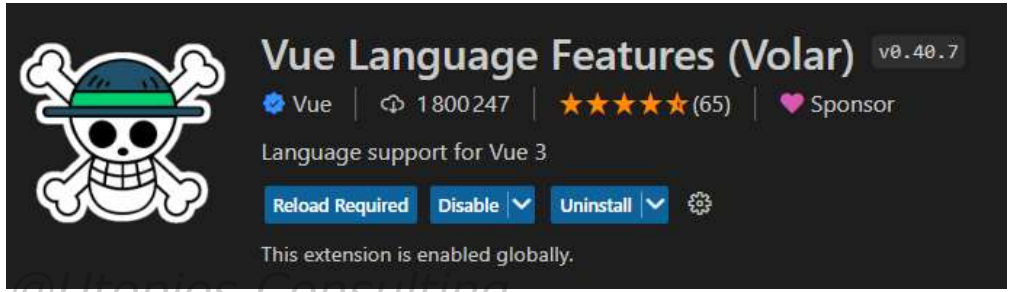
A l’instar de ses cousins, Vue se sert d’un **DOM virtuel** pour rendre la navigation plus réactive et optimiser les rafraichissements de la page dans le but de ne pas perdre les données stockées au niveau de nos variables JS.

Notre environnement de code : VS Code

Durant notre apprentissage de Vue, nous allons utiliser comme environnement de développement le logiciel Visual Studio Code. Très utilisé dans le monde du web, il est l'un des meilleurs éditeur de code du marché à l'heure actuelle. Son lien de téléchargement est disponible à cette adresse : https://code.visualstudio.com/



Pour en tirer pleinement profit, il va également nous falloir y adjoindre quelques extensions. Pour installer une extension, il faut, une fois le logiciel installé et configuré, aller dans l'onglet « Extensions » se trouvant sur le côté gauche de la fenêtre principale. L'extension qui va nous intéresser dans l'utilisation de Vue sera Volar. Pour les plus aventureux, il est également à noter que les extensions Rainbow brackets, Prettier, Indent Rainbow et LiveServer sont très utiles lors de la réalisation de code front-end.



@Utopios Consulting

Notre première application

Nous allons maintenant réaliser notre première application avec VueJS. Cette application sera sous la forme d'un Widget, et pourra se trouver n'importe où dans une page HTML classique. Pour ce faire, nous allons utiliser le **CDN** pour nous offrir l'import de Vue sans avoir à installer quoi que ce soit sur notre ordinateur.

Une fois le CDN en place, il nous est possible de réaliser notre première application. Il nous faudra utiliser un élément HTML qui servira d'élément dans lequel Vue va être utilisé. Ici, ce sera notre ** ** à l'ID « **app** ». Via l'utilisation de la fonction **createApp()** récupérée depuis l'objet Vue, il est possible de définir notre code métier.

Cette fonction prend en paramètre un objet qui possède plusieurs propriétés (ici **data()** est une fonction retournant les différentes données disponibles dans notre application Vue).

Une fois ceci fait, il nous faut « monter » notre application sur l'élément HTML voulu, via la méthode **.mount()**

```
<script src="https://unpkg.com/vue@3"></script>

<div id="app">{{ message }}</div>

<script>
  const { createApp } = Vue

  createApp({
    data() {
      return {
        message: 'Hello Vue!'
      }
    }
  }).mount('#app')
</script>
```

@Utopios Consulting

Lier nos Variables

Le Data Binding

Pour commencer notre apprentissage de Vue, voyons déjà comment faire en sorte d'afficher des valeurs Javascript dans notre page internet. Pour cela, il n'y a rien de plus simple. Il suffit de se servir de la fonction **v-bind** (ou **v-html** pour de l'HTML brut)ou des **chaines de caractères interpolées.**

Dans les deux cas, il va nous falloir avoir une variables retournée par notre propriété **data()** de la sorte. Via ce retour, la variable sera accessible dans tout l'ensemble de notre application Vue, que cela soit dans notre code Javascript ou dans notre HTML via l'une des deux méthodes citées juste avant. La liaison avec **v-bind** va ensuite se présenter de la sorte:

```
createApp({
    data() {
        return {
            dogURL: "
        }
    }
}).mount('#app')
```

```
<img v-bind:src="dogURL" alt="Pho
```

```
<img :src="dogURL" alt=
```

D'un autre côté, si l'on veut afficher dans le corps de notre page des valeurs textuelles, il est possible de la faire via les **strings interpolées.** Si l'on veut que cet affichage ne soit pas rafraichit par la suite, il nous faut utiliser la directive **v-once**

```
<p>
    Mon fils s'appelle {{ children.firstName }} {{ children.lastName }}
</p>
```

```
<p v-once>
    Mon fils s'appelle {{ children.firstName
```

@Utopios Consulting

Le fonctionnement de Vue

Notre application n'est capable de fonctionner que dans l'élément HTML auquel elle est liée. Il est cependant possible de créer plusieurs Widgets fonctionnant indépendamment au sein d'une même page internet. Pour ce faire, il suffit d'avoir plusieurs balises aux identifiants uniques (pensez aux Ids) dans lesquels le script.

A côté de cela, les variables stockées dans le retour de notre propriété **data()** seront utilisables seulement dans l'élément HTML dans lequel l'application est montée. Il est donc important de noter que les méthodes de liaisons de variables que l'on a vu ne seront possible qu'au sein d'une application montée.

```
<div>
    Mon fils s'appelle {{ children.firstName }} {{children.lastName }}
</div>

<div id="app">
```

Mon fils s'appelle {{ children.firstName }} {{children.lastName }}

Les Méthodes

Bien entendu, il n'est uniquement possible de passer des informations dans le sens **Javascript -> HTML**. Il sera aussi possible de faire des modifications de notre code métier depuis l'interface construite par le code HTML. Pour ce faire, il nous faut utiliser une autre propriété de notre objet passé dans **createApp** : la propriété **methods**. Cette propriété va prendre en valeur un objet contenant plusieurs fonctions, qu'il sera possible d'appeler depuis notre HTML via les évènements du DOM. La propriété **methods** sert également à stocker des fonctionnalités dans notre code métier. C'est à cet emplacement que l'on peut par exemple créer une fonction de lancement de dés :

```
methods:{
    rollDice(minValue, maxValue) {
        return Math.random() * (maxValue - minValue) + minValue
    }
}
```

Il nous est désormais possible d'afficher le retour de cette méthode dans notre code HTML via l'utilisation de chaines de caractères interpolées (ces chaines peuvent également servir à effectuer du code métier JS simple, par exemple des ternaires)

```
Résultat du dé : <b> {{ rollDice(1, 6) }} </b>
```

Accéder à nos propriétés dans les méthodes

Pour pouvoir manipuler nos valeurs stockées dans la propriété **data** de notre application, il nous faut utiliser une syntaxe particulière. Par exemple, faisons en sorte que le résultat de notre dé soit stocké dans une variable de type texte :

```
},
diceValue: 'Great !'
```

En sortie du résultat de la méthode **rollDice**(), il nous est possible de modifier la valeur de cette propriété via l'utilisation d'un bloc **switch**()

```
switch (result) {
    case 1:
        this.diceValue = "Bad roll !"
    case 2:
        this.diceValue = "Could be better."
```

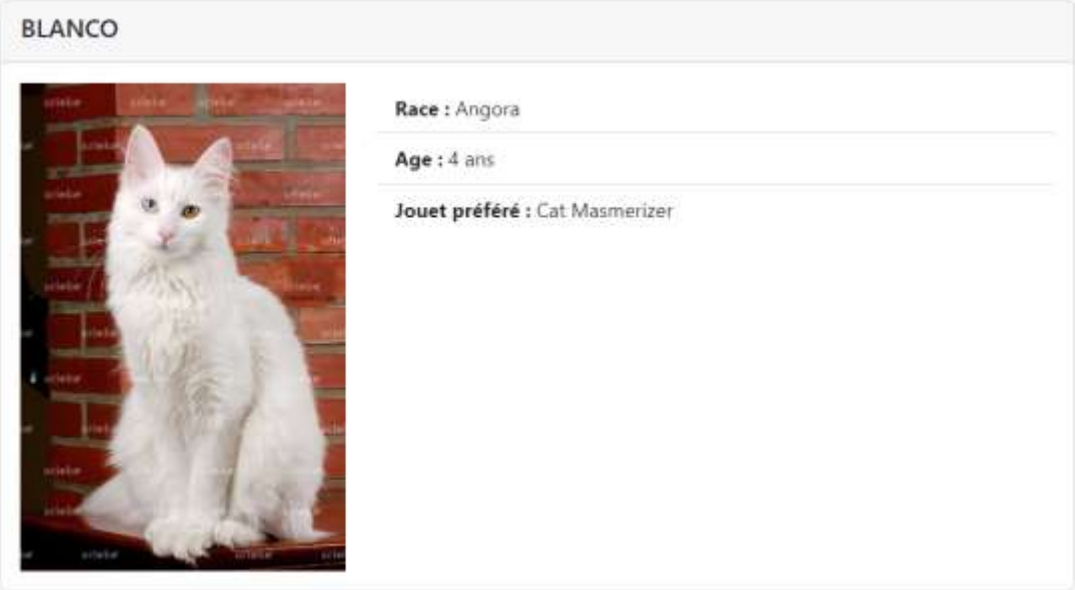
On remarquera que la syntaxe oblige l'utilisation du mot-clé **this** pour accéder aux propriétés de notre application (**this** va ici référencer notre application, et non pas la fenêtre du navigateur comme cela pourrait être le cas dans des fonctions JS standard)

Liaison de Données

EXERCICE

En vous servant du framework Vue, vous devrez, dans une page internet, afficher une carte de visite d'un animal. Cette carte de visite devra être alimentée par des valeurs stockées dans le script de code Javascript et proposer un design un minimum travaillé.

Vous devrez ainsi alimenter le nom, la race, le portrait, l'âge et
l'un des jouet contenu dans la liste des jouets préférés de l'animal
en question, qui pourraient par exemple être contenu dans un objet



Nous lier à nos évènements

Désormais, nous allons voir comment lier nos méthodes à des évènements du DOM, pour pouvoir déclencher des fonctionnements de notre code métier lors du clic sur un bouton par exemple.

Pour nous lier à des évènements, il va nous falloir cette fois-ci utiliser la directive **v-on:<event>** pour déclencher une méthode donnée lors de l'évènement spécifié après les deux points de notre directive :

```
v>
  <button v-on:click="rollDice(1,6)">Lancer un dé
```

```
v>
  <button @click="rollDice(1,6)">
iv>
```

Si l'on le veut, il est possible de faire passer l'objet d'évènement à notre méthode dans le but de pouvoir le manipuler dans notre code métier. Pour ce faire, il faut ajouter un paramètre à notre méthode : **$event:**

```
<button @click="rollDice($event, 1,6)">Lancer un
  <input type="text" @input="setName($event)">
```

```
setName(event) {
    this.children.firstName = event.target.value
},
```

@Utopios Consulting

Les Modificateurs d'évènements

En plus de pouvoir nous lier à des évènements, il nous est possible d'y ajouter des modificateurs. Par exemple, pour empêcher le rafraichissement automatique de notre page en cas d'envoi d'un formulaire, on peut utiliser le modificateur **prevent** de la sorte :

```
<form @submit.prevent="handle
```

D'autres modificateurs plutôt utiles peuvent être les modificateurs **key** dans le cas d'un évènement de type **keydown** par exemple. De la sorte, on peut déclencher une fonction particulière avec une touche du clavier, par exemple F, et une autre fonction avec la touche ENTER :

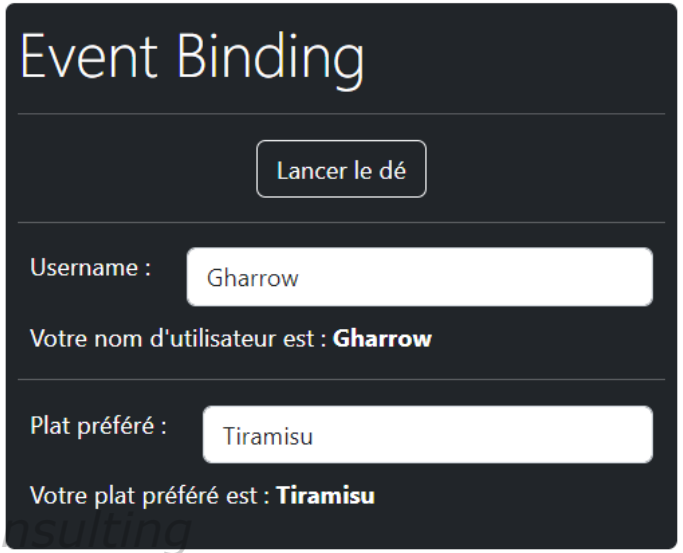
```
' @keyup.f="rollDice(1, 6)" @keyup.enter="setName($event)">
```

Manip’ des Events

EXERCICE

En vous servant du Framework Vue, vous devrez, dans une page Internet, proposer :

- Après l’appui sur une bouton, l’affichage d’une alerte contenant un message aléatoire indiquant le résultat de lancement d’un dé à 6 faces.
- Un input qui va être relié à une zone de votre page pour afficher en direct le contenu de votre input. Par exemple « Votre nom d’utilisateur est <valeur de l’input> »
- Un input qui va permettre, lors de l’appui sur la touche entrée pour le valider, d’afficher son contenu dans un emplacement de votre page internet.



@Utopios Consulting

Le Two-Way Data-Binding

Si l'on veut désormais pouvoir à la fois afficher le contenu de notre code métier ainsi que de pouvoir modifier les valeurs de nos variables dans le Javascript, il va nous falloir utiliser à la fois les directives **v-bind** et **v-on.**

```
<input type="text" :value="children.firstName" @input="setName($event)">
```

Une autre façon de faire est d'utiliser ce qui s'appelle le **Two-Way Data Binding**, via l'utilisation de la directive **v-model**. Cette directive se présente de la sorte :

```
<input type="text" v-model="children.firstName">
div>
```

Via l'utilisation de cette directive, la variable sera automatiquement liée à notre input. C'est-à-dire qu'elle sera affichée et son rafraichissement se verra modifié en cas de changement de la valeur dans le code Javascript. D'un autre côté, si l'on modifie la valeur de la variable dans l'input, alors la variable sera modifiée dans notre code Javascript, ce qui rafraichira également les emplacement où elle pourrait être visionnée dans d'autres parties de notre page.

Les Computed Properties

Imaginons par exemple que nous aurions trois variables : **firstName**, **lastName** et **fullName**. fullName contient la concaténation des deux variables précédentes. Malheureusement, le rendu de fullName par binding ne sera pas rafraichit en cas de modification de firstName ou de lastName, car Vue fonctionne par principe dans une logique de rafraichissement des valeurs et non pas de leurs dépendances.

```
computed: {
    fullName() {
        return this.children.firstName + ' ' + this.children.lastName
    }
}
```

Pour réaliser ce cas de figure, on a donc tendance à utiliser des **computed properties**. A côté des propriétés se trouvant dans la propriété **data** de notre **createApp()** existent ce que l'on appelle les **computed properties.** Ces propriétés doivent être ajoutées dans la propriété **computed,** tout comme **data** ou **methods**.

Ce nouvel objet va contenir des méthodes qui devront absolument avoir un retour. Ces méthodes pourrons être utilisées comme des propriétés classiques, par exemple dans une **string interpolation**.

```
<p v-once>
    Mon fils s'appelle {{ fullName }}
</p>
```

Les Watchers

Une autre feature possible de Vue est l'utilisation des **watchers.** Cette feature est semblable aux **computed properties** en cela que l'on peut demander à Vue de vérifier les dépendances dans le but de rafraichir une propriété. La différence se situe dans le fait que pour qu'un watcher fonctionne, il faut que la propriété de base soit existante (dans les propriétés de **data**). Les watchers seront exécutés à chaque fois que l'on modifie une variable portant le même nom qu'eux. Par défaut, un watcher va automatiquement récupérer la nouvelle valeur en paramètre.

Par exemple, on peut demander de vérifier la propriété **firstName**.

```
watch: {
    firstName(value) {
        this.fullName = value + ' ' + this.lastName
    }
},
```

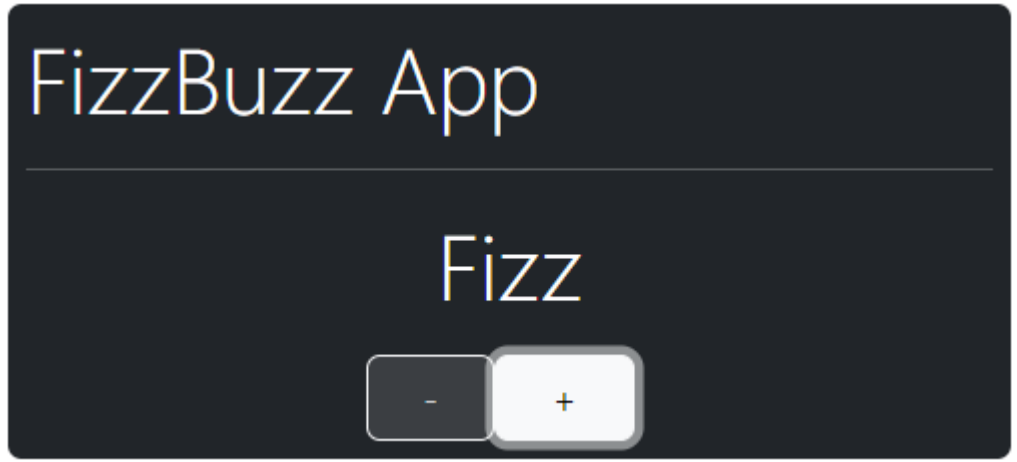
FizzBuzz’

EXERCICE

En vous servant du Framework Vue, vous devrez, dans une page Internet, réaliser un display (via utilisation d’une propriété calculée) qui va afficher un message personnalisé en fonction de si le nombre d’un compteur est un multiple de 3, de 5 ou de 15 :

- Dans la cas où le nombre est un multiple de 3, vous devrez afficher Fizz
- Dans le cas où le nombre est un multiple de 5, vous devrez afficher Buzz
- Dans le cas où le nombre est un multiple de 15, vous devrez afficher FizzBuzz
- Dans le cas où aucune des condition préalable n’est rencontrée, alors vous affichez simplement la valeur du compteur

Via l’utilisation d’un watcher, vous ferez en sorte que le compteur se réinitialise après 15 secondes.



@Utopios Consulting

Styliser notre application

Maintenant que l'on a vu les différentes façon d'afficher ou de manipuler les valeurs de notre Javascript dans l'HTML, il pourrait être intéressant de voir comment améliorer le design global de notre application.

Pour commencer, nous allons voir comment faire pour changer le style de notre widget avec la directive v-bind fixée sur la propriété **style** ou sur **class**. Pour les deux cas, il est possible de lier la chose à un objet Javascript pour pouvoir rendre dynamique les valeurs de notre style ou la présence et l'absence de classes dans notre élément HTML.

```
<form @submit.prevent="handleForm($event)"></form>
<p :style="{backgroundColor: isAdult ? 'green':'transparent'}">
```

```
<span :class="{'adult': isAdult}" >I
```

Boite Magique

EXERCICE

En vous servant du Framework Vue, vous devrez, dans une page Internet, styliser de façon dynamique une boite. Le contenu de cette boite pourra changer de couleur en fonction de l'entrée utilisateur (en anglais) dans un input de type texte. Pour ce faire, vous vous lierez à l'attribut **style** du contenu de la boite.

De plus, en vous servant d'un élément de type <select> vous rendrez le contenu de la boite dynamique (vous pourrez par exemple y afficher un carré, un cercle ou un rectangle). Pour ce faire, vous vous lierez au contenu de la boite, en particulier à son attribut **class**.

Le Rendu Conditionnel

Afficher ou masquer du contenu

Lorsque l'on veut, il est possible de faire en sorte que le contenu de notre page soit ou non masqué. Pour ce faire, il existe deux méthode.

La première est l'utilisation de la directive v-show. Cette directive va prendre un paramètre un booléen et va afficher le contenu de l'élément HTML dans la page en cas de valeur vraie. Dans le cas contraire, alors la propriété CSS **display: hidden** sera appliquée à notre élément.

```
<p v-show="isAdult">
    L'enfant est un adulte !
</p>
```

Cette méthodologie n'est bien entendu pas optimale, car des éléments pourraient être lus en analysant le code source de la page. L'autre option est donc d'utiliser les directives **v-if**, **v-if-else** et **v-else**. Par utilisation de ces trois directives, il est possible de rendre conditionnel les éléments HTML mais également de les faire disparaitre du flux de la page en cas de condition fausse. Ces trois directives doivent se suivre en cas d'utilisation commune sous peine de poser problèmes (les éléments HTML doivent être au même niveau et leurs balises se succéder)

```
<p v-if="age >= 21">L'enfant est majeur aux USA</p>
<p v-else-if="age >= 18">L'enfant est majeur en France</p>
<p v-else>L'enfant bien mineur</p>
```

@Utopios Consulting

Utiliser la directive v-for

Si maintenant nous voulions pouvoir alimenter une série de balises <li> de données présentent dans notre code métier sous la forme d'un tableau, comment ferions-nous ? La solution est assez simple, et consiste en l'utilisation de la directive **v-for**. Dans cette directive, il est possible de générer un élément basé sur l'itération d'une variable conteneur comme un tableau.

```
<p>Liste des jouets de l'enfant :</p>
<ul>
    <li v-for="t in toys">{{ t }}</li>
</ul>
```

Aller plus loin avec v-for

Dans le cas où nous serions amené à afficher l'index des éléments en même temps que leur valeur, il est possible d'obtenir ce dernier en se servant d'une notation légèrement altérée. De même, dans le cas d'un objet, la récupération de l'ensemble clé-valeur est disponible par la même syntaxe :

```
<li v-for="(k, v) in {name: 'Hello', age: 15}"><b>{{ k }}:</b> {{ v }}</li>
```

```
<li v-for="(t, i) in toys">{{ i }} - {{ t }}</li>
```

A côté de cela, il faut faire attention, car la modification d'une liste va causer la création ou la suppression d'un élément de notre DOM. Cependant, il faut être prudent, car par défaut Vue va faire en sorte d'optimiser ce fonctionnement, et pourra par exemple réutiliser un élément similaire au lieu de remplacer entièrement l'élément. Pour éviter ce problème, il convient d'ajouter l'attribut **key** à notre élément de sorte à rendre cette partie de notre HTML réellement unique.

```
<li v-for="(t, i) in toys" :key="i">{{
```

Jeu de Rôle

TP

En vous servant du Framework Vue, vous devrez réaliser une application permettant à un joueur de faire un combat de jeu de rôle d'un héros contre une créature. Votre application devra permettre plusieurs actions pour le joueur et pour le monstre :

- Attaquer : Permet d'infliger des dégâts à l'adversaire en se basant sur le lancé d'un dés de dégâts et sur sa classe d'armure (sa défense). Les dégâts seront au minimum de 1
- Coup spécial : Permet d'infliger des dégâts à l'adversaire basés sur le lancé d'un dés de dégâts multiplié par une valeur spéciale. Encore une fois, les dégâts seront au minimum de 1
- Soigner : Permet à l'utilisateur de recouvrir de la vie en se basant sur un lancé de dés de soin. Ce soin aura un coup en point de magie
- Garde : Permet à l'utilisateur de prévenir une quantité de dégâts supplémentaires en se basant sur un dés de garde. Le coup spécial aura également un coup en points de magie

Le monstre ainsi que le joueur pourront à chaque tour de jeu effectuer l'une de ces actions (celles du monstre seront choisies aléatoirement).

L'affichage de l'application devra permettre de visualiser en direct les changement sur les points de vie ou de magie ainsi que de suivre de façon textuelle le déroulé du combat dans un log permettant de voir les 10 dernières entrées textuelles.

@Utopios Consulting

Comment fonctionne Vue ?

La Réactivité de Vue

Vis-à-vis des propriétés que l'on défini dans une application Vue, le framework se charge de garder une trace de leurs changements potentiels. L'objet de données **data** est transformé en un objet de **Proxy** Javascript, permettant ainsi d'obtenir un rafraichissement des valeurs dans la page en cas de changement de la propriété.

Il est possible de simuler le fonctionnement de Vue nous-même en réalisant l'objet de Proxy de la sorte:

```
const data = {
    message: "Hello",
    longMessage: "Hello Mark"
}

const handler = {
    set(target, key, value) {
        console.log(target) // Ici, ce sera l'objet de données
        console.log(key) // Ici, le nom de la propriété qui va être changée
        console.log(value) // Ici, la valeur qui sera affectée à la propriété

        if (key === 'message') {
            target.longMessage = value + " Mark"
            target.message = value
        }
    }
}

const proxy = new Proxy(data, handler)

proxy.message = "Bye"
```

@Utopios Consulti

Plusieurs widgets avec Vue

Il est tout à fait possible d'avoir plusieurs widgets avec Vue. Pour ce faire, il suffit d'avoir plusieurs éléments HTML dans lesquels on peut monter une application Vue. Attention cependant, chaque application possède son propre objet résultant de **createApp,** et donc chaque application aura ses propriétés et ses méthodes qui ne seront pas disponibles dans les autres applications.

Lorsque nous verrons les **SPA (Single Page Application)**, nous verrons qu'il est possible de réaliser plusieurs sous-ensembles qui eux seront capable de communiquer entre eux : les **composants**.

Les Templates

Jusqu'à maintenant, nous avons utilisé les templates sans le savoir. En réalité, le fait de monter notre application Vue sur un élément de notre HTML correspond à fixer la template de notre application à ce que contient cet élément HTML. Il est tout à fait possible cependant de créer l'HTML de notre application non pas dans l'HTML en soit, mais dans la propriété **template** de notre objet de création d'application.

```
createApp({
    template: `
        <p>Je suis le template de l'application <b>{{ appName }}</b></p>
    `,
```

De la sorte, l'élément HTML dans laquelle on a monté notre application contiendra le template que l'on aura fixé. Cette technique n'est cependant pas souvent utilisée car elle réduit grandement les capacités de notre éditeur de code en plus de rendre potentiellement confuse la limite entre l'HTML et le JS.

@Utopios Consulting

Les Références

Il est possible, via l'utilisation des **références**, de récupérer les éléments HTML de notre application Vue à la volée en cas de besoin. Par exemple, il n'est pas obligatoire, et c'est d'ailleurs moins optimisé, d'utiliser les différentes directives vues précédemment (**v-bind**, **v-on** et **v-model**).

Par exemple, pour récupérer la valeur d'un input au moment de l'envoi d'un formulaire, on pourrait se servir de la référence de l'input, qui permettrait d'accéder à son attribut **value** seulement au moment du clic sur le bouton « envoi » du formulaire.

```
<input type="text" placeholder="Surname" ref="surnameInput">
```

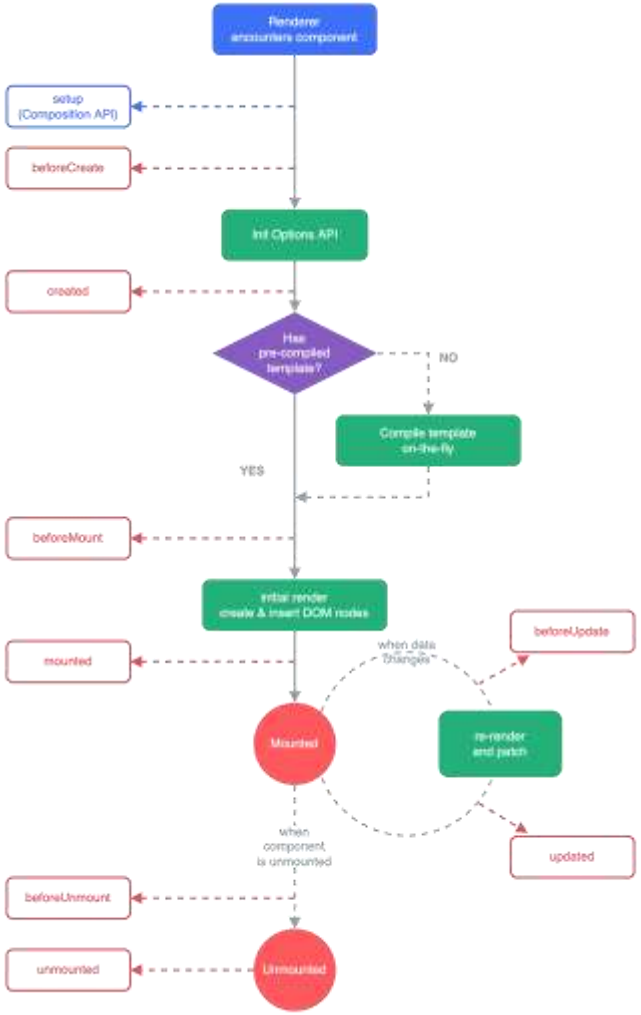
Pour créer une référence au niveau d'un élément HTML, Vue propose l'attribut simple et explicite **ref** qui permet de nommer l'élément HTML vis-à-vis de notre application. Une fois nommé, son accès se fait via la syntaxe **this.$refs.nomReference**, comme ci-dessous :

```
methods:{
    submitForm() {
        let surname = this.$refs.surnameInput.value
    }
},
```

@Utopios Consulting

Les Cycles de Vie

Vue fonctionne avec un système d'étapes lors de la création et de la manipulation d'une application. Ces différentes étapes peuvent être utilisées pour exécuter du code à des moments clés. Il est également nécessaire de les connaître pour savoir quelles variables, éléments du DOM ou valeurs sont disponibles à tel instant de la vie de l'application.



Pour se servir des **hooks** (méthodes exécutées à des moments clés du cycle de vie de l'application), il suffit de définir la méthode correspondante dans l'objet de création de l'application. Par exemple, pour exécuter du code avant la création de l'application, nous pourrions procéder de la sorte :

```
  },
  beforeCreate() {
      console.log("Application will be created! ")
  }
).mount('#app')
```

A quoi servent les composants ?

Contrairement à notre fonctionnement précédent, nous allons désormais migrer progressivement vers la création d'applications composées de plusieurs composants. Contrairement à de multiples applications (widgets) incapables de communiquer entre eux, une application composée de plusieurs composants est capable de proposer un rendu d'éléments dans lesquels **les données peuvent transiter et communiquer entre les éléments**.

Par exemple, une liste de clients peuvent être affichées de la même manière et proposer les mêmes fonctionnalités car cette liste n'est en réalité qu'une **liste de composants** clients dans lesquels on envoie chacun de nos clients. De la sorte, il est possible de proposer les même fonctionnalités pour chaque élément HTML final.

Notre premier composant

Pour créer un composant, il suffit de procéder de la même façon dont nous créerions une application, à ceci près que nous allons utiliser la méthode **.component()** sur notre variable d'application **app.**

Le premier paramètre qu'il faudra donner à cette méthode sera l'identifiant de notre composant (le nom de la balise HTML qu'il nous faudra utiliser à chaque fois que l'on voudra proposer l'ajout de notre composant à un emplacement donné). Il est important que ce nom soit un nom composé afin de le différencier des éléments HTML classiques.

Ensuite, il nous faudra lui donner un objet de configuration, de la même façon que pour notre application. Cet objet de configuration proposera également les propriétés **data**, **methods**, **computed**, **watchers**, etc…

Pour l'ajouter à notre application, il suffit de demander le composant dans son template en se servant de la balise HTML que nous avons configuré :

```
app.component('client-component', {
    template: `
        <div>
            {{client.firstName}} {{client.lastName}}
            <ul>
                <li v-for="job in client.jobs">{{job}}</li>
            </ul>
        </div>
    `,
    data() {
        return {
            client: {
                firstName: "John",
                lastName: "MARTIN",
                jobs: ['Scientist', 'Teacher', 'Researcher']
            }
        }
    }
})
```

```
<div id="app">
    <client-component></client-component>
</div>
```

@Utopios Consulting

Vue CLI

Vérifier notre installation de NodeJS

Au vu de l'augmentation future de la complexité de nos applications, nous allons désormais utiliser ce qui s'appelle la **Vue CLI.** Cette interface en ligne de commande va nous être utile pour la création d'application de type SPA plus poussées dans lesquels nos composants pourrons être disponible séparément dans leur propre fichier. Chaque fichier **.Vue** va être composé de plusieurs sections : HTML, JS et CSS.

Pour pouvoir utiliser la Vue CLI, il faut nous assurer que nous possédons **NodeJS.** Pour en tester l'installation et vérifier sa version, il suffit d'entrer dans un invité de commande la ligne ci-dessous :

```
PS C:\Users\gharr> node -v
v16.17.0
```

Si l'on obtient en résultat de cette commande une version de Node, c'est que l'installation est correcte. Dans le cas contraire il nous faut l'installer en allant au lien ci-après : https://nodejs.org/en/download/

Il est préférable de prendre la version **Long Term Support (LTS)** afin d'éviter tout problème éventuel. L'installation de modules additionnels comme **Chocolatey** peut aider à l'avenir à conserver notre installation de Node à jour.

Installer Vue CLI

Maintenant que nous avons installé NodeJS et vérifié sa version, il nous faut procéder à l'installation de Vue CLI. Pour l'obtenir, nous allons passer par NodeJS. La commande à entrer est la suivante :

```
> npm install --global @vue/cli
```

Une fois installée la CLI de Vue va nous permettre de créer une application directement avec la ligne de commande dédiée (nous prendrons, lorsque l'on nous le demandera, le choix de réaliser une application Vue 3):

```
r> vue create my-project
```

Enfin, pour pouvoir visualiser notre application, nous allons devoir par la suite utiliser un serveur de développement, qui va être créé par l'utilisation de la commande :

```
> cd my-app
\my-app> npm run serve
```

Ce serveur va offrir un fonctionnement similaire à Live Server en cela qu'il va permettre un rafraichissement automatique de notre page en cas de changement dans notre code. Pour les possesseurs de plusieurs moniteurs, cette fonctionnalité arrivera donc à point nommé ! De plus, nous serrons de la sorte plus proche de l'expérience utilisateur finale via l'utilisation du protocole HTTP pour la récupération de la page web au lieu du protocole de fichiers.

@Utopios Consulting

La Structure du projet

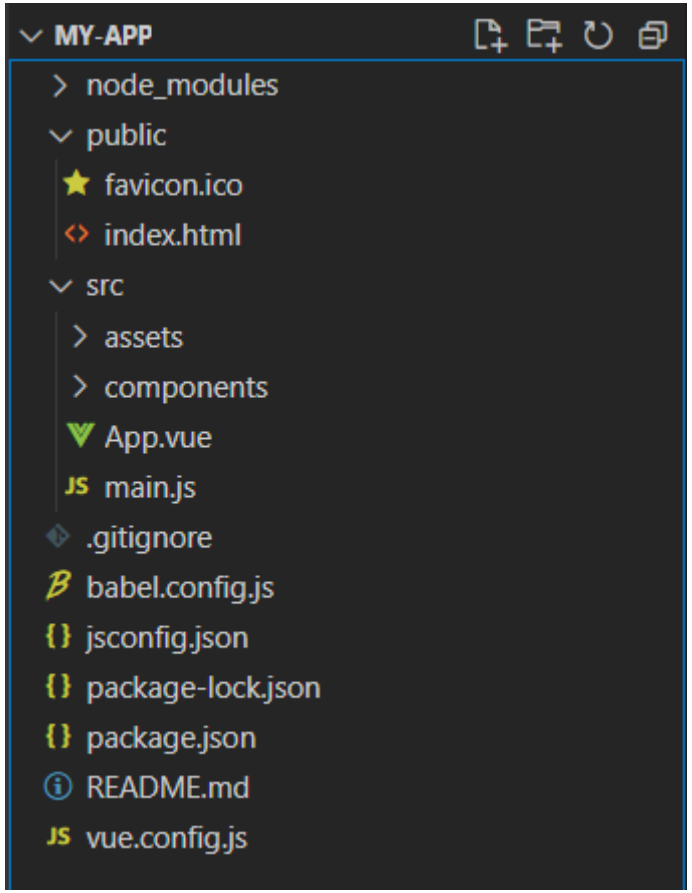
Notre application générée par Vue possède de nombreux fichiers que nous allons désormais expliquer afin d'éviter toute confusion ou incompréhension.

Pour commencer, le dossier **node_modules** est un dossier qui va contenir toutes les dépendances (les packages Javascript) permettant à notre application de fonctionner.

Le dossier **public** va de son côté concerner les fichiers qui seront envoyés par le serveur de développement à notre navigateur. On y voit par exemple le fichier **index.html** qui sera la seule et unique page utilisée par notre navigateur pour naviguer dans notre application. Le changement de page se fera via le Javascript, qui se contentera de modifier le DOM présent dans cette unique page.

Le dossier **src** sera le dossier central de notre application. C'est à cet emplacement que nous créerons les fonctionnalités, les templates et tous les éléments de notre application, qui seront par la suite transformé pour être envoyé au navigateur sous la forme de librairies Javascript.

Au niveau des fichiers, ceux qui peuvent nous intéresser sont les fichiers **package.json** qui contient sous la forme d'un JSON toutes les dépendances (les différents packages se trouvant dans le dossier node_module) de notre application. Lorsque l'on effectue la commande **npm install** à la racine de notre application, Node se sert de ce fichier pour constituer le dossier **node_modules**



Les fichiers .Vue

Désormais, nous allons travailler au moyen de fichiers portant l'extension **.vue** De tels fichiers se présentent sous la forme de trois grandes balises qui sont :

- **template** : Cette balise va contenir l'ensemble de l'HTML de notre fichier. A la façon dont nous faisions des applications de type widget, cette section va servir de template (d'où son nom !) à notre code métier pour que Vue puisse interagir avec l'HTML après montage du composant ou de l'application entière.
- **script:** Cette section va correspondre à l'objet de création d'application ou de composant que nous utilisions précédemment. Dans les grandes lignes, il va nous falloir un objet exporté par défaut afin de pouvoir l'importer dans les autres parties de notre application.
- **style:** Cette section va tout simplement contenir l'ensemble de notre CSS pour modifier le style de notre application.

Derrière, Vue va transformer ce fichier .vue pour le rendre compatible avec notre navigateur. La résultante sera un fichier Javascript standard qui lui sera lisible par le navigateur web.

Première application SPA

Pour notre première **Single-Page Application**, nous allons montrer comment faire une simple page. Pour ce faire, il va nous falloir un fichier portant l'extension .vue qui contiendra les mêmes éléments que ce que nous avons vu jusqu'à présent. Nous retrouvons donc les propriétés de l'objet de création, qui est l'objet exporté par défaut dans le fichier actuel. Cet export sera récupéré au niveau de notre fichier **main.js** dans le code ci-dessous :

```
main.js
import { createApp } from 'vue'

import App from './App.vue'

createApp(App).mount('#app')
```

```
  <div class="dog-cards">
    <div v-for="(dog, index) in dogs" class="dog-card" :key="index">
      <div class="dog-picture">
        <img src="..." alt="...">
      </div>
      <div class="dog-infos">
        <ul>
          <li><b>Nom:</b> {{dog.name}} </li>
          <li><b>Race:</b> {{dog.breed}} </li>
          <li><b>Age:</b> {{dog.age}} </li>
        </ul>
      </div>
    </div>
  </div>
</template>

<script>
export default {
  data() {
    return {
      dogs: [
        {
          name: "Bernie",
          breed: "Labrador",
          age: 7
        },
        {
          name: "Rex",
          breed: "Doberman",
          age: 4
        }
      ]
```

Manipuler les Composants

Ajouter un composant

Il est préférable, lorsque l'on travaille avec des composants, de les regrouper dans un dossier dédié. Nous allons donc créer un dossier **components** dans notre dossier **src** pour y stocker nos futurs composants.

```
<script>
    export default {
        name: 'dog-card',
        data() {
            return {
                dog: {
```

Le composant que nous allons réaliser sera un élément HTML que l'on pourrait vouloir réutiliser, comme par exemple la carte de notre chien. Ce composant sera amené à être multiplié via une boucle **v-for** par la suite. Dans notre structure de fichiers, ce composant va se traduire par un nouveau fichier .vue qui va comporter également les sections **template**, **script** et potentiellement **style**. Notre composant va devoir être enregistré au niveau de notre application via l'utilisation de sa propriété **components**. Cette propriété va être alimentée par un objet contenant les différents imports :

```
},
components: {
  DogCard
}
```

Création de Composants

EXERCICE

En vous servant du Framework Vue, vous devrez réaliser une application constituée de plusieurs composants.

Ces composants seront des **alert** de couleurs et de contenu différents et seront rendus les uns au dessus des autres via l'utilisation de leur sélecteurs propres.

Afin de pratiquer la répétition permise par l'utilisation des composants, vous ferez en sorte de placer :

- Une alerte de type success
- Deux de type danger
- Une de type success
- Trois de type warning ou info

Cet succession se fera dans un arrangement vertical et au sein d'une même page (par exemple la page racine).

Communication Parent -> Enfant

Bien entendu, nos composants doivent être capable d'offrir un transit des données du parent vers l'enfant afin de pouvoir alimenter à la volée nos composants en données. Pour le faire, nous avons recourt à l'utilisation des **props.** Cette propriété de l'objet de création de notre composant sert à définir des variables qui seront alimentées par le parent au moyen d'attributs personnalisés.

```
<test-component
first-name="Bernard"
last-name="DUPONT"></test-component>
```

```
<script>
    export default {
        name: 'test-component',
        props: ['firstName', 'lastName']
    }
</script>
```

Cette propriété se présente de la sorte. Chaque propriété récupérée doit simplement spécifier sous la forme d'un tableau le nom des propriétés qu'elle va réceptionner. Une fois alimentée, les props fonctionne comme les propriétés standards de notre application (elles sont accessibles dans le code métier via le mot-clé **this** et dans le template via leur nom en **camelCase**)

Validation des Props

Il est possible de pousser la chose plus loin, en transformant ce tableau de strings en un objet où chaque propriété spécifiera le nom de la **prop** ainsi que des informations supplémentaires sur les propriétés passées par le parent au composant.

```
props: {
    firstName: {
        type: String,
        required: true
    },
    lastName: {
        type: String,
        required: false,
        default: "LastName"
    }
}
```

Par exemple, il est possible de spécifier le type qui doit être fourni à l'attribut HTML, de forcer l'ajout d'une propriété sous peine de lever une erreur, ou au contraire de donner une valeur par défaut à la propriété en cas de non injection.

Si l'on le veut, on peut même réaliser des validateurs personnalisés via l'utilisation d'une fonction **validator()** qui va prendre en paramètre la valeur et retourner un booléen pour valider ou non la valeur envoyée par le parent à son enfant :

```
validator: value => {
    return value.length > 0
}
```

Alimenter dynamiquement les Props

Il est bien entendu possible d'alimenter nos propriétés de façon dynamique via l'utilisation des directives que nous avons vu précédemment. Ici, la directive **v-bind** va nous apporter l'aide dont nous avons besoin.

Par exemple, donnons à un composant les informations dont il a besoin à partir d'un tableau d'objets ci-contre:

```
export default {
  name: 'App',
  data() {
    return {
      appName: "Application Vue",
      dogs: [
        {
          name: "Bernie",
          breed: "Labrador",
          age: 7
        },
        {
          name: "Rex",
          breed: "Doberman",
          age: 4
        }
      ]
```

Nous obtenons ainsi une série de composants dont les props sont alimentées à la volée par l'itération sur notre tableau contenu au niveau des propriétés du parent.

Attention: Il ne faut pas oublier l'utilisation de la clé lors de l'utilisation de composants afin de les différencier !

```
<dog-card v-for="dog, index in dogs"
:key="index"
:dogName="dog.name"
:dogBreed="dog.breed"
:dogAge="dog.age"></dog-card>
```

Communication Enfant -> Parent

Passons maintenant à l'autre côté du miroir. Faisons communiquer nos enfants vers les parents en utilisation la liaison sur les évènements.

Pour ce faire, il va nous falloir émettre nos propres évènements dans les composants enfants et réceptionner ces évènements au niveau du parent avec la directive **v-on.**

Pour émettre nos propres évènement, il va falloir utiliser la méthode fournie par Vue qui se trouve être **this.$emit()** et qui sera déclenchée par une méthode de notre composant (cette méthode pourra être elle-même déclenchée par un évènement au sein du composant, comme par exemple le clic d'un bouton) :

```
methods: {
    adoptDog() {
        this.$emit('dog-adopted', this.dogID)
    }
}
```

```
<button @click="adoptDog">Adopter</button>
```

Cette méthode va avoir comme argument le nom de l'évènement, et en paramètres supplémentaire des données qui seront envoyées en même temps et pourrons être réceptionnées par le parent :

```
methods: {
  handleDogAdoption(dogID) {
    console.log(`Dog with ID ${dogID} adopted`)
  }
}
```

```
@dog-adopted="handleDogAdoption($event)"></dog-card>
```

@Utopios Consulting

Définir et Valider nos Evènements

Tout comme pour les **props**, il est possible d'utiliser une variante pour augmenter les fonctionnalités de nos évènements. Pour réaliser ceci, il nous faut créer une propriété dans notre objet exporté qui se nommera **emits.**

```
emits: ['dog-adopted'],
```

Cette propriété va être alors soit remplie d'un tableau de strings correspondant aux différents évènements émis par le composant, soit par des objets offrant plus de contrôle sur les évènements.

```
emits: {
    'dog-adopted': function(dogId) {
        if (dogId) return true
        else {
            console.warn("Dog ID is missing !")
            return false
        }
    }
},
```

@Utopios Consulting

Provide et Inject

Dans le cadre d'application de plus grande envergure, il est commun de se retrouver à devoir faire passer les données ou les évènements de X parents vers / depuis X enfants. Malheureusement, ce mode de fonctionnement pose des problèmes lors de nombreuses données car il nous faut être sur de faire bien transiter les éléments partout ainsi que d'éviter les fautes typographiques.

Heureusement pour nous, il existe une autre façon de faire. Il est possible d'utiliser la méthode se servant de **Provide** et **Inject.** Ces deux éléments sont en réalité des propriétés que l'on va ajouter à notre objet. **provide** permet d'informer Vue que les données vont être par la suite récupérées au niveau d'une propriété de type **inject**.

```
provide() {
  return {
    dogs: this.dogs
  }
},
```

```
inject: ['dogs'],
```

Cette propriété fonctionne de la même façon que la propriété **props** dont nous nous servions précédemment. Attention cependant, cette mécanique ne fonctionne qu'au sein d'éléments se trouvant dans un lien de parenté direct (il n'est donc pas possible de le faire entre composants voisins). Le parent va donc posséder une propriété **provide** qui va retourner un objet tout comme le faisait la propriété **data** et le descendant va avoir une propriété inject ayant pour valeur un table des éléments qu'il va récolter depuis son aïeul.

Et pour les Evènements ?

Une méthode similaire est disponible lors de l'utilisation d'évènements. En pratique, ce mécanisme se présente par le passage d'une fonction depuis le parent via son utilisation de **provide** qui va être activée et déclenchée par le descendant (ce dernier y accédant via sa propriété **inject**

De la sorte, on tire profit du fait qu'en Javascript, il est possible de passer des fonctions entières en tant que valeur de variable.

Carnet de Recettes

TP

En vous servant du Framework Vue, vous devrez réaliser une application constituée de plusieurs composants.

Votre composant principal sera un carnet vous offrant la possibilité de visionner les recettes de cuisines que vous aurez créés. Dans ce composant principal se trouveront deux composants :

- Le premier permettant, par l'utilisation d'un formulaire, d'entrer les informations des recettes. Par exemple, un titre et un texte. Une fois ces informations recueillies, il faudra inclure la recette dans la collection de recettes de son parent par l'appui sur un bouton se trouvant dans ce composant enfant
- L'autre composant sera un composant destiné à la présentation desdites recettes et sera répété par son parent pour chaque recette qu'il a besoin de présenter à l'utilisateur (N'oubliez pas d'utiliser les directives, par exemple **v-for**)

Via l'utilisation de ces trois composants, vous réaliserez une interface permettant une présentation agréable de plusieurs recettes au sein d'une même page, et une section destinée à l'ajout des recettes en cas de besoin.

Compteur pair / impair

EXERCICE

En vous servant du Framework Vue, vous devrez réaliser une application constituée de plusieurs composants et permettant d'offrir à l'utilisateur un compteur de nombres pairs et impairs triés et présentés différemments.

Le composant principal servira de socle à l'ajout et à la présentation de plusieurs composants enfant. Dans ce parents se trouveront idéalement un composant enfant destiné à la manipulation du compteur (son lancement, sa mise en pause et son retour à zéro) ainsi qu'une répétition d'un autre composant.

Ce composant répété sera un composant destiné à la présentation des nombres, mais devra avoir un style (une couleur et / ou un arrangement de son texte) différent en fonction de s'il s'agit d'un nombre pair ou d'un nombre impair. Le composant suivra l'avancement du compteur et affichera les nombres au fur et à mesure de leur génération. De plus, en cas de remise à zéro ou de mise en pause du compteur, il tiendra compte de la logique métier.

Les composants locaux et globaux

Pour enregistrer nos composants, il existe plusieurs façon de le faire. Notre méthode actuelle concerne l'ajout des composants en tant que composants globaux. De la sorte, on peut utiliser les composants dans toute notre application.

Cette méthode a de nombreux avantages mais également quelques soucis. En effet, lorsque l'on lance notre application, Vue va devoir ajouter tous les composants au lancement de l'application. Dans le cas d'une très longue liste de composants, ce processus est couteux en mémoire et en performances.

Il est cependant possible de rendre nos composants locaux, en les enregistrant au niveau des composants qui doivent s'en servir. Pour ce faire, il suffit de passer par l'utilisation de la propriété **components** de notre composant, afin de l'informer que l'on compte utiliser tel composant dans son template.

```
components: {
  'dog-card': DogCard
},
```

```
components: {
  DogCard: DogCard
},
```

```
components: {
  DogCard
},
```

Cette propriété peut être alimentée de clés et de valeurs de plusieurs façon, rendant possible plusieurs tags au niveau du template :

```
<dog-card></dog-card>
```

```
<DogCard />
```

Les Composants Dynamiques

Partons du principe que nous voulons faire en sorte de n'afficher qu'un composant parmi plusieurs en fonction des options de visionnage que l'utilisateur aurait choisit. Par exemple, à la façon d'onglets, nou pourrions avoir des boutons affichant ou masquant des composants via l'utilisation de multiples **v-if.**

Cet assemblage de notions peut être grandement simplifié en utilisant ce qui s'appelle les **composants dynamiques**. Vue permet d'utiliser une balise HTML personnalisée : **component,** avec un attribut **is** qui permet de définir quel sera le composant à rendre à cet emplacement.

```
<button @click="switchActiveComponent('dogs-componnent')">Dogs</button>
<button @click="switchActiveComponent('masters-componnent')">Masters</button>
<button @click="switchActiveComponent('adoptions-componnent')">Adoptions</button>

<component :is="selectedComponent"></component>
```

En liant cet attribut à une directive **v-bind**, il est faisable de facilement passer d'un composant à l'autre en fonction de nos besoins (un peu à la façon d'un switch).

Keep Alive

Si au sein de notre composant dynamique nous avions par exemple un champ de type input, alors il nous faut faire attention. En effet, lors du changement d'un composant dynamique, toutes ses données seront perdues car ce composant est détruit.

```
<keep-alive>
  <component :is="selectedComponent"></component>
</keep-alive>
```

Pour changer cela, il est possible d'entourer notre composant dynamique de balises **keep-alive** pour conserver son état dans le cache en cas de changement.

Teleport

Il est possible avec Vue de changer l'ordre des éléments HTML pour permettre à un composant de se retrouver par exemple à la racine de notre DOM d'application au lieu d'être imbriqué dans une chaine de parents et d'enfant.

Pour ce faire, il suffit d'utiliser la balise **teleport** ainsi que ses attributs:

- **to:** Pour fixer l'endroit auquel notre élément doit se retrouver. C'est ici que l'on dira que notre composant doit se retrouver à la racine de l'application et non dans la balise du composant où se trouve la balise **teleport.** Il se trouvera ensuite juste avant la balise de fin de l'élément HTML auquel on l'aurait envoyé.

Les Styles

La section style de nos composants / de notre application est actuellement configurée de façon globale. C'est-à-dire que si l'on cherche à modifier le style d'un élément de type **h1** dans l'un de composants de notre appli, alors ce seront tous les éléments de ce type qui seront modifiés, qu'ils soient ou non dans notre composant spécifiant cette règle CSS.

```
<style scoped>
p {
    color: red;
}
</style>
```

Pour y remédier, il nous faut ajouter à notre section style l'attribut **scoped** pour rendre le style unique à notre composant. Pour fonctionner, Vue va ajouter des attributs personnalisés dans les éléments de notre composant, et ajouter cette différence dans les sélecteurs de notre CSS.

Les Slots

Dans un soucis de séparer encore un peu plus notre code, il est possible d'utiliser ce que l'on appelle des **slots**. Par exemple, nous aurions réalisé un design assez poussé pour la réalisation d'une carte, mais voudrions rendre le contenu de sa section **template** dynamique dans le but de garder une base de design commune mais un contenu libre.

```
<base-card>
  <p>
    Je suis dans le composant BaseCard !
  </p>
</base-card>
```

En utilisant cette feature, il nous est possible de mettre le contenu HTML dynamique entre les deux balises de notre composant, comme dans l'exemple ci-contre.

```
<template>
    <div>
        <slot></slot>
    </div>
</template>

<script>
    export default {
        name: 'base-card'
    }
</script>


<style scoped>
    div {
        background-color: gray;
        padding: 5px;
        width: 50%;
        border-radius: 5px;
    }
</style>
```

Tout ce qu'il nous suffit de faire, c'est d'utiliser la balise **<slot>** dans notre composant dans lequel on veut injecter de l'HTML entre les balises de son sélecteur.

Avoir plusieurs slots

Il est également possible de se servir de plusieurs slots. Imaginons que notre composant de carte offre deux emplacements dans lesquels on peut injecter de l'HTML : le **header** et le **body** de notre carte. Il nous faut un moyen de pouvoir cibler dans lequel des deux slots l'HTML va être envoyé. Pour ce faire, il existe les **slots nommés.** Il suffit d'ajouter l'attribut name ainsi que sa valeur à chacun de nos slots.

```
<template>
    <div>
        <header>
            <slot name="header"></slot>
        </header>
        <body>
            <slot name="body"></slot>
        </body>
    </div>
</template>
```

Au niveau de l'HTML qui doit être envoyé, il va nous falloir entourer les différentes parties dans des balises **template** comportant la directive **v-slot** dans le but de cibler lequel des slots va recevoir toute cette section d'HTML. Cette syntaxe, tout comme pour **v-bind** et **v-on**, peut être raccourcie par l'utilisation du symbole « **#** »

```
<base-card>
<template v-slot:header>
  <h3>
    BaseCard title
  </h3>
</template>
<template v-slot:body>
  <p>
    Je suis dans le composant BaseCard !
  </p>
</template>
</base-card>
```

Un peu plus d'infos sur les slots

Il est intéressant de noter que lorsque l'on n'envoi aucune donnée à notre slot via l'utilisation d'un template au niveau du parent, il est possible de spécifier de l'HTML qui sera présent par défaut. Pour ce faire, il suffit de placer de l'HTML entre les deux balises de notre slot, comme dans l'exemple ci-contre:

```
<div>
    <header>
        <slot name="header">
            <h3>Header par défaut</h3>
        </slot>
    </header>
    <body>
```

De plus, au niveau de Vue, il existe une propriété **this.$slots** qui contient les informations sur les données passées à nos différents slots. En se servant de cette propriété, nous pouvons avoir un peu plus de contrôler sur les slots. Par exemple, empêcher le rendu d'un élément HTML vide correspondant à notre slot non alimenté. Il suffit pour cela de passer par la vérification que notre slot est différent de **undefined**:

```
<header>
    <slot v-if="$slots.header" name="header">
        <!-- <h3>Header par défaut</h3> -->
    </slot>
</header>
```

Scoped Slots

Pour en finir avec les slots, imaginons désormais que l'on ait au sein de notre composant enfant l'envie de créer plusieurs slots basés sur l'utilisation d'une **variable de type conteneur** (un tableau d'objet par exemple) et que l'on souhaite définir la structure HTML de cet assemblage en dehors de notre composant, via l'utilisation de **slot.**

```
<base-card>
  <template #default="slotProps">
    <b> {{ slotProps.anotherProp}} </b> {{ slotProps.propName }}
  </template>
</base-card>
```

```
<template>
    <div>
        <ul>
            <li v-for="name, index in names" :key="index">
                <slot :propName="name" :anotherProp="index"></slot>
            </li>
        </ul>
    </div>
</template>
```

Pour ce faire, Vue nous offre ce qui s'appelle les **scoped slots.** Pour s'en servir, il nous faut ajouter des attributs à notre balise **slot** dans l'élément enfant. Agrémentés de la directive v-bind, ces attributs vont être disponibles au niveau du parent via la création d'un objet les contenant sous la forme de propriétés. Pour créer et avoir accès à cet objet, il suffit d'ajouter un signe égal suivit du nom que l'on veut lui donner après la directive **v-slot** dans la balise **template.**

Les Formulaires

Lier nos inputs

Dans la plupart des cas, le moyen de plus simple d'avoir un lien direct entre ce que contient nos inputs et notre code métier (nos variable) est d'utiliser la directive **v-model**.

De la sorte, nous nous assurons que toute modification de notre variable sera visible dans notre interface et que tout changement de notre interface va se répercuter par un changement de valeur de notre variable.

Il ne nous restera plus qu'à créer une variable pour chacun des champs de nos inputs afin de conserver ce phénomène dans tout le formulaire.

L'avantage de v-model sur les références vues précédemment est également le fait qu'il prend en compte le type de l'input (un nombre restera un nombre au niveau de son type, et ne deviendra pas un string comme cela serait le cas avec les références)

Et le reste ?

Si l'on veut, il est aussi possible d'utiliser **v-model** pour nos select de la même façon que pour nos inputs. Pour obtenir une valeur par défaut dans notre select, il suffit de mettre comme valeur à notre variable liée à la directive la valeur stockée dans l'attribut **value** de notre balise **option.**

Pour les checkboxes et les radio buttons, **v-model** est également disponible.

Dans le cas d'une **série de checkbox** portant le même nom, il nous faut comme type de variable un tableau. En sortie, nous aurons un objet comportant sous sa propriété target les différentes valeurs sélectionnées. Si l'on se sert d'une **checkbox unique**, la variable liée devra être un booléen.

Lors de l'utilisation avec des **boutons radios**, le type lié devra être une valeur unique, tel une string ou la valeur null.

La Validation

Pour créer une méthode de validation de nos inputs, il suffit de se lier à l'un de leur évènement, par exemple **blur,** via la directive **v-on.** Suite à cela, il nous faut nous assurer que cet évènement déclenchera une méthode chargée d'effectuer la vérification du contenu de notre input, puis de modifier une variable dans notre code métier.

```
data() {
    return {
        username: null,
        usernameValidity: 'pending'
    }
},
methods: {
    validateInput(name) {
        if (name === 'username') {
            if (this.username !== '') {
                this.usernameValidity = 'valid'
            } else {
                this.usernameValidity = 'invalid'
            }
        }
    }
}
```

Cette variable sera de son côté utilisée dans des directives **v-if** pour afficher ou masquer des éléments HTML en fonction de la validité de notre input

```
<form @submit.prevent="onSubmit($event)">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" v-model="username" @blur="validateInput('username')">
    <p v-if="usernameValidity ==='invalid'">Username is invalid!</p>
</form>
```

@Utopios Consulting

Créer nos propres composants de formulaire

Si l'on a créé notre propre composant de formulaire, comme par exemple une série de valeurs via des boutons, alors il peut être souhaitable de pouvoir utiliser v-model sur notre propre élément. Pour ce faire, il faut savoir que l'ajout de la directive v-model sur un composant personnalisé va lier v-bind sur une **prop** particulière. De plus, l'émission d'un évènement bien précis va être écoutée.

- **Pour la prop**: Il va nous falloir obtenir la valeur de **modelValue** pour lier une variable du parent à notre élément enfant.
- **Pour l'évènement**: Il va nous falloir émettre l'évènement **update:modelValue** pour modifier la valeur de la variable du parent.

Sachant cela, il ne nous reste plus qu'à initialiser notre composant en prenant en compte la valeur de **modelValue** et de déclencher l'émission de l'évènement **update:modelValue** à chaque changement de cette valeur.

Formulaire de contact

TP

En vous servant du Framework Vue, vous devrez réaliser une une page permettant à un utilisateur d'envoyer un formulaire de contact.

Ce formulaire devra avoir comme informations le **nom**, le **prénom**, l'**adresse mail** et le **numéro de téléphone** du client, avec en supplément le **type de service** qu'il cherche à joindre, le **sujet de sa question** ainsi que le **corps du message** qui se verra transmis au service choisi.

Le formulaire devra procéder à toutes les **vérifications requises** afin d'empêcher l'envoi vers un service de formulaire incomplet. Du moment que le formulaire n'est pas correct, il sera par exemple impossible à l'utilisateur d'appuyer sur le bouton d'envoi du formulaire. Afin de vérifier l'envoi du formulaire, vous ferez en sorte :

- soit d'afficher dans une section de la page le contenu du formulaire en cas de réussite
- soit de loguer dans la console le fameux formulaire

Requêtes HTTP

L’utilité d’un Backend

Les application front-end ont l’avantage de proposer une interface des plus agréable et bien souvent des plus réactive. Cependant, aussi belle et fonctionnelle que puisse être notre application, elle possède un problème de taille: les variables et donc les données ne sont disponibles que dans la mémoire de notre ordinateur. Il n’est pas possible de posséder une réelle persistance de données et même impossible de faire transiter nos données à travers la même application mais sur plusieurs ordinateurs. Pour ce faire, nous avons besoin d’un **backend.** Un Backend est généralement une base de données couplée avec une API servant de contrôle des accès à cette base de données. En se servant de ce que l’on appelle les verbes d’une API REST, il nous est possible d’effectuer diverses manipulation des données :

- **Get:** Sert à la récupération des données
- **Post:** Sert à l’ajout de données
- **Put / Patch:** Servent à l’édition des données
- **Delete:** Sert à la suppression des données

Dans le cadre de nos besoins, il est possiblede recourir à des serveurs simples d’utilisation, comme par exemple **Google Firebase** ou de **JSON Server**

Axios à la rescousse

Lorsque l'on utilise les requêtes HTTP, les premiers verbes que l'on est généralement amené à utiliser sont sans doute les verbes **GET** et **POST**. Pour réaliser une requête de ce type, il nous faut tout d'abord installer le package axios pour ajouter ses fonctionnalités à notre application. Pour ce faire, il suffit d'utiliser une ligne de commande dans notre terminal :

```
$ npm i axios
```

Une fois installé, son utilisation passera donc par un import dans notre section **script** de sorte à bénéficier de ses nombreuses méthodes :

```
<script>
import axios from 'axios'
```

Enfin, pour réaliser un envoi, il nous faut par exemple utiliser la méthode **.post()** proposée par **axios:**

```
axios.post("https://vue-demo-http-requests-default-rtdb.europe-west1.firebasedatabase.app/surveys.json",
  {
    name: this.enteredName,
    rating: this.chosenRating
  })
```

@Utopios Consulting

Récupérer les données

Pour récupérer les informations, il va nous falloir envoyer une requête à la même adresse, mais en utilisant le verbe GET. Pour ce faire, il va également falloir prendre en compte qu'une requête HTPP créé ce que l'on appelle une **Promise.** Une promesse est un objet Javascript qui est amené à se terminer avec ou non une erreur, et qu'il va falloir attendre pour obtenir les données (De sorte, notre application ne se fige pas lorsque l'on réalise une requête HTTP).

Pour manipuler nos données, il va nous falloir ajouter un **.then()** à la fin de notre méthode **.get().** Cette partie sera executée lorsque la promesse ne sera plus **pending** (Lorsque son **resolve()** aura été déclenché) : Dans cette fonction nous allons placer une autre fonction qui va prendre en paramètre la réponse (ce paramètre est envoyé automatiquement par la promesse) et qu'il va être possible de traiter pour par exemple l'afficher :

```
loadSurvey() {
  let baseURL = "https://vue-demo-http-requests-default-rtdb.europe-west1.firebasedatabase.app/surveys.json"

  console.log("Loading surveys...")
  axios.get(baseURL).then((response) => {
    if (response.status == 200) {
      console.log("Response arrived !")
      const results = []
      for (const id in response.data) {
        results.push({id: id, ...response.data[id]})
      }
      this.results = results
    }
  })
}
```

@Utopios Consulting

Ajouter un message de chargement

Afin de rendre notre interface un peu plus « user friendly », il est tout à fait possible d'ajouter un message de chargmeent pour l'informer que des données sont en cours de récupération. Pour ce faire, le moyen le plus simple est d'utiliser la directive **v-if** ainsi qu'un booléen :

```
<p v-if="isLoading">Is Loading...</p>
<ul v-if="!isLoading">
```

```
export default {
  data() {
    return {
      results: [],
      isLoading: false
    }
  },
  components: {
```

```
console.log("Loading surveys...")
this.isLoading = true
axios.get(baseURL).then((response) => {
  if (response.status == 200) {
    console.log("Response arrived !")
    const results = []
    for (const id in response.data) {
      results.push({id: id, ...response.data[id]})
    }
    this.results = results
    this.isLoading = false
  }
})
```

Et les problèmes ?

En plus d'un message de chargement, l'autre ajout que nous pourrions faire serait de gérer l'absence de données ainsi que de réaliser un message d'erreur qui s'afficherait en cas de non résolution de notre promesse.

En cas de non récupération de données, nous pourrions faire en sorte de vérifier simplement la longueur de notre tableau de résultats :

```
</div>
<p v-if="isLoading">Is Loading...</p>
<p v-else-if="!isLoading && (!results || results.length == 0)">No stored surveys in the database !</p>
<ul v-else-if="!isLoading && results && results.length > 0">
```

Pour les vrais messages d'erreur, il va falloir ajouter à la fin de notre / nos méthodes **.then()** la méthode **.catch()** Cette méthode va récupérer l'erreur soulevée par la promesse dans le cadre de son **reject()**

Dans le template, il sera possible de se servir d'une variable pour la combiner avec une directive v-if dans le but d'afficher un message d'erreur le cas échéant.

```
<p v-else-if="!isLoading && error"> {{ error }} </p>
```

```
}).catch((error) => {
  this.error = 'Failed to fetch data, please try again later.'
})
```

Les Codes d'erreurs

Si maintenant l'erreur vient du Backend, alors il va nous falloir traiter les différents code dans la propriété **status** de notre réponse. En fonction du code que cette propriété contient, il va nous falloir soit afficher le résultat, soit afficher un message d'erreur adéquat pour l'utilisateur. Par exemple, une erreur courante pourrait être une erreur de la famille 4xx comportant la plupart des erreurs d'authentification (en cas de non droit pour accéder à telle ou telle ressource de l'API).

```
.then((response) => {
  if (response.status == 200) {
    console.log("Response arrived !")
    const results = []
    for (const id in response.data) {
      results.push({id: id, ...response.data[id]})
    }
    this.results = results
    this.isLoading = false
  } else if (response.status === 401) {
    this.isLoading = false
    this.error = 'You are not authorized to get access to these data. Please contact the website admin.'
  }
}).catch((error) => {
```

@Utopios Consulting

Collection Musicale

TP

En vous servant du Framework Vue, vous devrez réaliser une application constituée de plusieurs composants et permettant à un utilisateur de stocker des informations concernant sa collection d'albums de musique via une persistance en base de données et un appel à une API par votre application Front.

Un album sera constitué d'un **titre**, d'une **date de parution**, d'une **couverture** (l'URL suffira), d'une **collection de pistes** ainsi que d'un **score** (pensez au système de notation par étoiles, un grand classique !). Chaque piste sera de son côté constituée par un **titre**, un **artiste** ainsi qu'une **durée**.

La collection d'albums devra pouvoir être manipulée de façon CRUD, en permettant à l'utilisateur de voit les détails d'un album, d'ajouter un album, d'en modifier le contenu ou d'en supprimer un. Toutes ces actions auront recourt à un appel à une API pour offrir une persistance des informations.

Le Routing

Les Bases

Pour avoir recourt au Routing dans notre application, il va nous falloir réaliser plusieurs modifications. Tout d'abord, il va nous falloir installer le package supplémentaire pour bénéficier du router :

```
$ npm install --save vue-router
```

Via ce nouveau package, il nous est possible de bénéficier de l'import de la fonction createRouter de la sorte :

```
import { createRouter } from 'vue-router';
```

Une fois fait, il nous est donc possible de créer notre routeur en appelant cette fonction. En paramètre, il va nous falloir un objet contenant les propriétés **history** et **routes**.

L'history correspond à la notion d'historique et doit idéalement être alimenté par la fonction **createWebHistory()** elle aussi importée de notre package. Via l'utilisation de cette fonction, on informe notre application Vue que l'on va utiliser le mécanisme de base de notre navigateur pour créer le système d'historique de l'application.

Enregistrer nos routes

Dans la propriété routes, il va nous falloir alimenter un tableau de routes qui seront à chaque fois des objets. Ces objets auront pour propriété le chemin - leur **path**, et leur propriété **component** qui va relier ce chemin à un composant en particulier.

```
const router = createRouter({
    history: createWebHistory(),
    routes: [
        {path: '/teams', component: TeamsList},
        {path: '/users', component: UsersList}
    ]
});
```

Une fois nos routes ajoutées, il va falloir informer l'application qu'elle doit utiliser notre router nouvellement configuré. Pour cela, il suffit de faire appel à la méthode **.use()** de notre objet d'application :

```
app.use(router);
```

Enfin, il va nous falloir spécifier à Vue où nos routes devront être affichées. Pour cela, nous allons devoir utiliser

```
<main>
  <router-view></router-view>
</main>
```

Changer de route avec router-link

Il nous reste maintenant à faire en sorte que l'on puisse passer d'une route à l'autre en appuyant sur un lien ou un bouton de notre application. Pour ce faire, il est possible d'avoir recourt à **router-link.** router-link est un composant que l'on peut utiliser de part notre import du router. Pour l'utiliser, il suffit de faire appel à sa balise et d'alimenter son attribut **to** du chemin vers la route :

```
<ul>
  <li>
    <router-link to="/teams">Teams</router-link>
  </li>
  <li>
    <router-link to="/users">Users</router-link>
  </li>
</ul>
</nav>
```

Au bout du compte, cette balise va réaliser une balise **<a>** mais va également s'assurer qu'il n'y aura pas apparition d'un fonctionnement par défaut dans la redirection pouvant causer la perte de l'état de notre application.

router-link-active / router-link-exact-active

Lorsque l'on travaille avec le routing, on va voir apparaître sur nos **router-link** des classes du type de **router-link-active**. Cette classe permet d'altérer le CSS de notre élément en se basant sur la présence ou non de la classe.

Une variante : **router-link-exact-active**, existe également. La différence est que celle-ci n'est présente que si la route correspond exactement à l'attribut **to** de la balise, et non pas quand elle comprend cet attribut dans le chemin complet de la barre d'URL.

En altérant les propriétés dans notre objet de router, il est possible de modifier quelles classes seront ajoutées à notre balise lors des deux cas vus plus haut.

Naviguer par le code

Si désormais, pour une raison X ou Y, nous voudrions que la navigation dans notre application web se fasse par le code et non lorsque l'utilisateur va cliquer sur un lien en particulier, il va nous falloir utiliser la méthode **.push()** présente sur notre objet de router. Cet objet nous est accessible via **this.$router** dans nos méthodes:

```
confirmInput() {
  // Code B4

  setTimeout(() => {
    this.$router.push('/teams')
  }, 1000)
}
```

Les paramètres de route

Si l'on veut, il est possible d'utiliser des routes avec des variables. Par exemple, si l'on veut pouvoir modifier ou afficher le client numéro X, il va falloir spécifier la valeur de X dans la route menant aux détails du client. Pour ce faire, il nous faut une route de ce type :

```
{path: '/teams/:id', component: TeamMembers}
```

Une fois envoyé, un paramètre peut ensuite être récupéré par l'utilisation de la propriété **this.$route** et de sa sous-propriété **params**

```
const teamId = this.$route.params.id

const selectedTeam = this.teams.find(x => x.id == teamId)
const membersOfTeam = selectedTeam.members
```

Envoyer des chemins paramétrés

Si l'on veut désormais créer des **router-link** porteurs de paramètres, il va nous falloir modifier un peu leur syntaxe en nous servant de **v-bind** pour y placer non plus une valeur statique mais désormais une valeur se servant de variables pour fonctionner :

```
<router-link :to="`/teams/${id}`">View Members</router-link>
```

On peut aussi par une propriété **computed** pour obtenir un résultat plus élégant de la sorte :

```
<router-link :to="teamMembersLink">View Members</router-link>
```

```
computed: {
  teamMembersLink() {
    return `/teams/${this.id}`
  }
}
```

Attention cependant, lors de la modification des routes, les composants ne sont pas forcément détruits et il se peut que leur valeurs ne soient pas actualisées. Pour éviter ce problème, on a recourt à des watchers pour observer le changement de la variable **$route**

Passer les paramètres de routes via props

Si l'on le veut, il est possible d'avoir recourt à l'utilisation des props pour récupérer les paramètres de route. Par défaut, le router n'envoie pas les valeurs via ce mode de fonctionnement, mais il nous est possible d'altérer ce fait.

Pour le faire, il nous faut modifier la configuration de notre route en ajoutant la propriété **props** avec pour valeur **true** de la sorte :

```
{path: '/teams/:id', component: TeamMembers, props: true}
```

Movin’Out App

TP

En vous servant du Framework Vue, vous devrez réaliser une application constituée de plusieurs composants qui pourra être utilisée par une entreprise de gestion de biens immobiliers.

Par l’utilisation des routes, vous réaliserez deux sections ayant chacune plusieurs routes internes permettant la manipulation de type CRUD sur des **personnes** et sur des **logements**.

Chaque personne aura comme informations son **nom**, son **prénom**, son **adresse mail** et son **téléphone**.

Chaque logement sera constitué d’un **numéro de voie**, d’un **intitulé de voie**, d’un **code postal** ainsi que d’un **nom de commune**.

Les pages de logements et de personnes permettront donc de naviguer via l’appui sur des boutons vers les pages **d’ajout**, de **modification**, de **suppression** ou de **visionnage** des informations de l’entité à laquelle la page est rattachée.

En bonus, il sera possible d’avoir le lien entre les deux entités, pour lier une ou plusieurs personnes à une ou plusieurs adresse. De la sorte, il sera possible de voir le nombre d’habitants à une adresse donnée ou le nombre de logements possédés par une personne depuis leur listing dédié.

La redirection

Si l'on veut faire en sorte que notre utilisateur n'arrive pas sur une page vierge en cas de lancement de notre application, il nous est possible de rediriger ce chemin vierge vers l'une des pages de notre application. Pour ce faire, il va nous falloir créer un chemin, qui va non pas renvoyer un composant mais une redirection vers une autre route de notre application :

```
{path: '/', redirect: '/teams'},
```

Il est également possible de définir un alias sur notre route pour prendre en compte plusieurs chemins pouvant y mener :

```
{path: '/teams', component: TeamsList, alias: '/'},
```

Cependant, en usant de cette méthode, l'URL ne changera pas. Il nous faut donc choisir quelle logique nous voulons appliquer et faire un choix de méthodologie adéquat.

Une page 404

Dans le cas où notre utilisateur entre une URL non valide, on veut également pouvoir y réagir et renvoyer une page contenant un message d'erreur de type 404 (plus ou moins personnalisé). Pour ce faire, il va nous falloir gérer une route d'un format spécifique :

```
{path: '/:catchAll(.*)', component: NotFound}
```

Ici, catchAll va prendre la valeur de n'importe quelle route pouvant être de n'importe quel assemblage de caractères via l'utilisation d'une expression régulière **.*** Dans ce cas, cette route, se plaçant à la fin du tableau des routes, va renvoyer un composant NotFound comprenant le code de notre page d'erreur.

Les routes imbriquées

Si l'on le veut, il est possible, au sein d'une route, de charger une autre sous-route. Si par exemple nous voulions faire une route pour afficher les recettes de cuisine d'un livre et qu'au sein de ces recette on puisse, en cliquant sur l'une d'elle, avoir à proximité les ingrédients de la recette, alors nous pourrions le faire via des routes imbriquées. Pour ce faire, il suffit d'utiliser dans l'un des objets de route la propriété **children**.

```
{path: '/teams', component: TeamsList, children: [
    {path: ':id', component: TeamMembers, props: true}
]},
```

Cette propriété va pouvoir contenir un nouveau tableau de routes, qui sera rendu au sein de notre composant parent par appel à la balise **router-view** mais cette fois-ci dans le composant où doit être rendue la route enfant.

Une autre syntaxe pour :to du router-link

Dans le cadre d'une application de grande échelle, nous allons rapidement nous retrouver avec des liens de routage à rallonge, et l'écriture de ces derniers, avec tous leurs paramètres éventuels, deviendra rapidement fastidieuse…

Pour y remédier, il est possible d'utiliser un autre type de paramètres pour notre directive : un **objet**. Cet objet pourra contenir plusieurs informations, dont par exemple : **name** et **params**.

```
{ name: 'team-members', params: { id: this.id }}
```

name correspond au nom d'une route, qui peut lui être affecté en ajoutant à notre chemin la propriété name et lui affectant une string. De son côté, **params** va être un objet contenant un ensemble de clés et de valeurs représentant les paramètres demandés par notre route :

```
{name: 'team-members', path: ':id', component: TeamMembers, props: true}
```

@Utopios Consulting

Les paramètres de requête

En plus d'améliorer la scalabilité de notre application, la syntaxe vue précédemment permet d'ajouter à notre route des paramètres de requête très facilement. Pour ce faire, il suffit de faire appel à la propriété **query** qui fonctionne de la même façon que **params.** Optionnels de nature, les paramètres de requêtes n'ont pas besoin d'être ajouté à la liste des routes de notre router dans le fichier **main.js**

```
return {
  name: 'team-members',
  params: { id: this.id },
  query: { sort: 'asc'}}
```

Pour les utiliser, une fois dans le composant, il est possible d'y accéder via la syntaxe **this.$route.query** qui contiendra notre objet avec l'ensemble des clés et des valeurs.

Avoir plusieurs router-view

Il nous est possible d'avoir plusieurs **router-view** au sein d'un même composant. Pour ce faire, il est plus utile de les nommer, pour permettre dans chacun d'entre eux l'envoie de composants différents. Pour ce faire, on va utiliser l'attribut **components** à la place de **component** dans notre route, et donner une clé correspondant au nom de notre **router-view**.

```
components: { default: TeamsList, footerRouter: TeamsFooter},
```

Une fois fait, il nous suffit de nommer nos router-views via l'attribut **name.** Comme pour les slots, il existe un nom **default** qui n'est disponible que pour un seul **router-view.**

```
<main>
  <router-view></router-view>
</main>
<footer>
  <router-view name="footerRouter"></router-view>
</footer>
```

Contrôler le scrolling

Si l'on le veut, il est possible au sein d'une application Vue de déclencher le scrolling de notre page en cas de changement de route. Par exemple, on pourrait vouloir renvoyer l'utilisateur tout en haut de la page en cas de passage à un autre emplacement de notre site web.

Pour ce faire, il va nous falloir utiliser la méthode **scrollBehavior(to, from, savedPosition)** de notre router et retourner un objet possédant une propriété **left** et un **top.** En retournant à chaque fois un objet avec 0 aux deux valeurs, on s'assure que à chaque changement de route, on va se retrouver au sommet de la page.

```
],
scrollBehavior(to, from, savedPosition) {
    return {top: 0, left: 0}
}
```

Les Navigation Guards

Lorsque l'on veut empêcher un utilisateur d'accéder à une route de notre application s'il n'est pas logué, ou empêcher un utilisateur de quitter une page en cas de modifications non sauvegardées, on a tendance à recourir aux **Navigation Guards**. Ces guards sont des méthodes de router.

```
router.beforeEach((to, from, next) => {
    console.log(to, from)
    next(true);
})
```

Notre premier guard sera le **.beforeEach()** qui prendra en paramètre une fonction et lui enverra trois paramètres : **to**, **from** et **next**. to et from serviront à connaître la route de départ et la route d'arrivée.

next() sera de son côté une fonction s'assurant du contrôle du passage vers cette route d'arrivée. Si l'on l'appelle, alors on peut lui passer un paramètre booléen true pour autoriser ou non la navigation, ou lui passer une string ou un objet en cas de redirection (on peut aussi lui passer false pour simplement l'empêcher).

Déplacer notre guard

Il est aussi possible de ne pas rendre notre guard global, en plaçant dans les propriétés d'un objet de route la propriété **beforeEnter.** Cette méthode va également prendre en paramètre les trois to, from et next et aura un fonctionnement semblable à notre guard global, à ceci près qu'il ne sera testé que si l'on va sur cette route en particulier.

```
beforeEnter(to, from, next) {
    console.log(to, from)
    next(true)
}
```

Enfin, il est possible de se servir des cibles de vie de Vue pour faire appel au guard dans le composant. Pour ce faire, il faut utiliser **beforeRouteEnter()** qui est aussi une méthode possédant en paramètre **to**, **from** et **next**.

beforeRouteUpdate() / afterEach()

Dans le composant, il existe également le guard **beforeRouteUpdate()** qui peut nous servir au moment de modification d'un composant. Si par exemple nous sommes en train de visionner une recette de cuisine, et que l'on charge une autre recette. Alors le composant sera le même, et notre route modifiera seulement le contenu de ce composant. Dans ce genre de cas, le guard **beforeRouteUpdate()** sera appellé.

Au niveau global, le guard **afterEach(to, from)** peut nous être utile pour exécuter du code après la navigation. Il pourrait être utile dans le cas où l'on veut par exemple faire des logs de navigation et les stocker dans notre serveur afin de garder une trace des pages les plus visitées sur notre site.

beforeRouteLeave()

Ce guard va nous être utile en cas d'utilisation d'un formulaire, car il sert à empêcher la navigation depuis une page. Par exemple, on peut vouloir informer l'utilisation qu'il n'a pas sauvegardé les changements qu'il vient d'effectuer sur la page actuelle, et le faire tant qu'il n'a pas confirmé être sur de vouloir quitter le formulaire.

C'est le problème que tente de résoudre **beforeRouteLeave(to, from, next),** un autre hook disponible dans notre composant. Ce guard va être appelé avant ceux vu précédemment.

```
},
beforeRouteLeave(to, from, next) {
  if (this.savedChanges) {
    next()
  } else {
    alert("Vous avez des changements non sauvegardés !")
  }
}
```

@Utopios Consulting

Les MétaDonnées de route

Dans notre objet de route, il est possible d'utiliser une propriété **meta** et de lui passer en valeur un objet. Cet objet contiendra un ensemble de clés et de valeurs qui seront également accessible dans notre composant via la syntaxe **this.$route.meta**. De plus, il est aussi possible de s'en servir dans les guards via l'utilisation de leurs paramètres **to.meta** ou **from.meta**.

```
{path: '/teams', meta: {needsAuth: true}, compo
```

Via cette méthodologie, on peut faire en sorte de vérifier si une route nécessite l'authentification pour être consultée, et réagir au fait que notre utilisateur est connecté ou non.

Plusieurs fichiers pour les routes

Il est intéressant de savoir qu'il est possible de séparer nos routes dans plusieurs fichiers. Pour exporter notre logique de routing du fichier **main.js** nous allons placer toute cette logique dans un fichier **router.js**

```
import { createApp } from 'vue';
import router from './router.js'

import App from './App.vue';


const app = createApp(App)

app.use(router);

app.mount('#app');
```

Une fois fait, il ne restera plus qu'à exporter notre router et à l'importer dans le fichier **main.js** pour l'ajouter à l'application via sa méthode .**use()**

JdR Beyond

TP

En vous servant du Framework Vue, vous devrez réaliser une application constituée de plusieurs composants et de plusieurs routes. Cette application devra permettre à des utilisateurs de **stocker des informations sur leurs personnages de jeu de rôle**, par exemple de Dungeons & Dragons.

Via le passage par les **routes**, l'utilisateur pourra naviguer vers lesdites fiches, les manipuler via un **CRUD** (qui sera idéalement persistant via des appels à une API). Les fiches posséderont par exemple comme information le **nom** du personnage, sa **classe**, sa **race**, son **genre**, le **nom du joueur** le possédant, ses **caractéristiques** (Force, Vitesse, etc…) et son **histoire**. N'oubliez pas de contrôler dans les formulaire la validité des informations avant de réaliser le stockage des modifications.

L'utilisateur aura aussi la possibilité de créer, dans une **route dédiée** à cette fonctionnalité, des **groupes de personnages** pour une **campagne**, et d'**accéder directement à la constitution du groupe** dans le but de pouvoir ensuite rapidement lancer des tests de caractéristiques (via des lancers de dés par exemple) des personnages du groupe en cas de besoin, ou d'altérer les informations (par exemple les points de vie, l'expérience ou le butin en or) des personnages au fur et à mesure de l'avancement de la campagne. Pour ce faire, idéalement, une interface sous la forme d'une liste succincte (un tableau avec plusieurs boutons) fera l'affaire.

En bonus, vous pourrez également faire en sorte de **séparer les différentes sessions de jeu** afin de conserver une trace de l'évolution des personnages durant la campagne de jeu de rôle.

@Utopios Consulting

Composition API

Pourquoi existe-t-elle ?

Jusqu'à maintenant, nous créions nos application via l'utilisation d'un objet de création qui combinait en son sein plusieurs propriétés : **data**, **methods**, **computed** et **watch**

Le problème majeur que génère une telle méthode de travail est la répartition de nos fonctionnalités non pas par fonction, mais pas fonctionnement. Il est donc difficile de suivre le fil d'une fonctionnalité car ses variables se trouvent par exemple dans la partie **data**, son fonctionnement dans la partie **method**, et ses contrôles éventuels dans **watch.** Pour y remédier, Vue propose désormais la **Composition API:** une méthode de création de composant qui centralise au sein d'un seul élément **setup** toutes les sections que nous utilisions précédemment.

L’utilisation de ref()

Pour remplacer l’utilisation de notre propriété **data**, on utilisera désormais ce qui s’appelle les **refs. ref** est en réalité une fonction qu’il va falloir importer du module vue dans le but de pouvoir l’utiliser. Son but est la création de variables réactives qui seront automatiquement suivies et rafraichies par Vue.

Cependant, contrairement à l’utilisation de la méthode avec la propriété data, il ne suffit pas de créer les variables pour les utiliser. Il faut que notre propriété **setup** retourne sous la forme d’un objet les clés exposées au niveau du template.

L’utilisation de **ref** créé en réalité un objet contenant la valeur sous une propriété **value**. Pour changer la valeur de notre variable réactive, il nous faut donc passer par la propriété dédiée de l’objet référence.

De la sorte, nous obtenons une structure de code comme dans l’exemple ci-contre:

```
<template>
  <h1>Bienvenue dans mon application !</h1>
  <p>
    <b>Username:</b> {{ userName }}
  </p>
</template>

<script>
  import { ref } from 'vue'

  export default {
    setup() {
      const myName = ref('John')

      setTimeout(() => {
        myName.value = 'Héléna'
      }, 2000)

      return {
        userName: myName
      }
    }
  }
</script>
```

@Utopios Consulting

<script setup>

La méthode de l'export default ainsi que de la propriété setup peut encore être simplifiée en ajoutant simplement l'attribut **setup** à notre balise script. De la sorte, peut se passer des instructions d'export, de la propriété setup ainsi que de l'objet de retour pour avoir cette syntaxe épurée :

```
<template>
  <h1>Bienvenue dans mon application !</h1>
  <p>
    <b>Username:</b> {{ userName }}
  </p>
</template>

<script setup>
import { ref } from 'vue'

const userName = ref('John')

setTimeout(() => {
  userName.value = 'Héléna'
}, 2000)
</script>
```

Travailler avec des objets (valeurs références)

Dans le cas où nous serions amenés à travailler non plus avec des variables simples mais avec des objets complexes, il faut savoir que l'utilisation de la méthode précédente risque de poser problème lors du rafraichissement du DOM. En effet, l'utilisation de **ref()** va créer un objet qui englobera dans sa propriété **value** l'objet que nous voulons référencer. Cette propriété va se retrouver être un objet Proxy dans le but de garder une trace des changements internes de l'objet.

```
<script setup>
import { reactive } from 'vue'

const dog = reactive({
  name: "Bernie",
  breed:"Labrador",
  age: 7
})

setTimeout(() => {
  dog.name = "Rex"
  dog.breed = "Doberman"
  dog.age = 4
}, 2000)
</script>
```

Pour travailler avec des objets, il existe également la fonction **reactive()** également à importer du module **vue.** Ne fonctionnant qu'avec les types références, cette méthode offre l'avantage de ne pas avoir à passer par l'utilisation de la propriété **value** si l'on veut modifier de façon réactive les propriétés de notre objet. On obtient donc un résultat de la sorte :

Remplacer methods

Maintenant que nous avons vu comment obtenir le même fonctionnement que l'ancienne propriété **data**, passons à la propriété **methods.** Pour réaliser des fonctionnalités au niveau de notre code métier, il suffit de créer des fonctions, puis de les lier à des évènements dans notre DOM. De la sorte, nous avons le même fonctionnement que l'ancienne propriété method. Si l'on le veut, il est également possible d'avoir le passage des évènements en paramètre, comme auparavant

```
  </ul>
  <button @click="switchDog">Toogle Dog</button>
</template>

<script setup>
import { reactive } from 'vue'

const dog = reactive({
  name: "Bernie",
  breed:"Labrador",
  age: 7
})

const switchDog = () => {
  dog.name = "Rex"
  dog.breed = "Doberman"
  dog.age = 4
}
</script>
```

Remplacer les computed properties

Pour obtenir des propriétés basées sur des propriétés réactives de notre application, nous utilisions au préalable la propriété **computed** de notre objet de création. Désormais, avec l'approche de la Composition API, il nous faut procéder différemment. Pour réaliser une variable basée sur d'autres variables et qui doit changer dés que l'une de ses dépendances est modifiée, on stocke le résultat de la fonction **computed()** dans une variable. Cette fonction doit être également importée du module **vue**

De la sorte, la réalisation d'un nom complet basé sur le prénom et le nom de famille ressemble désormais à cela :

```
const fullName = computed(() => {
  return firstName.value + ' ' + lastName.value
})
```

@Utopios Consulting

Les Watchers

Il ne nous reste plus qu'à voir comment fonctionne les **watchers** dans le cadre de la **Composition API**. Il va nous falloir encore une fois importer une fonction de l'API : **watch**

```
watch(dog, (oldValue, newValue) => {
  console.log("Ancienne valeur : " + oldValue.age)
  console.log("Nouvelle valeur : " + newValue.age)
  if (newValue.age < 0 || newValue.age > 20) console.log("Le chien ne semble pas avoir un âge correct !")
})
```

Son fonctionnement est semblable à celui de **compute**, à ceci près qu'il lui faut également en premier paramètre la variable qui sera vérifiée à chacun de ses changements. Il est également possible de passer un tableau de dépendance en tant que premier paramètre. Dans ce cas, les valeurs récupérées par la fonction seront multiples et se présenterons également comme un tableau.

Les Références de Template

Si l'on veut désormais utiliser les références dans notre template, pour obtenir au déclenchement d'une fonction la référence vers un élément HTML dans le but d'en parcourir les propriétés et d'en extraire ou d'en contrôler les valeurs, il n'est plus possible d'utiliser la syntaxe **this.$refs.** A la place, il va nous falloir enregistrer la référence de l'élément HTML dans notre script JS de la sorte :

```
const dogAgeInput = ref(null)
```

```
<input type="text" placeholder="Nom du chien" ref="dogAgeInput">
```

Une fois fait, il nous faut utiliser le même nom que notre variable dans l'attribut ref de notre input. De la sorte, les deux seront reliés automatiquement par Vue et il nous sera possible d'y accéder de la sorte :

```
dog.name = dogAgeInput.value.value
```

Utiliser les Props

Pour utiliser les props et passer des valeurs d'un composant parent à un composant enfant, l'utilisation de la Composition API nous force à altérer un peu notre méthode. En effet, sans présence de l'objet **this**, il nous est impossible d'y accéder par la syntaxe **this.nomDeLaProp.** Pour y pallier, nous allons devoir utiliser une nouvelle fonction : **defineProps()** de sorte à obtenir un objet composé de nos différents propriétés passées par le parent.

Son fonctionnement passe par la création d'un objet props qui porte dans ses propriétés les différentes valeurs passées par le parent à ce composant.

```
<template>
    <div>
        <h3>Mon chien</h3>
        <ul>
            <li><b>Nom:</b> {{props.name}}</li>
            <li><b>Race:</b> {{props.breed}}</li>
            <li><b>Age:</b> {{props.age}} ans</li>
        </ul>
    </div>
</template>

<script setup>
import { defineProps } from 'vue'
const props = defineProps({
    name: {
        type: String,
        required: true
    },
    breed: {
        type: String,
        required: true
    },
    age: {
        type: Number,
        required: false,
        default: 1
    }
})
</script>
```

Utiliser les Events

Dans le cadre de la communication **enfant -> parent,** il nous est possible d'utiliser cette fois-ci la fonction **defineEmits()** importée du module Vue. De la sorte, il nous est possible d'enregistrer des évènements que notre composant va pouvoir émettre. Une fois fait, il suffit de passer de nouveau par cette fonction en spécifiant en premier paramètre le nom de l'évènement qui va être émit, puis en paramètres suivants les différentes données à y joindre :

```
const emit = defineEmits(['adopted-dog'])

const adoptDog = () => {
    emit('adopted-dog', props.name)
}
```

Provide / Inject avec la Composition API

Pour réaliser l'envoie de variables depuis un parent vers son descendant, il est possible dans le cadre de la Composition API d'utiliser non plus la propriété provide, mais la fonction **provide()**, qui doit encore une fois être importée du module **vue.**

Dans cette fonction, il suffit de donner en paramètre la clé qui donnera le futur nom de la variable, et en valeur la fameuse variable qui va être envoyée aux descendants :

```
provide('dogAge', dog.age)
```

Dans le descendant, on va devoir importer la fonction **inject()**, et s'en servir pour stocker son retour dans une variable contenant sous la forme d'un objet les différentes variables que l'on aurait voulu récupérer :

```
const dogAge = inject('dogAge')
```

Les Mixins & Composables

Leur Objectif

Dans un soucis de réutilisabilité du code, il est important que les notions des **mixins** (pour l'Options API) et des **fonctions de composition personnalisées** (pour la Composition API) soient vues.

Lorsque l'on pense à une grosse application, les éléments que l'on pourrait vouloir réutiliser souvent sont bien entendu les composants. Ces composants englobent en eux de l'HTML, du CSS et une série de fonctionnalités rendues possibles par la logique métier (par utilisation de variables, de méthodes, de props, d'évènements, etc…).

Imaginons désormais que ce ne soit plus forcément de l'HTML et du CSS que l'on souhaite reprendre à un emplacement de notre application, mais uniquement du fonctionnel. Pour ce faire, il faudrait pouvoir notre script et l'utiliser autre part. C'est ce que nous proposent les mixins et les fonctions de composition personnalisées.

Un premier mixin

Dans le cas où deux composants partageraient exactement le même objet de configuration, il est tout à fait envisageable de le placer dans un fichier .js à part, qui sera importé par les deux composants et leur servira de base commune. Dans cette base commune, il n'est cependant pas possible de placer la dépendance envers les composants et il conviendra donc de la conserver au niveau des objets de configuration de composants.

```
export default {
    data() {
        return {
            alertVisible: false,
        }
    },
    methods: {
        showAlert() {
            this.alertVisible = true
        },
        hideAlert() {
            this.alertVisible = false
        }
    }
}
```

Tout le reste est cependant déplaçable (data, methods, watchers, etc…) et il suffira dans le composant de réaliser un import de notre mixin puis de le référencer au niveau de la propriété **mixins** de son objet de configuration.

```
<script>
    import BaseAlert from './BaseAlert.vue'
    import AlertMixin from '../mixins/alert.js'

    export default {
        name: 'dog-alert',
        components: {
            BaseAlert
        },
        mixins: [AlertMixin]
    }
</script>
```

@Utopios Consulting

Les Composables

Les mixins ont de bons avantages, mais posent problème lors du travail dans de grandes application. En effet, comment savoir d'où viennent les variables utilisées dans les composants dépendant de plusieurs mixins ? Si l'on utilise la Composition API, il est possible de pallier à ce problème en utilisant les fonction de composition personnalisées : les **composables** (ou **hooks** en fonction des préférences de nommage de chacun).

En effet, le fait que l'API de Composition se base sur le principe de centraliser en une seule propriété setup l'ensemble de nos différentes propriétés d'objet de composition rend bien plus facile la création de fonction qui seront importées et qui seront utilisées dans un ou plusieurs composants.

Créer un premier hook

Pour revenir à notre exemple précédent dans le cas de la réutilisation d'une fonctionnalité d'Alert dans plusieurs composants, nous allons refaire ce processus cette fois-ci en nous servant des hooks / composables.

Pour ce faire, il nous faut encore une fois avoir un fichier .JS qui va simplement exporter l'ensemble de no fonctionnalités et de nos données, mais qui cette fois-ci ne sera pas un objet mais une fonction. En récupérant le retour de cette fonction, nous pourrons accéder à son ensemble de fonctionnalité dans notre composant comme dans l'exemple ci-contre:

```
import { ref } from 'vue'

export default function useAlert() {
    const alertIsVisible = ref(false)

    function showAlert() {
        alertIsVisible.value = true;
    }

    function hideAlert() {
        alertIsVisible.value = false;
    }


    return {
        alertIsVisible,
        showAlert,
        hideAlert
    }
}
```

Bien entendu, comme il s'agit de fonction contrairement aux mixins, il est tout à fait possible d'avoir des paramètres en entrée, pour par exemple mettre la visibilité de notre alert à true de base :

```
export default function useAlert(baseVisibility) {
    const alertIsVisible = ref(baseVisibility)
```

```
<script setup>
import useAlert from '../composables/alert.js';
import UserAlert from './UserAlert.vue';

const { alertIsVisible, showAlert, hideAlert } = useAlert()
</script>
```

@Utopios Consulting