Vue.js (2.6) dans le contexte de Laravel et Blade

Document réalisé en LaTeX 2ε avec le logiciel LyX

Table des matières

1 Introduction

Vue est un framework Javascript qui permet de faciliter la construction d'interfaces frontend, à l'instar de React ou Angular. Il est utilisé notamment par Netflix, Adobe, Alibaba et Gitlab. Ce framework est relativement plus facile à utiliser que ses concurrents, tout en restant très performant. Vue propose une documentation en français : https://fr.vuejs.org/v2/guide

2 Mise en place dans Laravel

2.1 Outils par défaut

Laravel intègre Vue.js par défaut. D'autres outils sont également présents par défaut dans Laravel, et nous pourrons nous en servir avec Vue. Tous ces outils sont précompilés dans le fichier *public/js/app.js* :

Outil	Type	Description
Bootstrap	Framework	Fonctionnalités js
jQuery	Bibliothèque js	Facilités js
Popper	Bibliothèque js	Tooltips, popovers
Lodash	Bibliothèque js	Facilités js
Axios	Bibliothèque js	XMLHttpRequest

Remarque : dans ce cours, nous allons utiliser Laravel-mix pour compiler les fichiers sources js. Les modifications devraient donc être apportées dans le dossier resources/js. Nous n'expliquons pas ici comment procéder, veuillez vous référer à la documentation de Laravel : https://laravel.com/docs/5.8/mix

2.2 Préparation de la vue de travail

Nettoyons un peu la vue *welcome.blade.php* :

```
<!doctype html>
<html lang="fr">
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <title>Vue.js</title>
    </head>
<body>
</body>
</html>
```

Pour l'instant, rien ne s'affiche. Incluons nos outils js (dont Vue.js) dans la vue welcome :

```
<body>
    <script src="{{ asset('js/app.js') }}"></script>
</body>
```

En rechargeant la page, on peut constater en console une erreur levée par la bibliothèque Axios :

CSRF token not found...

Cette erreur, qui concerne la sécurité des requêtes de type Ajax, est corrigée en ajoutant une balise meta dans la vue :

```
<title>Vue.js</title>
<meta name="csrf-token" content="{{ csrf_token() }}">
```

2.3 Installation de l'extension "Vue.js devtools" sur le navigateur

Une extension « Vue.js devtools » est disponible pour Chrome et Firefox. Après installation, une icône apparaît à droite de la barre d'adresse. Pour l'instant, est elle grisée :

2.4 Activation de Vue.js et des outils de développement spécifiques à Vue

En fait, une instance de Vue a déjà été créée à la fin script *app.js*, avec le code suivant :

```
const app = new Vue({
    el: '#app'
});
```

Cela signifie que cette instance de Vue va chercher à se monter sur une élément du DOM possédant l'attribut *id="app"*. On pourra d'ailleurs créer plus tard d'autres instances. Ajoutons une div qui corresponde à cette description :

```
<div id="app"></div>
```

Maintenant, l'icône de l'extension est verte.

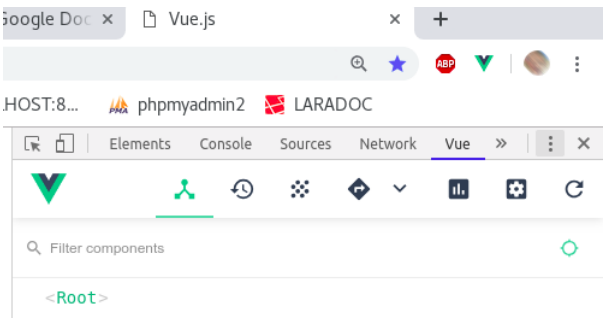
Si vous n'avez pas encore compilé vos sources js avec npm, vous obtenez le message suivant en cliquant sur l'icône verte :

> *Vue.js is detected on this page. Devtools inspection is not available because it's in production mode or explicitly disabled by the author.*

Après avoir exécuté npm install et npm run dev, les scripts js précompilés en mode production sont écrasés avec des scripts en mode développement. Après rafraîchissement, le message change :

> *Vue.js is detected on this page. Open DevTools and look for the Vue panel.*

En affichant les outils de développement du navigateur, on peut observer l'ajout d'un onglet "Vue" :



Pour terminer cette préparation, déplaçons la déclaration de la constante app dans la vue welcome elle-même (en remplaçant const par var, pour garder une cohérence avec le documentation de Vue, et en la supprimant bien du fichier app.js), et exécutons à nouveau npm run dev. On doit obtenir dans l'onglet Vue le même affichage que précédemment.

```
<div id="app"></div>
...
<script>
var app = new Vue({ el: '#app' });
</script>
```

3 Première approche de Vue

A ce stade, nous vous invitons à lire la documentation française de Vue (https://fr.vuejs.org/v2/guide) conjointement avec ce tutoriel, qui est un **résumé adapté à l'intégration de Vue dans Laravel dans le contexte du moteur de gabarits Blade**.

3.1 Les « moustaches » pour lier un contenu

Vue effectue, entre autres choses, du rendu de données « classique ». Pour illustrer cette fonctionnalité, nous allons :

1. Déclarer une variable et attribuer une valeur à cette variable,
2. Afficher la valeur de la variable dans un élément HTML,
3. Modifier la valeur de la variable dynamiquement en console.

Pour déclarer et attribuer (1) une variable message, il faut l'ajouter dans l'objet data de l'instance :

```
var app = new Vue({
    el: '#app',
    data: { message: 'Hello Vue!' }
})
```

Remarque : la valeur de la variable peut également être obtenue via une requête Ajax, ou préparée côté serveur avec Blade par exemple :

```
message: {{ $message }}
```

Une fois déclarée, la variable peut être utilisée (2) dans un élément HTML. Mais attention : Vue utilise des "moustaches" identiques à celles de Blade, et le code suivant fait lever une exception par Blade :

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

Heureusement, les concepteurs de Blade ont prévu ce cas de figure : un @ devant les moustaches ouvrantes indique à Blade que l'expression ne doit pas être interprétée, et laissée à Vue (ou à tout autre framework).

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

Vue lie dynamiquement les éléments du DOM et les données. Le code ci-dessus affiche « Hello Vue ! ». Lorsqu'on tape en console « app.message » (on se souvient que l'on a déclaré une variable *app* qui contient une instance de Vue), c'est également ce message qui s'affiche.

```
> app.message
< "Hello Vue!"
```

Attribuons maintenant en console une autre valeur à notre variable (3). On peut constater que le contenu de la *div* a changé de façon dynamique.

```
> app.message="Wow Vue !"
< "Wow Vue !"
```

3.2 Les types de variables

Vue utilise les types JS standards :

```
message: true // booléen
messages: ['hello1', 'hello2'] // tableau
messages: [ // JSON (Javascript Object Notation)
            { id:0, content: 'hello0' },
            { id:1, content: 'hello1' }
          ]
```

Le contenu des tableaux et des JSON est accessible selon la syntaxe JS courante :

```
messages[0] // accès à l'élément du tableau d'index 0
messages[0].content // l'élément du tableau d'index 0 est un JSON
                    // qui possède une propriété content
```

3.3 Les directives

Les directives de Vue sont de nouveaux attributs qui s'ajoutent aux attributs natifs des éléments HTML.

3.3.1 Lier des attributs avec la directive *v-bind*

Les moustaches permettent de lier simplement le contenu d'un élément du DOM avec les variables définies dans Vue. En revanche, cette syntaxe ne peut pas être utilisée pour changer les attributs HTML. Pour cela, il faut utiliser une directive *v-bind*, qui s'utilise à l'intérieur des balises ouvrantes comme un attribut, avec la syntaxe *v-bind :attribut="valeur"*. Voici quelques illustrations :

```
<p v-bind:title="message"></p>
    // affiche le contenu de la variable message au survol.
<p v-bind:id="'par_' + id"></p>
    // Si la variable id vaut "1", l'attribut id estest "par_1"
    // Remarquer la concaténation js avec les guillemets simples.
<input type="text" v-bind:disabled="inputDisabled">
    // Si la variable inputDisabled vaut null, undefined ou false,
    // l'attribut n'est pas inclus dans l'élément input.
<input type="text" v-bind:[attrtype]="inputDisabled">
    // Avec les crochets, on peut également lier les attributs à une variable.
    // Par exemple, attrtype == "disabled"
    // Noter qu'un argument entre crochets doit être en minuscules.
<p :title="message"></p>
    // Ecriture abrégée pour v-bind
```

3.3.2 Affichage conditionnel avec la directive *v-if*

Le code suivant insère ou retire l'élément selon l'état du booléen display :

```
<p v-if="display">Hello</p> // Cette ligne apparaît dans le DOM si display == true
```

3.3.3 Ecouter les évènements avec la directive v-on

Nous verrons plus loin comment définir des méthodes avec Vue. Le code ci-dessous appelle une méthode « doSomething » au clic :

```
<a href="#" v-on:click="doSomething">Lien</a>
```

Comme avec v-bind, on peut lier les attributs à une variable :

```
<a href="#" v-on:[event]="doSomething">Lien</a> // p. ex. avec event = "click"

// La directive v-on peut être écrite de façon plus concise :
<a @click="doSomething"></a>
```

3.3.4 Générer une liste avec la directive *v-for*

Nous verrons plus loin comment générer une liste avec *v-for*, qui fonctionne comme l'instruction *for* en JS ou *foreach* en PHP :

```
<li v-for="item in items">@{{ item.content }}</li>
...
items: [
    { content: 'hello0' },
    { content: 'hello1' }
]
```

3.3.5 Générer du HTML avec la directive *v-html*

Les moustaches ont pour particularité d'afficher du texte brut. Le code ci-dessous ne permet donc pas l'interprétation des balises HTML éventuellement contenues dans la chaîne de caractères. La directive v-html résout ce problème.

```
message: '<b>Hello Vue!</b>'
...
<div id="app">
@{{ message }}
    // Les moustaches affichent les caractères bruts : <b>Hello Vue!</b>
<span v-html="message"></span>
    // v-html affiche des caractères gras : Hello Vue!
</div>
```

3.4 Opérations sur les propriétés

Comme l'indique la documentation, on peut utiliser directement la syntaxe JS pour calculer ou modifier la valeur d'une propriété. En termes plus simples, les variables peuvent être calculées ou modifiées avec des fonctions JS à l'intérieur des éléments HTML, et à chaque changement de la propriété, le résultat sera mis à jour.

```
bananas: 0.99
...
<div id="app">@{{ Math.round(bananas) }}
    // la méthode "round" de la classe JS "Math" renvoie la valeur 1
</div>
```

Cette approche a pour inconvénient de compliquer la lecture de la page HTML, notamment avec des opérations longues ou complexes. Mieux vaut alors utiliser les solutions proposées par Vue : les méthodes, les propriétés calculées et les propriétés observées.

3.4.1 Méthodes

Reprenons le calcul ci-dessus en le plaçant dans la méthode *roundBananas* elle-même contenue dans l'objet methods de Vue :

```
@{{ roundBananas() }}
...
var app = new Vue({
    el: '#app',
    data: { bananas: 0.99 },
    methods: {
        roundBananas: function() {
            return Math.round(this.bananas)
        }
    }
});
```

3.4.2 Propriétés calculées

Les méthodes évitent le code verbeux dans les éléments HTML, mais présentent encore un inconvénient : les méthodes sont appelées sur chaque rendu effectué par Vue, ce qui peut être inutilement lourd. C'est pour cela que le framework propose en plus le calcul ou l'observation des propriétés.

A la différence des méthodes, les propriétés calculées sont mises en cache par Vue, et ne sont recalculées que si nécessaire. Dans le code ci-dessous, nous avons remplaçé la méthode par une propriété calculée. Remarquez la suppression des parenthèses à l'intérieur des moustaches : en effet, *roundBananas* n'est plus une méthode, mais bien une propriété dont la valeur est calculée et placée dans le cache avant le rendu [1] Afin de mieux comprendre le fonctionnement d'une propriété calculée, nous avons ajouté une propriété *apples* et un affichage du mot « round » en console à chaque exécution de la closure attribuée à *roundBananas*.

```
@{{ roundBananas }} <br> @{{ apples }}
...
var app = new Vue({
    el: '#app',
    data: { bananas: 1.85, apples: 5 },
    computed: {
        roundBananas: function() {
```

1. La closure qui est attribuée à la propriété *roundBananas* correspond donc à un accesseur. Généralement, on a surtout besoin d'un accesseur sur les propriétés, et dans ce cas on peut directement écrire la closure comme dans l'exemple. Si un mutateur est requis, il faut utiliser une expression plus complète avec get : et set : (v. la documentation).

```
            Console.log('round')
            return Math.round(this.bananas)
        }
    }
});
```

On peut observer en console l'apparition du mot « round » au chargement de la page. En revanche, si on change la propriété *apples* en console, il est intéressant de constater que le mot ne réapparaît pas. Cela est dû à la mise en cache de la valeur de la propriété *bananas*. Comme la propriété *bananas* n'a pas été modifiée, il n'y a pas de raison de la recalculer. Avec une méthode, la propriété *bananas* serait recalculée (et le mot « round » réaffiché) à chaque changement de la propriété *apples*.

De façon générale, il faut préférer les propriétés calculées aux méthodes pour optimiser le rendu.

3.4.3 Propriétés observées

Les propriétés calculées ou les méthodes ne sont pas suffisantes pour gérer les opérations asynchrones d'accès à une API, les opérations très coûteuses en calcul ou avec des données changeantes pour lesquelles il faut réduire la fréquence d'accès. Dans ces cas de figure, il faut utiliser l'option watch qui permet d'oberver les changements d'une propriété. Analysez l'exemple fourni par la documentation officielle de Vue : https://fr.vuejs.org/v2/guide/computed.html#Observateurs

3.5 Exercice

On dispose d'une liste de fruits dans la notation JSON :

```
[
    { nom: "pomme", couleur: "jaune", qpbarquette: 50, barquettes: 0, total: 0, dispo: true },
    { nom: "poire", couleur: "vert", qpbarquette: 40, barquettes: 0, total: 0, dispo: true },
    { nom: "prune", couleur: "violet", qpbarquette: 100, barquettes: 0, total: 0, dispo: true },
    { nom: "mangue", couleur: "orange", qpbarquette: 20, barquettes: 0, total: 0, dispo: true },
]
```

1. Afficher les données dans un tableau pour obtenir ce résultat :

Nom	Couleur	Quantité par barquette	Nombre de barquettes	Total	Disponibilité
pomme	jaune	50	0	0	X
poire	vert	40	0	0	X
prune	violet	100	0	0	X
mangue	orange	20	0	0	X
TOTAL GENERAL				0	

2. Créer une méthode *ajouter* qui incrémente la propriété *barquettes* lorsqu'on clique sur une des cellules de la colonne « Nombre de barquettes ».
3. Afficher la quantité totale d'un type de fruit dans la colonne « Total » (utiliser une méthode *total* qui multiplie la quantité par barquette par le nombre de barquettes).
4. Créer un méthode *inverser* qui inverse le booléen *dispo* lorsqu'on clique sur une des cellules de la colonne « Disponibilité ».
5. Le total général est obtenu avec une propriété calculée *tousfruits*.
6. Ne pas écrire la liste des fruits dans le script : utiliser plutôt une requête Ajax (avec la bibliothèque Axios p. ex.) pour remplir cette liste.

4 Utilisation approfondie de Vue

4.1 Liaisons de classes et de styles

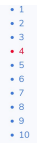
Précédemment, nous avons utilisé la directive *v-bind* pour lier des attributs à l'intérieur d'un élément HTML. Dans une balise HTML, *class* et *style* sont des attributs, ils peuvent donc être liés à des variables :

```
<div v-bind:class="classes">Hello !</div>
...
var app = new Vue({
    el: '#app',
    data: {
        classes: {
            'text-danger': true,
            'font-weight-bold': true
        }
    }
});
```

Cette partie ne présente pas de difficulté particulière, nous renvoyons le lecteur à la documentation pour voir quelles sont les différentes syntaxes pour ces types de liaison : https://fr.vuejs.org/v2/guide/class-and-style.html

4.1.1 Exercice sur les liaisons de classes

Avec Vue, générer une liste non ordonnée qui affiche la liste des chiffres de 1 à 10. Ces chiffres sont générés par Vue. Au chargement de la page, le premier item possède une classe Bootstrap « text-danger » (rouge), et les autres une classe « text-primary » (bleu). Lorsqu'on clique sur un des items, celui-ci devient rouge, et l'ancien item rouge (re)devient bleu. Cet exercice peut être résolu sans définir de méthodes ou de propriétés calculées.



4.2 Liaisons sur les champs de formulaire

Nous avons vu que *v-bind* liait des variables définies dans l'objet *data* de Vue à des attributs, et que les moustaches faisaient de même avec du contenu. Pour les éléments de formulaire, nous disposons de la directive *v-model*. Cette directive a la particularité de créer une **liaison bidirectionnelle adaptée à l'élément**.

4.2.1 Liaison bidirectionnelle

Considérons l'exemple ci-dessous. Au chargement de la page, l'élément *input* et les moustaches affichent 43. C'est également la valeur de l'attribut *value* de l'*input* (attribut qu'il est inutile d'ajouter lorsqu'on utilise *v-model*).

```
<input type="text" name="pointure" v-model="pointure"> // affiche 43
<div>@{{ pointure }}</div>                             // affiche 43
...
pointure: 43
```

Si l'utilisateur saisit au clavier un autre contenu dans l'*input*, les moustaches mettent à jour instantanément la valeur de la variable *pointure* : c'est une liaison bidirectionnelle.

4.2.2 Liaison adaptée à l'élément

Comme nous venons de la voir, dans un *input* de type text, la variable indiquée dans *v-model* correspond à l'attribut *value*. En fait, *v-model* s'adapte à l'élément de formulaire qui le contient :

Elément	Type	Multiple	*v-model*
<input>	text		*value*
<input>	checkbox		*checked*
<input>	checkbox	x	[*values*]
<input>	radio	(x)	*selected*
<select>			*value*
<select>		x	[*values*]

Dans l'exemple suivant, plusieurs éléments *input* de type checkbox sont liés à la même variable *chaussures*. Cette variable est définie au chargement de la page comme un tableau vide, et l'élément *label* affiche donc « [ ] ». Lorsqu'on coche une ou plusieurs cases, l'affichage est mis à jour avec les values des éléments. Avec toutes les cases cochées, on obtient « ['mocassins', 'derby', 'baskets'] »

```
<input type="checkbox" v-model="chaussures" value="mocassins">
<input type="checkbox" v-model="chaussures" value="derby">
<input type="checkbox" v-model="chaussures" value="basket">
<label for="checkbox">@{{ chaussures }}</label>
...
chaussures: []
```

4.2.3 Modificateurs

Le comportement des directives peut être changé grâce à des modificateurs. Voici par exemple deux modificateurs utiles associés à *v-model* :

Modificateur	Description
v-model.trim	Retire les espaces superflus des données saisies
v-model.number	Convertit les données saisies en nombre [2]

4.3 Rendu conditionnel

Nous avons déjà vu plus haut la directive *v-if* pour le rendu conditionnel. De cette section de la documentation, il faut retenir les points suivants :

- *v-if* peut être utilisé avec les directives ***v-else*** et *v-else-if*.
- ***v-show*** est presque similaire à *v-if*. Mais v-show garde les éléments à ne pas afficher dans le DOM (basculement du CSS en *display:none*), alors que *v-if* les supprime vraiment. De plus, *v-show* ne fonctionne pas avec *<template>* et *v-else*.
- Il faut **éviter d'utiliser *v-if* avec *v-for***.

4.4 Rendu de liste

Nous avons déjà vu plus haut la directive *v-for* pour le rendu de liste. De cette section de la documentation, il faut retenir les points suivants :

- Forme de base, performante :

  ```
  <li v-for="item in items">
  ```

- Forme avec index 0,1,2... :

  ```
  <li v-for="(item, index) in items">
  ```

- **Forme recommandée, un peu moins performante mais plus sûre, qui implique de fournir un id pour chaque item** :

2. Par défaut, les données sont des chaînes de caractères.

```
<li v-for="item in items" v-bind:key="item.id">
...
items: [
        { id:0, content: 'hello0' },
        { id:1, content: 'hello1' }
]
```

- La forme suivante est possible :

```
<li v-for="n in 10">  // n vaut successivement 1,2 .. 10
```

4.4.1 Méthodes de tableau

Certaines méthodes JS permettant de modifier les tableaux sont surchargées[3] par Vue afin de déclencher des mises à jour dans l'affichage. Elles sont dites **mutatives**, c'est-à-dire qu'elles modifient réellement le tableau. Les autres méthodes JS, plus nombreuses, comme *filter()* ou *slice()*, ne sont pas mutatives et renvoient un nouveau tableau. Pour que Vue détecte un changement dans un tableau, il faut donc réattribuer sa variable avec le nouveau tableau renvoyé par ces méthodes. Cela n'altère en rien les performances de Vue.

Méthode	Mutative	Description
push()	x	Ajouter un ou plusieurs éléments à la fin du tableau
pop()	x	Supprimer le dernier élément du tableau
shift()	x	Ajouter un ou plusieurs éléments au début du tableau
unshift()	x	Supprimer le premier élément du tableau
splice()	x	Insérer/supprimer un ou plusieurs éléments à un endroit précis du tableau
sort()	x	Trier le tableau
reverse()	x	Inverser les éléments du tableau
filter()		Filtre le tableau selon une expression rationnelle
slice()		Renvoie une partie du tableau
...		

4.4.2 Limitations dans la détection de changement dans un objet

Vue ne détecte pas les ajouts ou les suppressions directes de propriétés effectués sur un objet. Il existe heureusement une solution de contournement avec la méthode *Vue.set()* :

```
app.user.age = 33             // on ajoute une propriété 'age' à l'objet user :
                              // cela ne provoque pas de mise à jour de l'objet
Vue.set(app.user, 'age' , 33) // contournement, l'ajout est pris en compte
```

Ce problème concerne particulièrement les tableaux. Si on souhaite attribuer une valeur à un index donné, il faut utiliser *Vue.set()* :

```
app.items[1] = 42         // ne provoque pas de mise à jour
Vue.set(app.items, 1, 42) // contournement, le changement est pris en compte
```

4.4.3 Tri et filtrage

Le tri et le filtrage de tableaux sont des fonctionnalités importantes dans une application, qui ne peuvent être rendues de façon réellement fluide que du côté client. Pour effectuer le tri ou le filtrage, on utilise des méthodes ou, de préférence, des **propriétés calculées**.

```
<li v-for="nom in noms">{{ item }}</li>
    // affiche tous les noms
<li v-for="nom in filtreNoms(noms)">{{ item }}</li>
    // affiche seulement les noms renvoyés par la méthode filtreNoms()
<li v-for="nom in filtreNoms">{{ item }}</li>
    // affiche seulement les noms renvoyés par la propriété calculée filtreNoms
```

3. Surcharger une fonction (ou une méthode) consiste à ajouter des fonctionnalités à une fonction préexistante.

Par exemple, la propriété calculée ci-dessous renvoie les *n* premiers noms du tableau *noms* :

```
<li v-for="nom in filtreNoms">{{ nom }}</li>
...
computed: {
    filtreNoms: function () {
        return this.noms.slice(0, this.n)
    }
},
...
```

On peut aussi altérer le tableau en utilisant des méthodes mutatives (v. 4.4.1). Dans ce cas, le rendu de liste peut être effectué directement sur le tableau, et il n'est plus nécessaire de définir une propriété calculée ou une méthode. Le rendu est mis à jour à chaque changement :

```
<tr v-for="(item, key) in liste" v-on:click="liste.splice('key', 1)">
```

4.5 Gestion des évènements

Cette partie ne présente pas de difficulté particulière, nous renvoyons le lecteur à la documentation : https://fr.vuejs.org/v2/guide/events.html

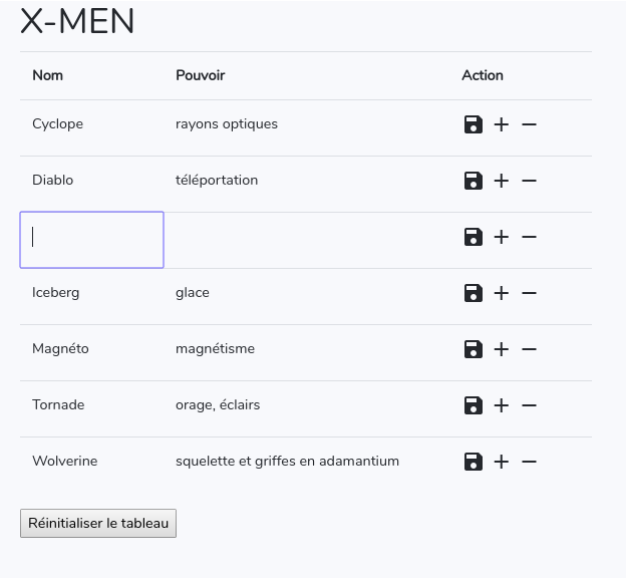
4.6 Exercices

4.6.1 Mutants éditables

On dispose d'un tableau JSON. Chaque élément du tableau comporte deux propriétés, « nom » et « pouvoir ».

```
[
    { nom: "Cyclope", pouvoir: "rayons optiques" },
    { nom: "Iceberg", pouvoir: "glace" },
    { nom: "Diablo",  pouvoir: "téléportation" },
    { nom: "Wolverine", pouvoir: "squelette et griffes en adamantium" },
    { nom: "Tornade", pouvoir: "orage, éclairs" },
    { nom: "Magnéto", pouvoir: "magnétisme" }
]
```

1. Disposer les données dans un tableau pour obtenir un résultat trié au chargement de la page :



2. Un clic sur « – » supprime le mutant concerné de la liste.
3. Un clic sur le bouton de réinitialisation recrée le tableau initial (sans rechargement de la page).
4. Un clic sur « + » ajoute un mutant, avec des propriétés vides au départ, mais éditables. On peut également éditer les propriétés des autres mutants.

5. Au relâchement de la touche  , les données éditées sont ajoutées ou mises à jour dans l'objet contenant la liste des mutants. La même action peut être réalisée avec l'icône figurant une disquette.
6. Lorsque l'on clique sur les entêtes « Nom » ou « Pouvoir » du tableau, la liste est triée par ordre alphabétique en fonction de la colonne choisie. Le premier clic déclenche un tri ascendant, le deuxième un tri descendant et ainsi de suite... Le tri doit être insensible à la casse.

Indices

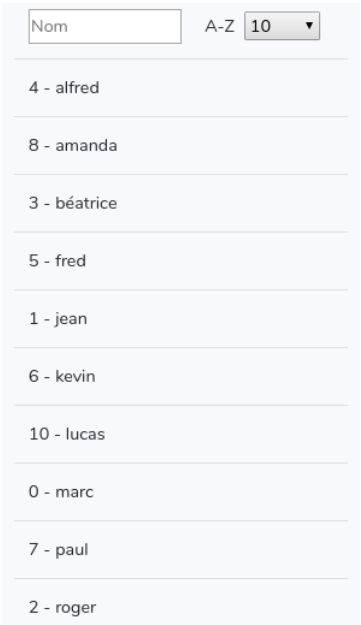
- Le tri est obtenu avec la méthode mutative *sort()*.
- Utiliser l'attribut *conteneditable* (ne pas utiliser l'élément *<input>*).

4.6.2 Filtrage de noms

On dispose d'un tableau JSON. Chaque élément du tableau comporte deux propriétés, « id » et « nom ».

```
[
    {  id: 0, nom:"marc" },
    {  id: 1, nom:"jean" },
    {  id: 2, nom:"roger" },
    {  id: 3, nom:"béatrice" },
    {  id: 4, nom:"alfred" },
    {  id: 5, nom:"fred" },
    {  id: 6, nom:"kevin" },
    {  id: 7, nom:"paul" },
    {  id: 8, nom:"amanda" },
    {  id: 9, nom:"thierry" },
    {  id: 10, nom:"lucas" },
    {  id: 11, nom:"serge" },
]
```

1. Disposer les données dans un tableau pour obtenir un résultat trié au chargement de la page :



2. Un clic sur « A-Z » inverse l'ordre du tableau, et « A-Z » devient « Z-A ».
3. Le *select* limite l'affichage à 5, 10 ou 15 éléments (10 par défaut).
4. L'*input* de type text permet de filtrer les noms, sans tenir compte ni de la casse, ni des espaces de début et de fin.

Indices

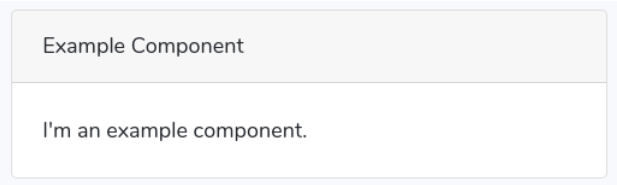
- Le filtrage est effectué par imbrication de closures utilisant les fonctions *filter()* et *match()*.
- *match()* implique l'utilisation du constructeur *RegExp*.

5 Les composants

5.1 Principes de base

Laravel intègre un composant Vue par défaut, défini dans le fichier `resources/js/components/ExampleComponent.vue`, et déclaré dans le fichier `resources/js/app.js`. Cela signifie qu'on peut utiliser ce composant dans un élément Vue comme s'il s'agissait d'un élément HTML :

```
<div id="app">
    <example-component></example-component>
</div>
...
var app = new Vue({ el: '#app' });
```



Utiliser des composants est une bonne pratique, car cela permet la fragmentation du code et sa réutilisation. Comme le montre l'exemple fourni dans Laravel, chaque composant peut faire l'objet d'une attribution à une variable, ou être détecté automatiquement. Il faut cependant recompiler avec *npm* à chaque ajout ou changement.

```
// fichier app.js

// SOLUTION 1 : déclarer chaque composant
Vue.component('example-component', require('./components/ExampleComponent.vue').default);

// SOLUTION 2 : détecter automatiquement les composants dans l'arborescence
const files = require.context('./', true, /\.vue$/i);
files.keys().map(key => Vue.component(key.split('/').pop().split('.')[0], files(key).default));
```

Pour commencer, nous écrirons nos composants directement dans la vue Blade.

Prenons l'exemple d'un compteur rudimentaire :

```
<button v-on:click="count++">Vous m'avez cliqué @{{ count }} fois.</button>
...
var app = new Vue({
    el: '#app',
    data: {
        count: 0
    }
});
```

On souhaiterait pouvoir réutiliser ce compteur dans notre application avec un composant que nous pourrions baptiser *<super-button>* :

```
<div id="app">
    <super-button side="gauche" />
    <super-button side="droite" />
</div>

<script>
Vue.component('super-button', {
  data: function () {
    return {
      count: 0
    }
  },
  template: '<button v-on:click="count++">Vous m\'avez cliqué @{{ count }} fois.</button>'
})

var app = new Vue({
    el: '#app'
});
</script>

</body>
```

Il y a plusieurs choses à observer dans cette nouvelle approche par composant :

- le nom du composant doit suivre de préférence la convention de nommage **kebab case**.
- les variables contenues dans l'élément *data* doivent être retournées par une **fonction** (dans le cas contraire, une exception est levée).
- l'élément *template* contient le code HTML brut du composant.
- lorsque le composant ne fait pas appel à ce qui est contenu entre des balises ouvrantes et fermantes (*<super-button></super-button>*), on peut peut utiliser des **balises orphelines** (*<super-button />* ou *<super-button>*).

5.2 Création globale, création locale et système de module

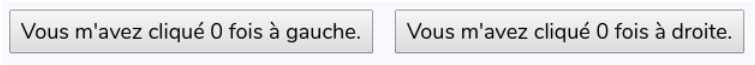
La méthode Vue.component() permet de définir des composants **globalement**, ce qui signifie qu'ils sont disponibles dans n'importe quelle instance de Vue, et même **entre eux** (v. 5.3.2). Avec *webpack*, les composants sont rangés dans des scripts séparés et des sous-dossiers. Ils sont ensuite importés et définis globalement.

5.3 Les *props*

L'élément *props* définit la transmission de données vers le composant. Dans l'exemple suivant, la *props side* (« côté ») est définie dans le composant *<super-button>*, et est présente dans l'élément *template*. *side* devient alors une sorte d'attribut dans l'élément HTML.

```
<div id="app">
    <super-button side="gauche" />
    <super-button side="droite" />
</div>

<script>
Vue.component('super-button', {
  props: ['side'],
  data: function () {
    return {
      count: 0
    }
  },
  template: '<button v-on:click="count++">
            Vous m\'avez cliqué @{{ count }} fois à @{{ side }}.
            </button>'
})
...
```



Lorsque les données sont contenues dans l'instance de Vue, on peut utiliser *v-bind* sur les *props* du composant :

```
<div id="app">
    <super-button v-for="side in sides" v-bind:side="side" />
</div>

<script>
Vue.component('super-button', {
  props: ['side'],
  data: function () {
    return {
      count: 0
    }
  },
  template: '<button v-on:click="count++">
            Vous m\'avez cliqué @{{ count }} fois à @{{ side }}.
            </button>'
})

var app = new Vue({
    el: '#app',
    data: {
```

```
        sides: ['gauche', 'droite']
    }
});
```

5.3.1 La balise *<slot>*

Les balises *<slot></slot>* permettent d'injecter dans l'élément *template* d'une composant le contenu de la balise présente dans le DOM.

```
<div id="app">
    <super-p name="Oscar">Je m'appelle</super-p>
</div>
...
Vue.component('super-p', {
    props: ['name'],
  template: "<p><slot></slot> @{{ name }}</p>"
})
...
```

Je m'appelle Oscar

5.3.2 Composants imbriqués

Comme nous l'avons indiqué précédemment, les composants peuvent faire appel à d'autres composants. Dans l'exemple ci-dessous, l'élément *template* du composant *<super-p>* utilise le composant *<super-span>*.

```
<div id="app">
    <super-p name="Oscar">Hello</super-p>
</div>
...
Vue.component('super-p', {
  props: ['name'],
  template: "<p><slot></slot> <super-span>@{{ name }}</super-span></p>"
})

Vue.component('super-span', {
  template: "<b><i><slot></slot></i></b>"
})
...
```

Hello *Oscar*

5.3.3 Types des *props*

<>

5.4 Exercices

5.4.1 Création d'un tableau adaptable

Créer un composant <adaptable> prenant en *props* un objet *content* de contenu variable et un booléan *stripped* correspondant à l'activation de la classe Bootstrap *table-striped*. La première ligne de l'objet *content* contient les noms de colonne du tableau. Le composant doit adapter le nombre de colonnes et de lignes du tableau en fonction de *content*.

```
<div id="app">
    <adaptable :content="content" striped />
</div>
...
content: [
    {  id: 4, titre:"Tintin chez les codeurs", year:2017 },
    {  id: 8, titre:"Lagaffe fait des bugs", year:2015 },
    {  id: 0, titre:"Martine démonte un PC", year:1999 },
]
...
```

id	titre	year
4	Tintin chez les codeurs	2017
8	Lagaffe fait des bugs	2015
0	Martine démonte un PC	1999