

Introduction à Vue.js 3

Titre

Réaliser une application web étape par étape afin de comprendre les notions de base d'une application réalisée avec **vue.js**.

Compétence

Implémenter, au moyen de la technologie définie, le front-end d'une application Web interactive permettant la gestion de données.

Table des matières

1. Introduction à Vue 3.....	5
2. Création de l'application Vue.....	6
2.1 Présentation du code de départ.....	6
2.2 Création d'une application Vue.....	7
2.3 Monter notre application.....	8
2.4 Affichage des données.....	8
2.5 Comprendre l'instance Vue.....	9
2.6 Réactivité de Vue	10
2.7 A vous de coder !.....	12
3. La liaison d'attributs (Attribute Binding)	13
3.1 Notre objectif.....	13
3.2 Ajout d'une image à nos données.....	13
3.3 Introduction à la liaison de l'attribut.....	14
3.4 Comprendre la directive v-bind	15
3.5 Une obligation de réactivité	16
3.6 Un raccourci pour la directive v-bind	16
3.7 A vous de coder !.....	17
4. Rendu conditionnel	18
4.1 Notre objectif.....	18
4.2 Afficher ou ne pas afficher.....	18
4.3 La directive "v-if"	19
4.4 La directive v-show.....	19
4.5 Logique conditionnelle.....	20
4.6 A vous de coder !.....	21
5. Rendu de liste	22
5.1 Notre objectif.....	22
5.2 Parcourir un tableau de données.....	22
5.3 Variantes de couleur pour chaque produit	23

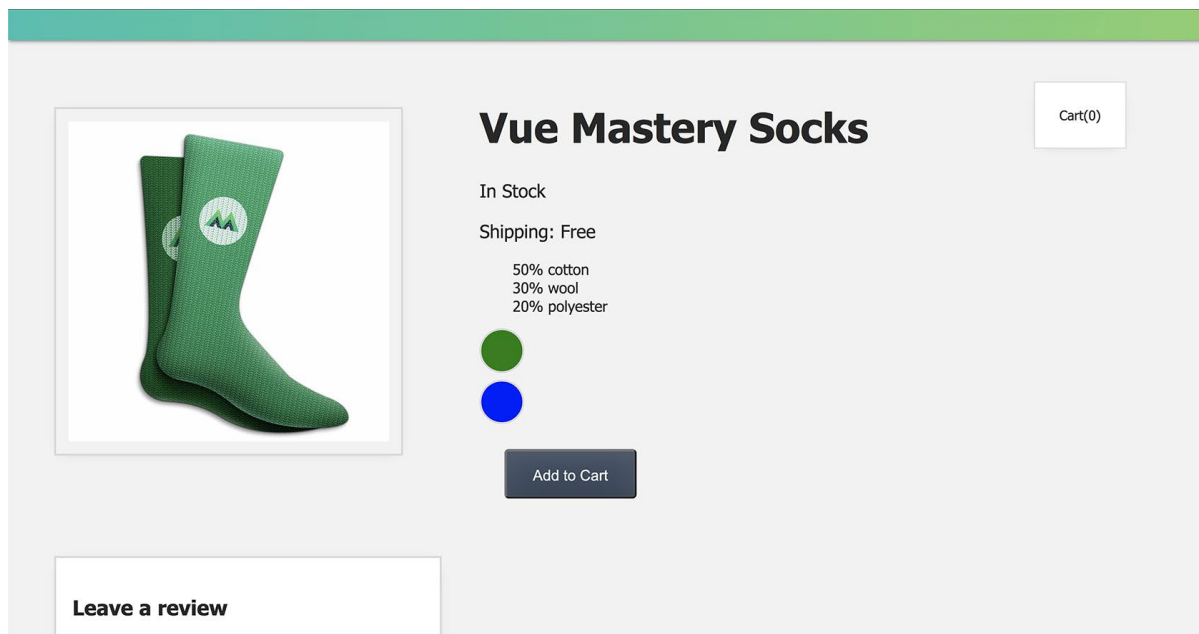
5.4	Attribut Key : un élément essentiel pour les éléments de la liste.....	24
5.5	A vous de coder !.....	25
6.	Gestion des événements.....	26
6.1	Notre objectif	26
6.2	Écouter les événements : La directive v-on	26
6.3	Déclenchement d'une méthode.....	27
6.4	Comprendre la directive v-on.....	28
6.5	Un raccourci pour la directive v-on.....	28
6.6	Un autre exemple : Événements de la Souris	29
6.7	A vous de coder !.....	31
7.	Liaison de Classe et liaison de Style	32
7.1	Notre objectif	32
7.2	Liaison du style.....	32
7.3	Comprendre la liaison de style	34
7.4	Camel vs Kebab.....	35
7.5	Liaison de style : Objets.....	36
7.6	Liaison de classe.....	36
7.7	Noms de classe multiples.....	39
7.8	Opérateurs ternaires.....	39
7.9	A vous de coder !.....	40
8.	Propriétés calculées (Computed Properties)	41
8.1	Notre objectif	41
8.2	Une propriété calculée simple.....	41
8.3	Comme une calculatrice	42
8.4	Calcul de l'image et de la quantité	43
8.5	A vous de coder !.....	48
9.	Composants et Props	49
9.1	Notre objectif	49
9.2	Blocs de construction d'une application Vue.....	49

9.3	Création de notre premier composant.....	50
	Template	51
	Données et méthodes.....	53
	Nettoyage de main.js	54
	Importation du composant.....	55
9.4	Props	57
9.5	Donner un Props à notre composant.....	58
9.6	Utilisation de Props	60
9.7	A vous de coder !.....	61
10.	Communiquer grâce aux événements.....	62
10.1	Notre objectif	62
10.2	Émettre l'événement.....	62
10.3	Ajouter l'identifiant du produit au panier.....	65
10.4	A vous de coder !.....	67
11.	Formulaires et modèle v.....	67
11.1	Notre objectif	67
11.2	Introduction à la directive v-model.....	68
11.3	Le composant ReviewForm	68
	Soumettre le formulaire ReviewForm	71
	Utilisation du formulaire de ReviewForm.....	72
	Ajout d'avis des produits.....	75
11.4	Affichage des avis/critiques.....	76
11.5	Validation du formulaire	80
11.6	A vous de coder !.....	81
12.	Conclusion.....	81

1. Introduction à Vue 3

Nous allons faire un voyage à travers l'univers de **Vue.js** pour explorer la technologie et construire une base solide de nouvelles compétences.

Tout au long de ce cours, nous apprendrons les bases de Vue.js et nous allons construire une application pour mettre ces concepts en pratique.



Pour tirer le meilleur parti de ce cours, vous allez devoir coder. Vous allez devoir récupérer le [dépôt git du cours](#) et cloner le projet. À partir de là, vous pouvez basculer entre les différentes branches pour récupérer le code de départ et/ou le code de fin de chaque leçon.

Une autre possibilité est de se référer directement au dépôt depuis github et de copier/coller le code à chaque étape. Pour naviguer facilement entre les différents fichiers du dépôt, je vous conseille d'installer l'extension navigateur [Octotree – Github](#) disponible sur chaque navigateur.

Par exemple, dans la leçon suivante, vous pouvez récupérer la branche git qui a pour nom L2-start pour le code de début puis vous pouvez obtenir la solution grâce à la branche L2-end.

Tout au long de ce cours, vous allez utiliser l'éditeur VS Code. Vous devez aussi installer une extension [es6-string-html](#). Vous comprendrez dans les leçons suivantes en quoi cette extension est utile.

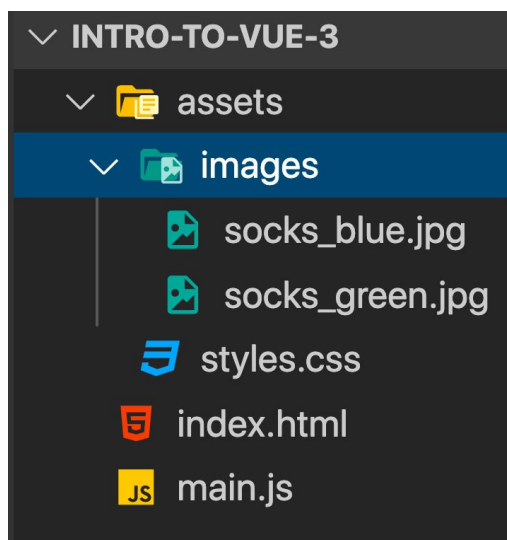
Enfin, à la fin de chaque leçon, un petit défi de code vous sera proposé afin que vous puissiez mettre les concepts en pratique. Maintenant, si vous êtes prêts, créons notre première application Vue dans la prochaine leçon.

2. Création de l'application Vue

Etes-vous prêt à créer une application Vue ? Pour commencer avec cette leçon, consultez le code de départ sur la L2-start branche du [repo](#).

2.1 Présentation du code de départ

Dans le code de départ, vous trouverez un répertoire **assets**. À l'intérieur, il y a un répertoire pour les images. Nous en avons une pour les chaussettes bleues, et une pour les chaussettes vertes. Nous avons aussi un fichier CSS pour tous nos styles : le fichier **style.css**.



À l'intérieur du fichier **index.html**, nous importons ces styles. En dessous, nous importons la bibliothèque Vue.js. L'importation de la bibliothèque Vue via un lien CDN est le moyen le plus simple de commencer à utiliser Vue. Vous remarquerez que j'utilise le lien CDN de la version bêta de Vue 3, mais au moment où vous,

vous suivez ce cours, Vue 3 est sorti depuis quelques temps déjà. Vous pouvez donc vous rendre sur [la documentation officielle de Vue.js](#) et copier le lien CDN de la dernière version de Vue 3 et le remplacer dans votre **fichier index.html**.

TODO : A vous de mettre à jour le CDN en vous référant à la doc officielle de Vue.js.

En bas du fichier, nous importons notre fichier **main.js**, qui est pour l'instant assez simple : juste un `const product = 'Socks'`

```
JS main.js ×
JS main.js > [?] product
1  const product = 'Socks'
2
```

Vous remarquerez que dans le modèle, il y a un `h1` « Product goes here ». La question est donc maintenant : **comment afficher le produit en utilisant Vue ?**

2.2 Création d'une application Vue

Pour afficher nos données dans notre HTML, nous devons d'abord créer une application Vue. Dans notre fichier **main.js**, nous allons créer notre application avec :

- **main.js**

```
const app = Vue.createApp({});
```

En argument, nous allons passer un objet et ajouter une propriété aux données. Ce sera une fonction qui retourne un autre objet, où nous stockerons nos données. Ici, nous allons ajouter `product` en tant que données.

- **main.js**

```
const app = Vue.createApp({
  data() {
    return {
```

```
      product: 'Socks'
    }
  }
})
```

Maintenant, nous devons nous assurer que nous importons notre application Vue dans le fichier `index.html`.

`index.html`

```
<!-- Import App -->
<script src="./main.js"></script>
```

2.3 Monter notre application

Maintenant que nous avons créé notre application, nous devons la monter dans le DOM. Nous allons le faire à l'intérieur d'une balise de script, dans notre fichier `index.html`.

`index.html`

```
<!-- Mount App -->
<script>
  const mountedApp = app.mount('#app')
</script>
```

Nous allons utiliser `app`, qui fait référence à l'application que nous venons de créer, et ensuite `.mount()`, qui est une méthode qui nécessite un sélecteur DOM comme argument. Cela permet de brancher l'application Vue dans cet élément du DOM.

2.4 Affichage des données

Maintenant que nous avons créé, importé et monté l'application Vue, nous pouvons commencer à afficher les données qui « vivent » en son sein.

Pour afficher le `product` à l'intérieur de la `h1`, nous allons écrire :

index.html

```
<div id="app">
  <h1>{{ product }}</h1>
</div>
```

Maintenant, si nous vérifions dans le navigateur, nous verrons que le « Produit » s'affiche. C'est super. Mais comment cela fonctionne-t-il exactement ?

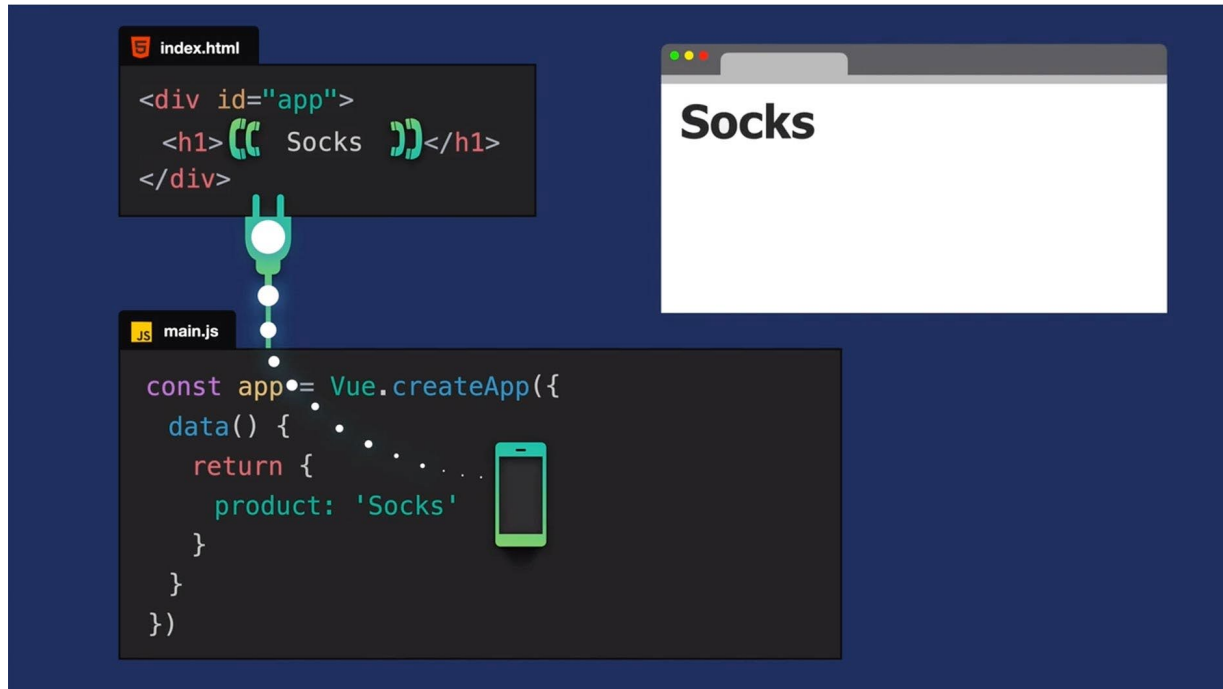
2.5 Comprendre l'instance Vue

Lorsque nous avons créé notre application Vue, nous avons transmis l'objet **options**, ce qui nous a permis d'ajouter des propriétés facultatives pour configurer l'application. Cela crée notre instance Vue, le cœur de notre application Vue, qui alimente le tout.

- **main.js**

```
const app = Vue.createApp({Options Object})
```

En l'important et en l'affichant sur le DOM, nous avons essentiellement branché l'application dans notre DOM, donnant à notre HTML une ligne directe dans l'application. De cette façon, notre code modèle peut accéder à des options à partir de l'application, telles que ses données.



Si vous vous demandez ce qui se passe avec cette syntaxe, vous pouvez l'imaginer comme un téléphone, qui a accès à un autre téléphone dans notre application Vue. À partir de notre template, nous pouvons demander à l'application « Hé, quelle est la valeur du produit ? » Et l'application répond : « Chaussettes ». Lorsque la page s'affiche, nous voyons « Chaussettes » s'afficher sur la page.

Si cette syntaxe à double {{ }}, ou syntaxe de moustache, est nouvelle pour vous, cela nous permet d'écrire des expressions JavaScript. En d'autres termes, il nous permet d'exécuter un JavaScript valide dans notre HTML.

2.6 Réactivité de Vue

Que se passerait-il si nous modifions la valeur de product de "Cocks" à "Bottes" ?

- **main.js**

```
const app = Vue.createApp({
  data() {
    return {
```

```
        product: 'Boots' // updated data value //  
      }  
    }  
  })
```

En raison de la façon dont Vue agit, l'expression `h1` qui s'appuie sur `product` recevrait automatiquement cette nouvelle valeur, et notre DOM serait mis à jour pour afficher « Bottes ».

index.html

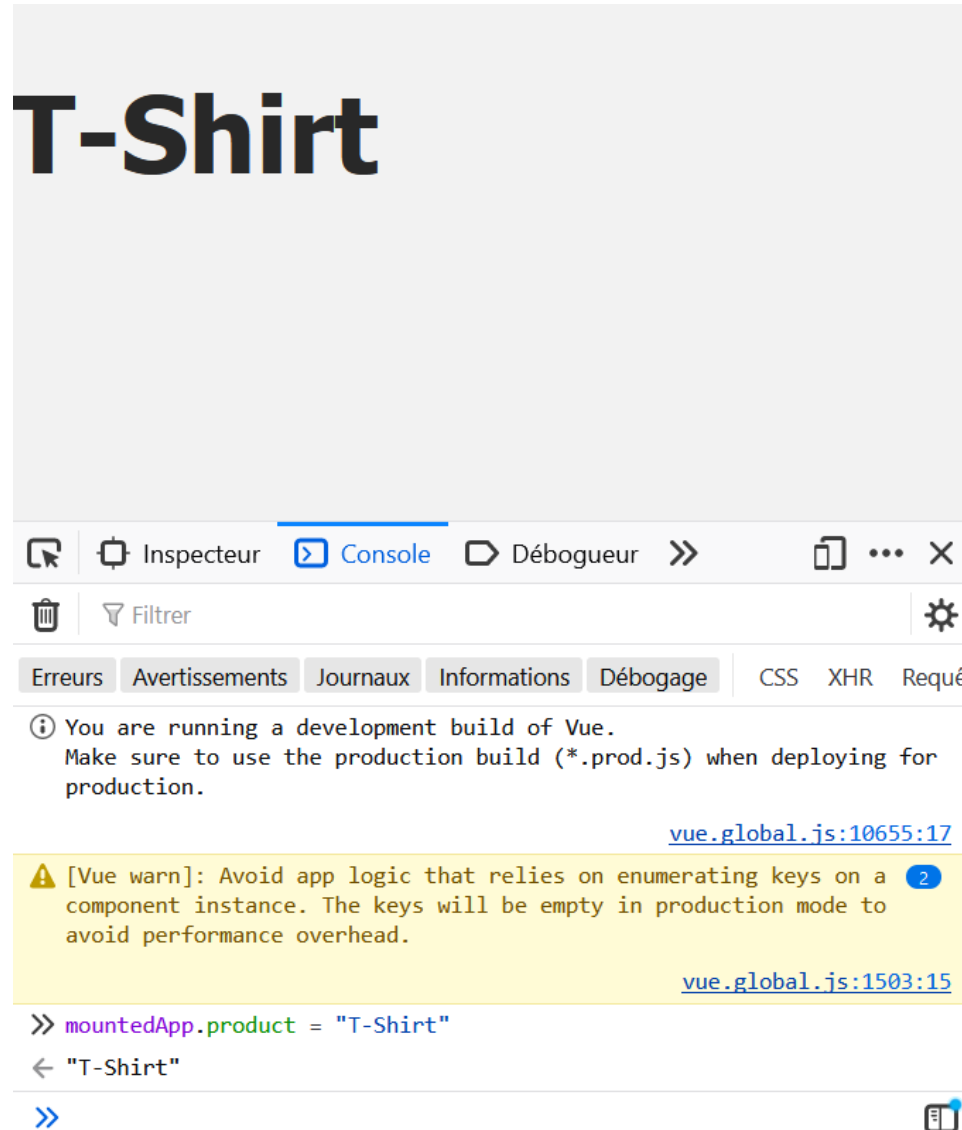
```
<div id="app">  
  <h1>{{ product }}</h1> <!-- will reactively receive any  
updates to product -->  
</div>
```

C'est parce que Vue est *réactive*. Sous le capot, Vue dispose d'un système de réactivité entier qui gère les mises à jour. Lorsqu'une valeur de données change, le tout sera automatiquement mis à jour pour nous. Nous n'avons rien à faire pour y parvenir.

Pour se convaincre de la réactivité de l'application grâce à Vue, essayez de lancer l'inspecteur de votre navigateur puis dans l'onglet console, saisissez le code :

```
mountedApp.product = "T-Shirt"
```

Comme ci-dessous :



La variable `mountedApp` a été définie dans **index.html**

2.7 A vous de coder !

Nous avons atteint la fin de cette leçon, qui nous amène au premier défi de codage. Vous pouvez trouver le code de la solution en vérifiant la branche L2-end du [repo](#).

Ajouter une description à l'objet de données.

Affichez la description à l'intérieur d'un tag p.

Ce tag p doit s'afficher en dessous de la balise h1.

3. La liaison d'attributs (Attribute Binding)

Dans cette leçon, nous allons étudier le concept de liaison des attributs. Récupérer le code de la [branche](#) L3-start.

3.1 Notre objectif

Dans le code de départ, nous avons un nouveau div avec la classe product-image.

index.html

```
<div class="product-image">
  <!-- image goes here -->
</div>
```

À la fin de la leçon, nous allons avoir un élément HTML img qui est lié de manière réactive à une nouvelle propriété image sur nos données. Chaque fois que la valeur de cette image change, notre image va être mise à jour dans le DOM.

3.2 Ajout d'une image à nos données

Rappelez-vous que dans notre répertoire **assets** nous avons un dossier d'**images**, avec des images pour les chaussettes vertes et bleues. Nous allons cibler l'une de ces images grâce à une nouvelle propriété de données sur notre application Vue. Nous le ferons en fixant image à un chemin pour qu'il puisse définir cette image.

- main.js

```
const app = Vue.createApp({
  data() {
    return {
      product: 'Socks',
```

```
        image: './assets/images/socks_green.jpg'
      }
    }
  })
```

Maintenant, nous sommes prêts à ajouter un élément `img` dans le HTML.

index.html

```
<div class="product-image">
  
</div>
```

Dans l'attribut `source`, nous avons mis `image`. À l'heure actuelle, cela ne fera rien. Nous devons définir dans l'attribut `src` le chemin de l'image à partir de nos données, de la même manière dont nous avons tiré la valeur de `product` des données pour l'expression JS dans le `h1` de la leçon précédente.

Donc la question est : **comment lier l'attribut `src` à l'image des données ?**

3.3 Introduction à la liaison de l'attribut

Pour créer un lien entre l'attribut d'un élément HTML et une valeur à partir des données de votre application Vue, nous utiliserons une directive Vue appelée **`v-bind`**.

index.html

```

```

Maintenant, nous avons créé un lien réactif entre ce qui vit dans cet attribut ("`image`") et l'image des données elles-mêmes.

En regardant dans le navigateur, nous allons maintenant voir notre image de chaussettes vertes qui s'affiche.



3.4 Comprendre la directive v-bind

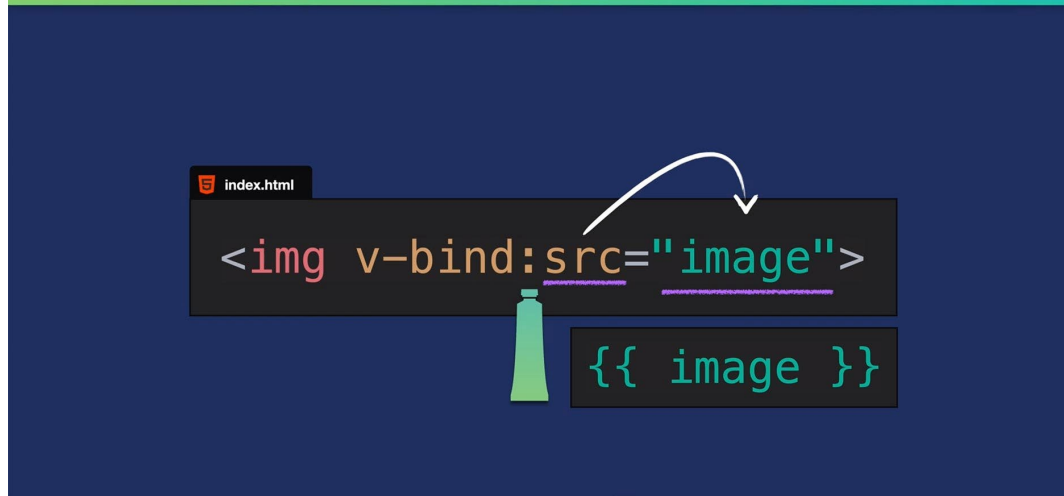
Comment la directive v-bind fonctionne-t-elle ? Nous utilisons cette directive pour lier dynamiquement un attribut à une expression. Dans ce cas, l'attribut est `src` et l'expression est ce qui est dans les données à savoir de l'attribut : `"image"`

index.html

```
 <!-- src attribute bound to the  
image data -->
```

Si vous pensez que cela ne ressemble pas à une expression JavaScript typique, vous pouvez imaginer qu'elle ressemble à ceci : `v-bind:src="{{ image }}"`. Sous le capot, Vue va l'évaluer.

v-bind: Dynamically bind an attribute to an expression



3.5 Une obligation de réactivité

En raison du système de réactivité de Vue, si nous mettons à jour notre donnée `image` vers un chemin qui pointe vers l'image de nos chaussettes bleues (`image: './assets/images/socks_blue.jpg'`), notre attribut `src` qui est lié sera mis à jour et notre navigateur afficherait l'image des chaussettes bleues.

A vous d'essayer en faisant ce petit changement !

3.6 Un raccourci pour la directive v-bind

L'utilisation de `v-bind` est très fréquente - si fréquente qu'il y a un raccourci pour ça :

```

```

Comme vous pouvez l'imaginer, puisqu'il y a de nombreux attributs HTML différents, il y a de nombreux cas d'utilisation pour `v-bind`. Par exemple, vous pouvez lier une description à un attribut `alt`, lier une URL à une `href`, en liant certains styles dynamiques à une `class` ou attribut `style`, désactiver et activer un bouton, et ainsi de suite.

3.7 A vous de coder !

Nous avons atteint la fin de la leçon et nous sommes prêt pour le nouveau défi :

Ajouter un attribut `url` à l'objet de données

Vous devez lier (bind) cette `url` à un attribut `href`

Le lien doit s'afficher en dessous du `<h1>`

Vous pouvez par exemple prendre l'URL du site de l'ETML <https://www.etml.ch/>

Pour rappel, vous pouvez vérifier le code dans la [branche](#) L3-end.

4. Rendu conditionnel

Dans cette leçon, nous allons étudier le concept de rendu conditionnel.

Vous pouvez trouver le code de départ dans la [branche](#) L4-start.

4.1 Notre objectif

Nous voulons afficher différents éléments HTML basés sur une condition. Nous allons afficher un tag `p` qui dit "en stock" quand notre produit est en stock, ou qui dit "en rupture de stock" quand il n'y a plus de produits correspondant.

4.2 Afficher ou ne pas afficher

Dans notre fichier **index.html**, nous allons ajouter deux nouveaux tags `p`.

index.html

```
<p>In Stock</p>
<p>Out of Stock</p>
```

Nous voulons que seulement l'un des deux apparaisse selon que notre produit est en stock ou non. Donc nous allons aller dans l'objet de données de notre application Vue et ajouter un nouveau booléen `inStock`.

- **main.js**

```
const app = Vue.createApp({
  data() {
    return {
      product: 'Socks',
      image: './assets/images/socks_blue.jpg',
      inStock: true // new data property //
    }
  }
})
```

Maintenant que nous avons ajouté les éléments, nous voulons afficher de manière conditionnelle une des 2 chaînes de caractères en fonction de la condition (`inStock`). Pour cela, c'est le bon moment pour apprendre une autre directive Vue.

4.3 La directive "v-if"

Nous pouvons ajouter la directive `v-if` sur un élément pour l'afficher en fonction d'une condition, de la manière suivante :

```
<p v-if="inStock">In Stock</p>
```

Maintenant, cet élément ne sera affiché que si `inStock` est vrai.

Nous pouvons combiner la directive `v-if` avec la directive `v-else` pour afficher un autre élément si la première condition s'avère fausse.

index.html

```
<p v-if="inStock">In Stock</p>
<p v-else>Out of Stock</p>
```

Maintenant, si `inStock` est false, nous verrons "Out of Stock" qui sera affiché dans la page.

4.4 La directive v-show

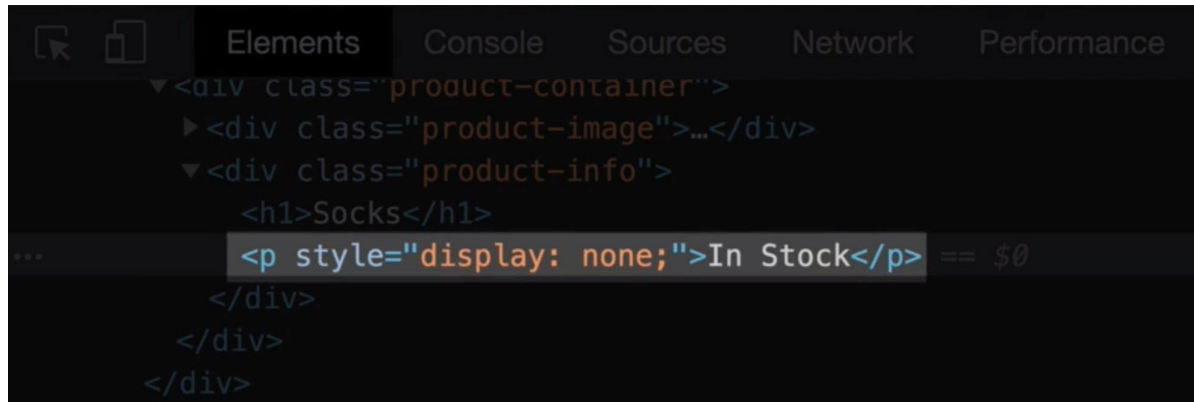
Il convient de noter que vous n'avez pas toujours besoin d'associer `v-if` avec `v-else`. Il y a nombreux cas d'utilisation où vous n'avez pas besoin de `v-else`. Dans ces cas, il est parfois préférable d'utiliser la directive `v-show`.

index.html

```
<p v-show="inStock">In Stock</p>
```

La directive `v-show` est utilisée pour basculer la **visibilité** d'un élément au lieu d'ajouter et de retirer l'élément du DOM entièrement, comme `v-if` le fait.

Comme vous pouvez l'imaginer, c'est une option plus performante si vous avez un élément qui « toggle ». Nous pouvons vérifier cela en fixant `inStock` à `false` et en visualisant l'élément dans l'inspecteur du navigateur. Quand `v-show` est utilisé, nous pouvons voir que l'élément est toujours présent dans le DOM, mais il est maintenant caché avec un « inline style » `display: none;`



4.5 Logique conditionnelle

Plus tôt, nous avons étudié `v-if` avec `v-else`. Jetons maintenant un coup d'œil à la façon dont nous pouvons ajouter des couches supplémentaires de logique conditionnelle.

Pour ce faire, nous allons remplacer `inStock` avec `inventory` :

- **main.js**

```
const app = Vue.createApp({
  data() {
    return {
      ...
      inventory: 100
    }
  }
})
...
```

Maintenant que notre condition (`inventory`) est un entier, nous pouvons utiliser une logique un peu plus complexe dans notre expression.

Par exemple :

index.html

```
<p v-if="inventory > 10">In Stock</p>
<p v-else>Out of Stock</p>
```

Maintenant, nous n'afficherons le premier tag p si inventory est supérieur à 10.

Imaginons maintenant que nous voulons afficher un nouveau message lorsque le produit est presque épuisé.

index.html

```
<p v-if="inventory > 10">In Stock</p>
<p v-else-if="inventory <= 10 && inventory > 0">Almost sold
out!</p>
<p v-else>Out of Stock</p>
```

La directive v-else-if nous donne une couche intermédiaire de logique. Donc dans cet exemple, si inventory vaut 8, nous afficherons « Almost sold out ! ».

Bien entendu, si inventory vaut zéro, nous allons par défaut au niveau final de v-else et afficher « Out of stock ».

4.6 A vous de coder !

Nous avons atteint la fin de la leçon et nous sommes prêt pour relever un nouveau défi :

Ajouter un Booléen onSale à l'objet de données.

Utilisation de onSale afin d'afficher de manière conditionnelle un tag p qui dit "On Sale", chaque fois que onSale est true.

Pour rappel, vous pouvez vérifier le code dans la [branche](#) L4-end.

5. Rendu de liste

Dans cette leçon, nous allons étudier le concept de rendu de liste.

Vous pouvez trouver le code de départ dans la [branche](#) L5-start.

5.1 Notre objectif

Afficher les listes HTML à partir d'un tableau présent dans nos données.

5.2 Parcourir un tableau de données

Dans le code de départ, nous avons maintenant un tableau details.

- **main.js**

```
const app = Vue.createApp({
  data() {
    return {
      ...
      details: ['50% cotton', '30% wool', '20%
polyester']
    }
  }
})
```

La question est maintenant de savoir comment afficher ces données sous forme de liste.

Nous commencerons par créer une liste non ordonnée dans notre **index.html**.

Dans le tag li, nous ajouterons une autre directive Vue : v-for

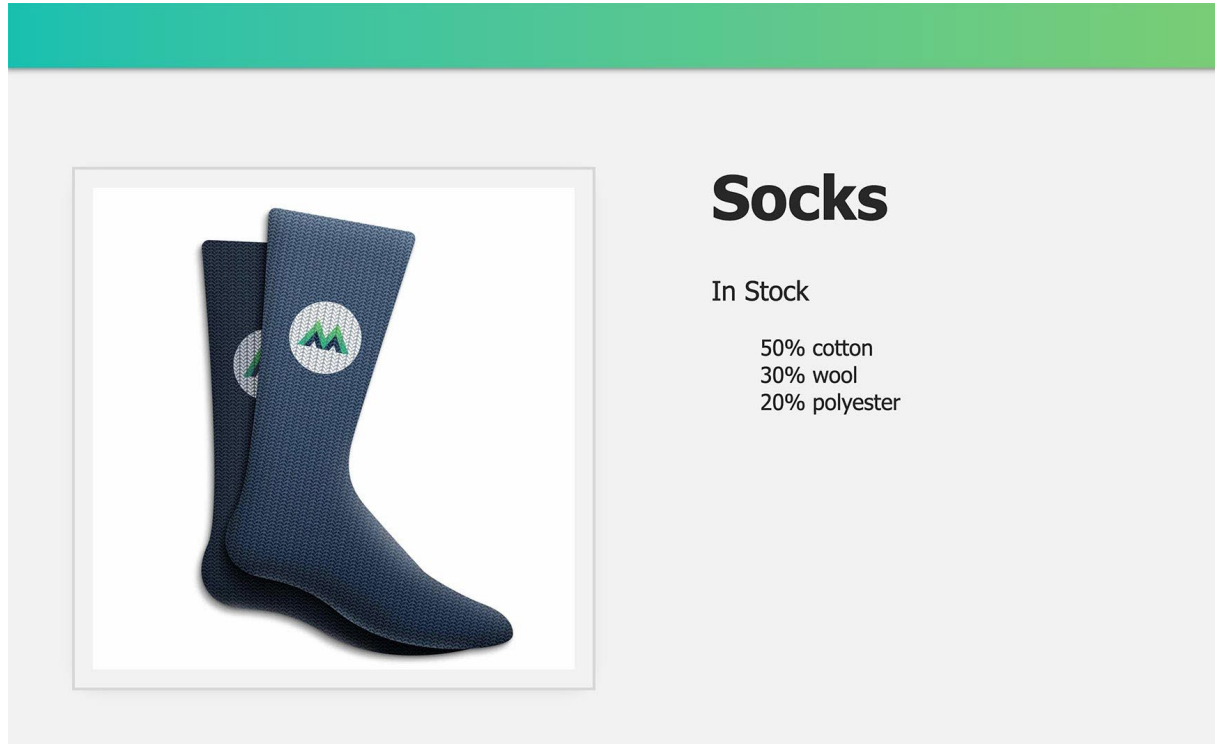
index.html

```
<ul>
  <li v-for="detail in details">{{ detail }}</li>
</ul>
```

À l'intérieur de v-for nous avons écrit : detail in details. Ici, details se réfère au tableau details de nos données, et detail est l'alias de l'élément courant de ce tableau, permettant d'afficher le contenu du li.

Dans le HTML nous utiliserons l'expression : {{ detail }} pour afficher chaque détail.

Si nous vérifions dans le navigateur, nous verrons qu'une liste de details est affichée.



Jusqu'à présent, c'est bien, mais comment v-for fonctionne réellement ?

5.3 Variantes de couleur pour chaque produit

Pour se familiariser avec le rendu de la liste avec v-for, nous allons travailler sur un autre exemple dans notre application. Ajoutons un nouveau tableau variants à nos données :

- main.js

```
data() {  
  return {  
    ...  
    variants: [  
      { id: 2234, color: 'green' },  
      { id: 2235, color: 'blue' }  
    ]  
  }  
}
```

Nous avons maintenant un tableau qui contient un objet pour chaque variante de notre produit. Chaque variante de produit comporte un id, et une color. Donc pour notre prochaine tâche, nous allons afficher chaque variante de couleur, et utiliser l'id pour aider Vue à suivre les éléments de notre liste.

index.html

```
<div v-for="variant in variants" :key="variant.id">{{  
  variant.color }}</div>
```

Remarquez comment nous utilisons la notation . pour accéder à chaque attribut de variant alors que nous sommes en train de parcourir le tableau variants.

Mais à quoi sert l'attribut :key ?

5.4 Attribut Key : un élément essentiel pour les éléments de la liste

En disant :key="variant.id", nous utilisons le raccourci de v-bind afin de lier l'id de la variante à l'attribut key. Cela donne à chaque élément du DOM une clé unique afin que Vue puisse s'accrocher à l'élément et ne pas en perdre la trace à mesure que les choses se mettent à jour dans l'application.

Cela permet d'améliorer les performances, et plus tard, si vous faites quelque chose comme « animer vos éléments », vous verrez que l'attribut key aide

vraiment Vue à gérer efficacement vos éléments au fur et à mesure qu'ils se déplacent dans le DOM.

5.5 A vous de coder !

Nous avons atteint la fin de la leçon et nous sommes prêt pour relever un nouveau défi :

Ajouter un tableau `sizes` à l'objet de données. Les tailles sont S, M, L ou XL

Utilisation de `v-for` pour afficher les `sizes` dans une liste HTML.

Pour rappel, vous pouvez vérifier votre code grâce à la [branche](#) L5-end.

6. Gestion des événements

Dans cette leçon, nous allons étudier le concept de gestion des événements. Vous pouvez trouver le code de départ dans la [branche](#) L6-start.

Dans le code de départ, vous verrez que nous avons maintenant un bouton « Add to Cart », avec un « panier », qui inclut une expression pour afficher la valeur de notre nouvelle donnée cart.

index.html

```
<div class="cart">Cart({{ cart }})</div>
...
<button class="button">Add to Cart</button>
```

- **main.js**

```
data() {
  return {
    cart: 0,
    ...
  }
}
```

6.1 Notre objectif

Nous voulons pouvoir cliquer sur le bouton et incrémenter la valeur de cart.

6.2 Écouter les événements : La directive v-on

Afin de savoir quand le bouton est cliqué, nous devons écouter les événements sur cet élément, en particulier les événements « clic ». Nous pouvons y parvenir en utilisant une autre directive Vue : v-on.

index.html

```
<button class="button" v-on:click="logic to run">Add to  
Cart</button>
```

Ici, nous disons à v-on quel type d'événement écouter : l'événement click. À l'intérieur des guillemets, nous plaçons la logique (ou le nom de la méthode) que nous voulons exécuter quand cet événement se produit.

Si nous écrivons `v-on:click="cart += 1"`, nous allons augmenter la valeur du « panier » de 1, lorsqu'un événement de clic se produit.

6.3 Déclenchement d'une méthode

Parce que la logique `cart += 1` est très simple, nous pourrions la garder in-line sur l'élément button. Mais souvent, nous devons déclencher une logique plus complexe. Dans ces situations, nous pouvons ajouter une méthode qui est appelée lorsque l'événement se produit. C'est ce que nous allons voir maintenant.

index.html

```
<button class="button" v-on:click="addToCart">Add to  
Cart</button>
```

Maintenant, quand le bouton est cliqué, la méthode `addToCart` sera exécutée. Ajoutons cette méthode à l'objet d'options de notre application Vue :

```
const app = Vue.createApp({  
  data() {  
    return {  
      cart: 0,  
      ...  
    }  
  },  
  methods: {  
    addToCart() {  
      this.cart += 1  
    }  
  }  
})
```

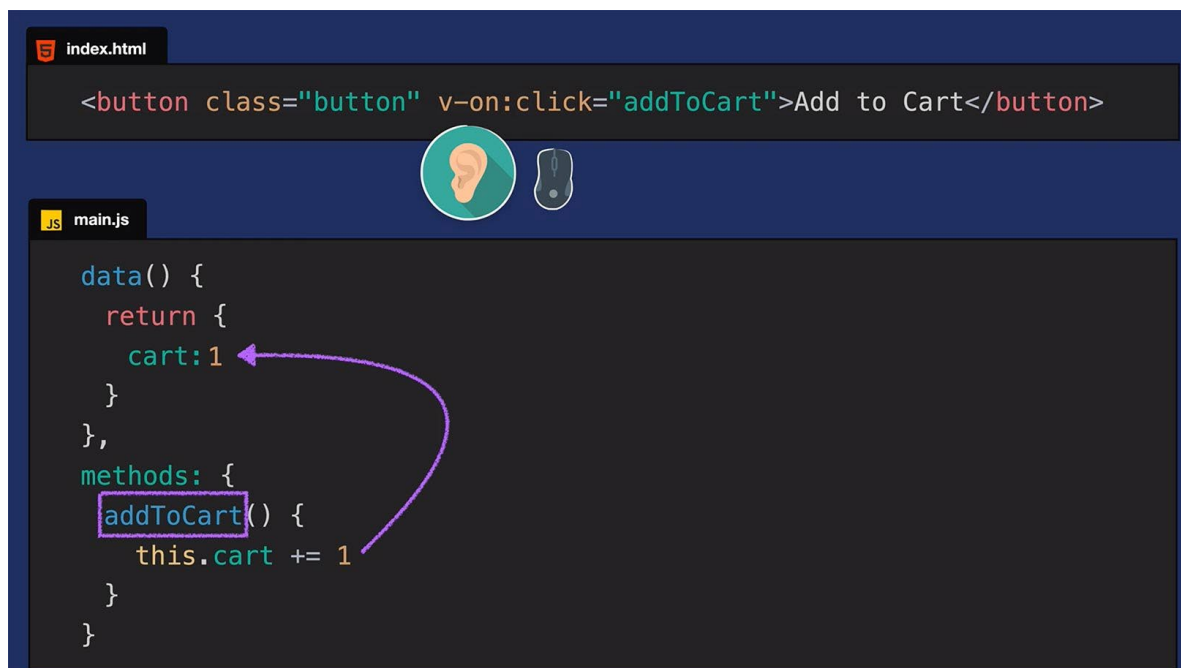
```
})
```

Remarquez comment nous avons ajouté l'option `methods`, et à l'intérieur de cela, nous avons ajouté la nouvelle méthode `addToCart`, qui contient la même logique que précédemment. La différence ici est `this.cart` fait référence à la donnée `cart` dans les données de cette instance `Vue`.

Dans le navigateur, nous devrions maintenant pouvoir cliquer sur le bouton de **l'ajout d'un panier** et voir la valeur du panier augmenter de 1.

6.4 Comprendre la directive `v-on`

Examinons de plus près le fonctionnement de cet événement.



En ajoutant `v-on` à un élément, nous lui donnons essentiellement une oreille qui peut écouter les événements. Dans ce cas, nous avons précisé que nous écoutons les événements de clics. Lorsqu'un clic se produit, la méthode `addToCart` s'exécute, c'est à dire, prend la valeur du « panier » et l'incrémente de 1.

6.5 Un raccourci pour la directive v-on

Comme vous pouvez l'imaginer, écouter des événements sur vos éléments est très commun. v-on a également un raccourci : @

Notre code pourrait donc être simplifié :

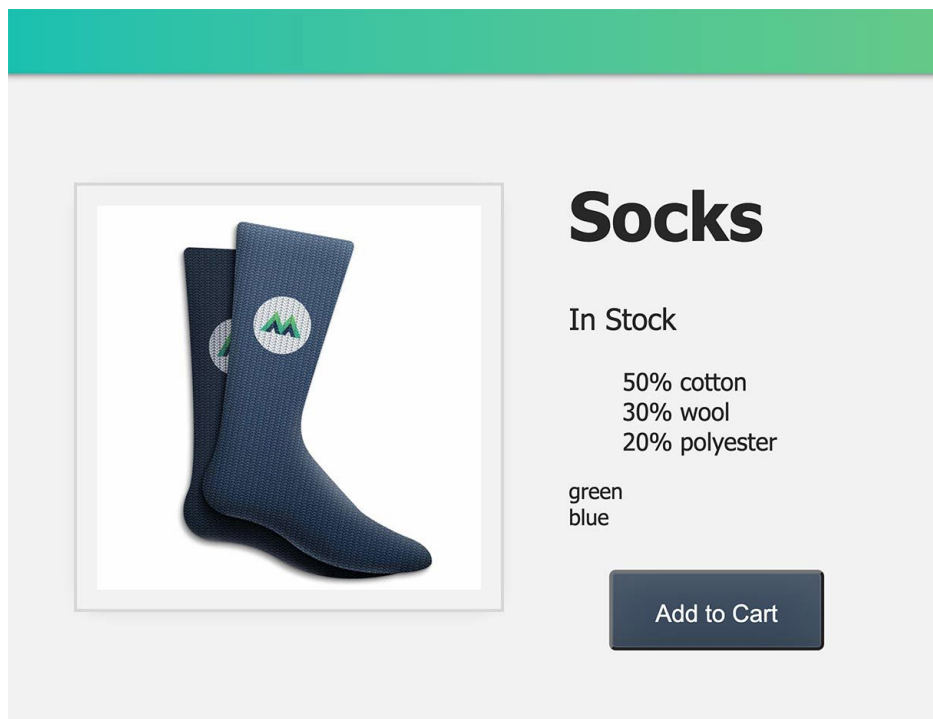
index.html

```
<button class="button" @click="addToCart">Add to  
Cart</button>
```

6.6 Un autre exemple : Événements de la Souris

Maintenant que nous comprenons les bases de la gestion des événements, écoutons un autre type d'événement dans notre application Vue.

Actuellement, nous affichons les couleurs des variantes, « vert » et « bleu », juste en dessous des détails du produit :



Ne serait-il pas bien si, quand nous passons notre souris sur « vert » et « bleu », nous déclenchons une mise à jour de l'image des chaussettes ? Ajoutons la possibilité d'écouter l'événement `mouseover` sur ces noms de couleurs.

Comme nous voulons mettre à jour l'image lorsque nous passons la souris sur les couleurs de chaque variante de chaussette, j'ai ajouté une nouvelle propriété à chaque objet de `variants`.

- `main.js`

```
data() {  
  return {  
    ...  
    variants: [  
      { id: 2234, color: 'green', image:  
'./assets/images/socks_green.jpg' },  
      { id: 2235, color: 'blue', image:  
'./assets/images/socks_blue.jpg' },  
    ]  
  }  
}
```

Maintenant, chaque variante a un chemin d'image pour les chaussettes vertes et bleues, respectivement. Nous sommes prêts à ajouter un « listener » pour l'événement `mouseover` sur la couleur de la variante `div`.

- `main.js`

```
<div  
  v-for="variant in variants"  
  :key="variant.id"  
  @mouseover="updateImage(variant.image)">  
  {{ variant.color }}  
</div>
```

Lorsque l'événement `mouseover` se produit, nous déclenchons la méthode `updateImage`, en passant en paramètre le chemin de l'image de chaque variante. Cette méthode ressemble à ceci :

```
methods: {  
  ...  
  updateImage(variantImage) {  
    this.image = variantImage  
  }  
}
```

Quand cette méthode est exécutée, elle affecte à `this.image` (dans les données de cette instance de Vue) l'image de la variante qui a été passée en paramètre.

Maintenant, dans le navigateur, quand nous survolons notre souris sur « vert », nous devrions voir l'image verte. Quand nous survolons « bleu », nous devrions voir l'image bleue. A vous de tester !

6.7 A vous de coder !

Nous avons atteint la fin de la leçon et voici le nouveau défi :

Créez un nouveau bouton ayant pour nom « Remove item » qui décrémente la valeur du panier.

Attention, nous ne voulons pas de valeur négative pour la valeur du panier !

Pour rappel, vous pouvez vérifier votre code grâce à la [branche](#) L6-end.

7. Liaison de Classe et liaison de Style

Dans cette leçon, nous allons étudier le concept de liaison de classe et de liaison de style. Vous pouvez trouver le code de départ dans la L7-start [branche](#).

7.1 Notre objectif

Lier les classes et les styles à nos éléments en fonction des données de notre application.

7.2 Liaison du style

Dans la dernière leçon, nous avons ajouté la fonctionnalité qui permet, au survol de la souris sur « vert » ou « bleu », de mettre à jour l'image affichée à savoir les chaussettes vertes ou bleues.

Mais l'expérience de l'utilisateur ne serait-elle pas plus agréable si, au lieu de survoler le mot « vert » ou « bleu », nous survolions les *couleurs* réelles vert et bleu ?

Nous allons créer des cercles verts et bleus que nous pourrions survoler. Nous pouvons y parvenir en utilisant la liaison de style.

Tout d'abord, pour styler nos divs comme des cercles, nous allons avoir besoin d'ajouter une nouvelle classe `.color-circle` à la variante div.

index.html

```
<div
  v-for="variant in variants"
  :key="variant.id"
  @mouseover="updateImage(variant.image)"
  class="color-circle">
</div>
```

Ceci est déjà dans notre fichier css. Comme vous pouvez le voir, il transforme simplement nos divs en un cercle de 50px de diamètre :

Styles.css

```
.color-circle {  
  width: 50px;  
  height: 50px;  
  margin-top: 8px;  
  border: 2px solid #d8d8d8;  
  border-radius: 50%;  
}
```

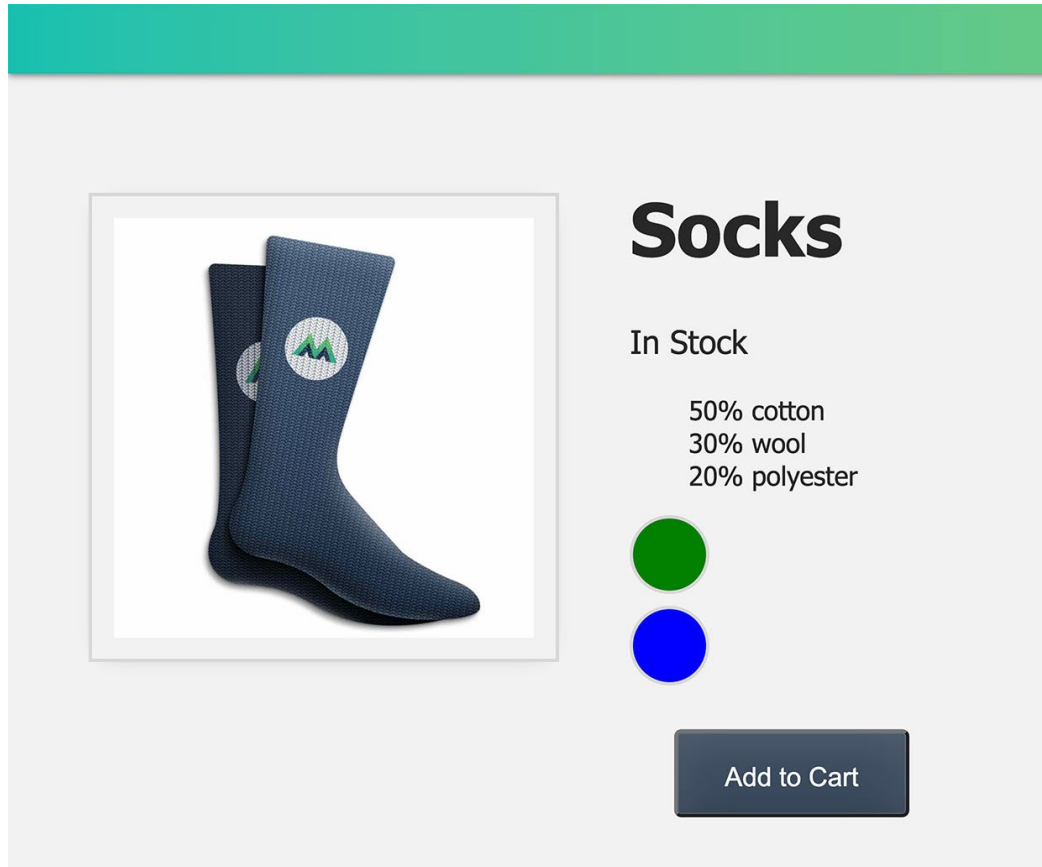
Maintenant que nous avons compris, nous pouvons passer à la liaison de style réelle. Nous voulons lier les styles aux variantes divs. Nous le faisons en utilisant `v-bind` (ou son raccourci :) sur l'attribut `style` ce qui va le lier à un objet.

index.html

```
<div  
  v-for="variant in variants"  
  :key="variant.id"  
  @mouseover="updateImage(variant.image)"  
  class="color-circle"  
  :style="{ backgroundColor: variant.color }">  
</div>
```

Ici, nous sommes en train de mettre sur les divs un `backgroundColor` ayant la valeur `variant.color`. Donc, au lieu d'afficher les mots, « vert » et « bleu », nous les utilisons pour définir la couleur de fond de nos cercles.

En vérifiant cela dans le navigateur, nous devrions maintenant voir deux cercles de couleur remplis d'un fond vert et bleu.



Cool ! Maintenant, essayons de comprendre comment tout cela fonctionne.

7.3 Comprendre la liaison de style

Sur notre variante div, nous avons ajouté l'attribut style et lié un objet à celui-ci.

index.html

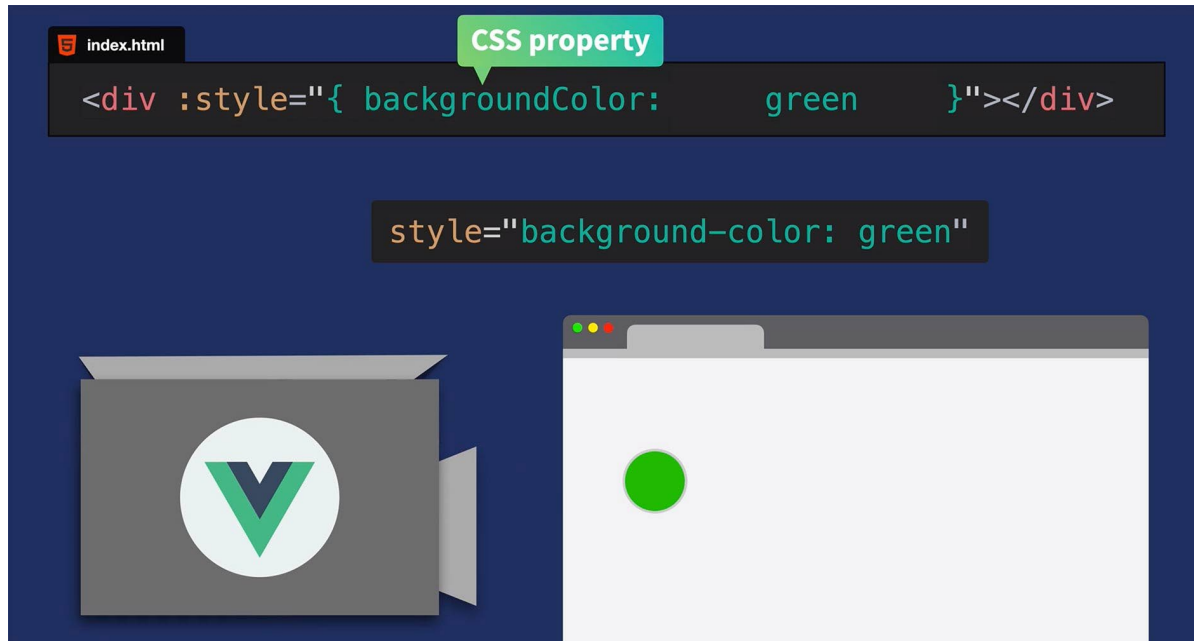
```
<div
  ...
  :style="{ backgroundColor: variant.color }">
</div>
```

Cet objet de style a la propriété CSS `backgroundColor`. Nous lui affectons la couleur de la variante courante de l'itération `v-for`.

Dans la première itération, `variant.color` est "green"

Vue prend cette information et la convertit en code : `style="{ backgroundColor: green }"`

Puis affiche un cercle de fond vert.



Il répète ce processus pour la deuxième variante de couleur pour créer le cercle bleu.

7.4 Camel vs Kebab

Il y a des choses importantes à prendre en compte lors de l'utilisation de la liaison de style.

```
<div :style="{ backgroundColor: variant.color }"></div>
```

À l'intérieur de cette expression, souvenez-vous que cet objet de style est en JavaScript. C'est pourquoi j'ai utilisé le camelCase dans le nom de la propriété. Si j'avais écrit `background-color`, il n'aurait pas pu interpréter le signe `-`.

Puisque nous sommes dans cet objet JavaScript, nous devons utiliser le camelCase.

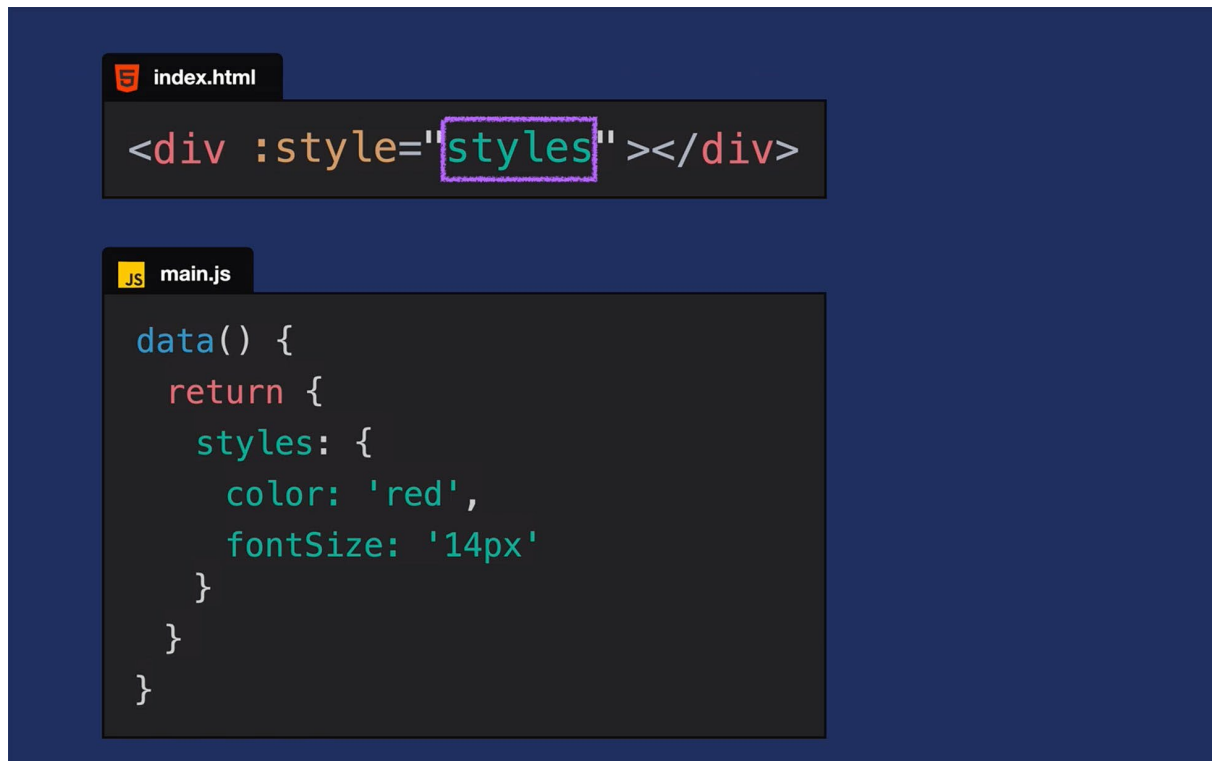
Une autre solution est d'utiliser le « kebab-case » :

```
<div :style="{ 'background-color': variant.color }"></div>
```

Attention avec le « kebab-case » de ne pas oublier les guillemets.

7.5 Liaison de style : Objets

Parfois, vous pouvez ajouter un tas de styles à un élément, mais les ajouter tous en « in-line » pourrait rendre tout cela chaotique. Dans ces situations, nous pouvons nous lier à un objet de style qui est défini dans nos données.



Maintenant que nous avons examiné le sujet de la liaison de style, regardons un sujet similaire : la liaison des classes.

7.6 Liaison de classe

Dans notre application, vous remarquerez que lorsque la valeur de `inStock` est à **false**, nous pouvons toujours cliquer sur le bouton « Ajouter au panier » et incrémenter la valeur du panier. Mais si le produit est en rupture de stock, peut-être que nous ne voulons pas que l'utilisateur soit en mesure d'ajouter le produit au panier. Alors changeons ce comportement en désactivant le bouton chaque fois que `inStock` est false ET en faisant *apparaître* le bouton désactivé, à l'aide de la liaison de classe.

Pour commencer, nous utiliserons le raccourci pour `v-bind` sur l'attribut `disabled` pour ajouter cet attribut chaque fois que notre produit n'est pas en stock.

index.html

```
<button
  class="button"
  :disabled="!inStock"
  @click="addToCart">
  Add to Cart
</button>
```

Maintenant, chaque fois que `inStock` est false Et que nous cliquons sur le bouton « Ajouter au panier », rien ne se passera car il est désactivé. Mais le bouton *semble* toujours actif, ce qui est trompeur pour nos utilisateurs. Utilisons donc la liaison de la classe pour ajouter un `disabledButton` à chaque fois que `inStock` est false.

Vous verrez dans notre fichier CSS que nous avons déjà la classe `disabledButton`, qui fixe le `background-color` à gris et qui rend le `cursor` « Non autorisé ».

Styles.css

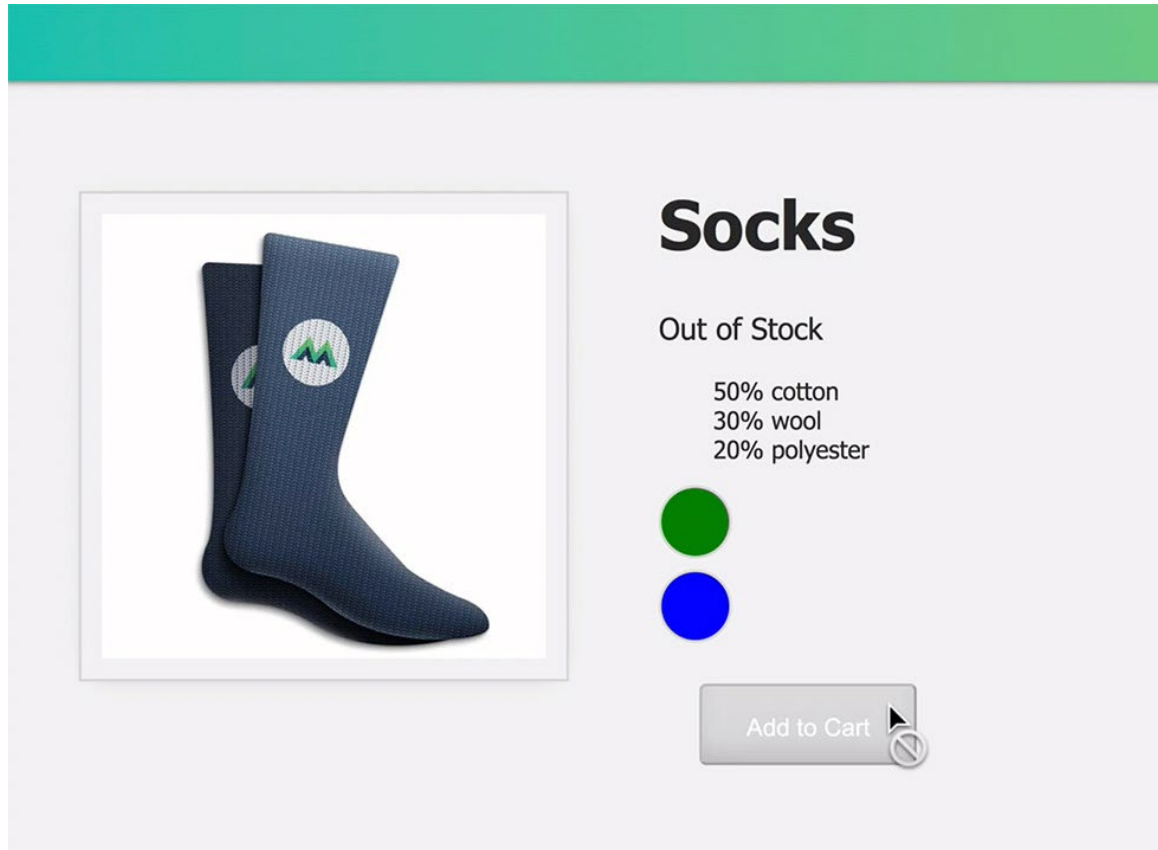
```
.disabledButton {
  background-color: #d8d8d8;
  cursor: not-allowed;
}
```

Nous devons appliquer cette classe de manière conditionnelle, en fonction de la valeur de `inStock`. Pour cela, nous utiliserons le raccourci pour `v-bind` sur la `class` attribut, et nous utiliserons une expression qui ajoute ou qui enlève la classe `disabledButton` grâce à `!inStock`.

index.html

```
<button
  class="button"
  :class="{ disabledButton: !inStock }"
  :disabled="!inStock"
  @click="addToCart">
  Add to Cart
</button>
```

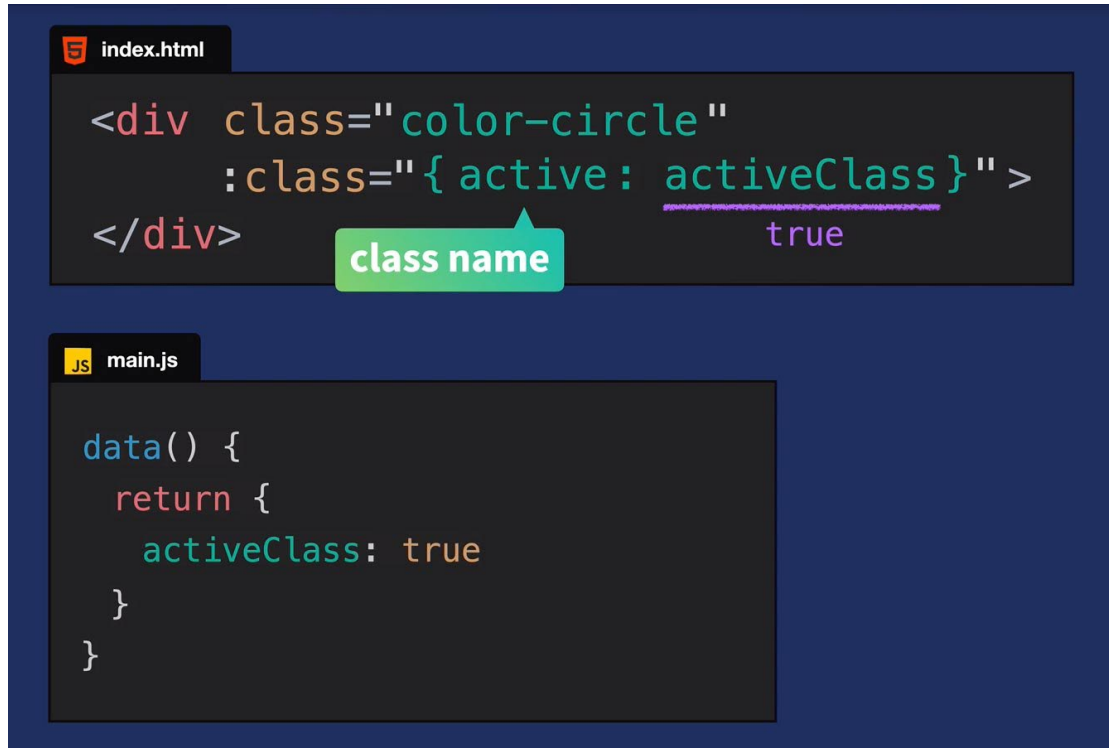
Maintenant, à chaque fois que `inStock` est `false`, non seulement le bouton sera désactivé, mais il apparaîtra également désactivé.



7.7 Noms de classe multiples

Quand on commence avec la liaison des classes, il y a des choses à noter. Par exemple, que se passe-t-il quand nous avons déjà une classe existante et que nous voulons ajouter une autre classe en fonction d'une condition basée sur une valeur des données ?

Par exemple, si nous avons déjà la classe `color-circle` sur la div, et nous ajoutons conditionnellement la classe `active`, à quoi cela ressemblera-t-il ?

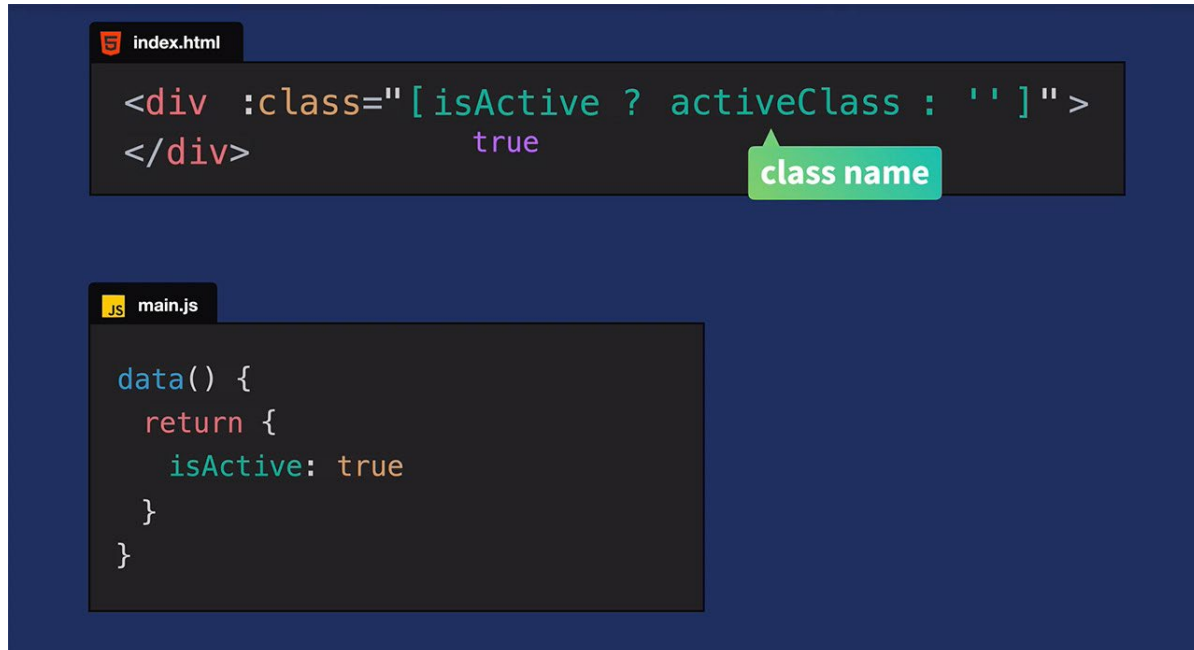


Ces classes vont être combinées comme cela :

```
<div class="color-circle active"></div>
```

7.8 Opérateurs ternaires

Un outil utile avec la liaison de classe est la possibilité d'utiliser des opérateurs ternaires « in-line » pour ajouter différentes classes basées sur une condition.



Dans ce cas, parce que `isActive` est `true`, nous ajoutons la classe `activeClass`. Dans le cas contraire nous aurions pu ajouter une classe complètement différente.

Les variations dans la syntaxe et dans les cas d'utilisation que nous venons de voir avec la liaison de classe et de style ne sont que le début. Vous pouvez consulter la documentation officielle de Vue pour davantage de cas d'utilisation et d'exemples.

7.9 A vous de coder !

Nous avons atteint la fin de la leçon et voici le nouveau défi :

Réaliser une liaison entre la classe `out-of-stock-img` et l'image des chaussettes (bleues ou vertes) chaque fois que `inStock` est `false`.

Pour rappel, vous pouvez vérifier votre code grâce à la [branche](#) L7-end.

8. Propriétés calculées (Computed Properties)

Dans cette leçon, nous allons étudier le concept de propriétés calculées. Vous pouvez trouver le code de départ dans la [branche](#) L8-start.

8.1 Notre objectif

Mettre à jour l'image de la variante ET gérer si elle est en stock ou non, en utilisant des propriétés calculées.

8.2 Une propriété calculée simple

Dans le code de départ, vous remarquerez que nous avons une nouvelle propriété de données :

- **main.js**

```
data() {  
  return {  
    product: 'Socks',  
    brand: 'Vue Mastery'  
  }  
}
```

Et si nous voulions combiner la donnée brand et la donnée product dans notre modèle ? Nous pourrions le faire dans une expression js comme suit :

index.html

```
<h1>{{ brand + ' ' + product }}</h1>
```

Si nous vérifions cela dans le navigateur, nous verrions « Vue Mastery Socks » affiché. Mais ne serait-il pas plus propre de gérer cette logique ailleurs que dans le template HTML ? Notre application a la capacité de calculer cette valeur pour nous. Par exemple, en prenant brand et product, on peut les additionner, et restituer cette nouvelle valeur.

Les propriétés calculées sont des propriétés que nous pouvons ajouter à une application Vue et c'est l'application qui calcule ces valeurs pour nous. Cela

nous aide à garder la logique de calcul hors du HTML et nous permet d'améliorer les performances. Pour l'instant, transformons cet exemple simple en propriété calculée. Nous modifierons le `h1` comme cela :

index.html

```
<h1>{{ title }}</h1>
```

Maintenant, `title` est le nom d'une propriété calculée que nous allons créer. Tout d'abord, nous ajouterons l'option `computed` à l'application, juste en dessous de `methods`, puis nous pouvons créer la propriété `title`.

- main.js

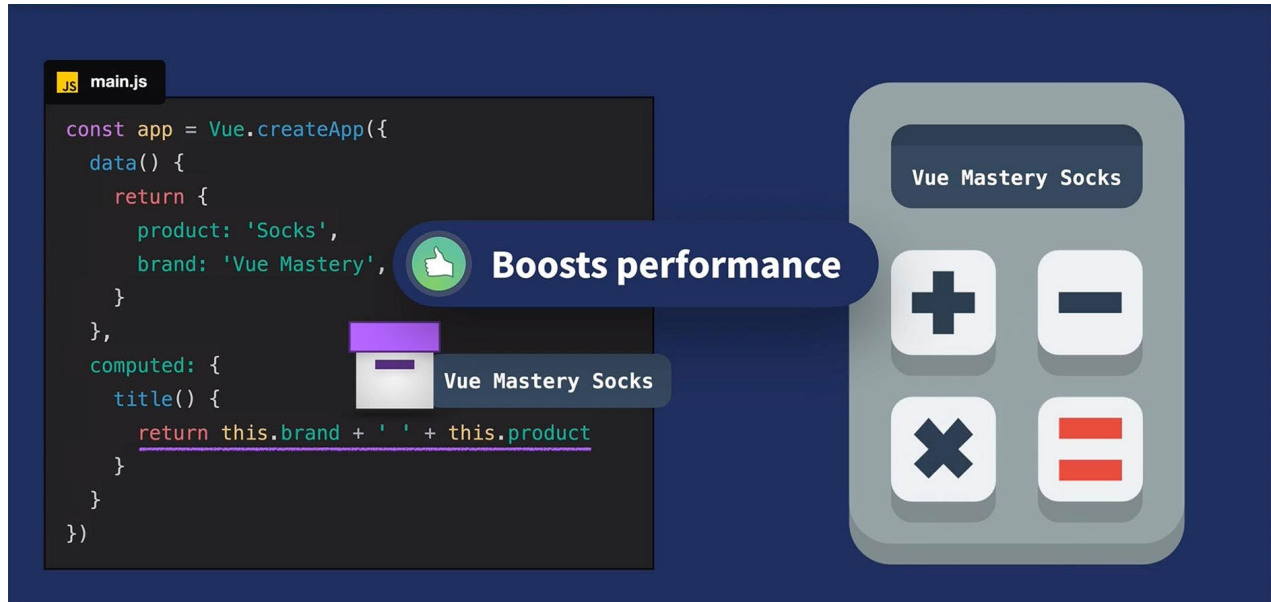
```
...
computed: {
  title() {
    return this.brand + ' ' + this.product
  }
}
```

Si nous vérifions le navigateur, nous verrons toujours « Vue Mastery Socks » s'affiché, sauf que nous avons maintenant extrait cette logique de calcul hors du HTML et l'avons bien positionné sur l'objet des options.

Mais comment fonctionnent exactement les propriétés calculées ? Regardons cela de plus près.

8.3 Comme une calculatrice ...

J'aime voir les propriétés calculées comme un calculatrice, parce qu'elles calculent ou *compute* des valeurs pour nous. Cette calculatrice prend nos valeurs, `brand` et `product`, les additionne et nous donne le résultat.



Comme je l'ai mentionné plus tôt, les propriétés calculées nous apportent une amélioration des performances. En effet, ils cachent la valeur calculée. La valeur ('Vue Mastery Socks') se stocke et ne se met à jour que lorsque cela est nécessaire, lorsque l'une de ses dépendances change. Par exemple, si le brand devait changer de 'Vue Mastery' à 'Node Mastery', notre propriété calculée recevrait cette nouvelle valeur de brand puis recalcule et restitue la nouvelle valeur : 'Node Mastery Socks'

Maintenant que nous commençons à comprendre les propriétés calculées, mettons en œuvre un exemple plus pratique dans notre application Vue.

8.4 Calcul de l'image et de la quantité

Retour à notre code, ajoutons une nouvelle propriété quantity à nos objets de variante.

- main.js

```
data() {
  return {
    ...
    variants: [
```

```
{ id: 2234, color: 'green', image:
'./assets/images/socks_green.jpg', quantity: 50 },
{ id: 2235, color: 'blue', image:
'./assets/images/socks_blue.jpg', quantity: 0 },
]
}
```

Remarquez que les chaussettes vertes ont une quantity de 50 alors que les chaussettes bleues ont 0. En d'autres termes, les chaussettes vertes sont en stock et les chaussettes bleues sont en rupture de stock. Cependant, nous affichons actuellement « En stock » ou « Out of stock » en fonction de `inStock` qui ne reflète plus la vérité sur notre produit et ses quantités. Nous allons donc vouloir créer une propriété calculée que nous pouvons utiliser pour afficher « En stock » ou « Out of stock » en se basant sur ces nouvelles quantités.

Actuellement l'événement `mouseover` de la souris déclenche la méthode `updateImage()`. Nous allons lui faire déclencher une nouvelle méthode appelée `updateVariant()`.

index.html

```
<div
  v-for="(variant, index) in variants"
  :key="variant.id"
  @mouseover="updateVariant(index)" <!-- new method -->
  class="color-circle"
  :style="{ backgroundColor: variant.color }">
</div>
```

Remarquez comment nous passons l'index de la variante en paramètre : `updateVariant(index)`. Nous avons eu accès à `index` en l'ajoutant un deuxième paramètre dans notre directive `v-for`:

```
v-for="(variant, index) in variants"
```

Pourquoi sommes-nous en train de passer en paramètre l'index ? Nous allons l'utiliser pour indiquer à notre application quelle variante est actuellement

survolée par la souris, afin qu'elle puisse utiliser ces informations pour déclencher la mise à jour à la fois de l'image ET si cette variante est en stock ou non.

Nous ajouterons une nouvelle propriété de données à notre application, qui sera mise à jour grâce à index.

- **main.js**

```
data() {  
  return {  
    ...  
    selectedVariant: 0,  
    ...  
  }  
}
```

Notre méthode `updateVariant()` définit la valeur `selectedVariant` en n'utilisant l'index de la variante survolée par la souris.

- **main.js**

```
updateVariant(index) {  
  this.selectedVariant = index  
}
```

Nous avons mis en œuvre un moyen permettant à notre application de savoir quelle variante de produit est sélectionnée, et nous pouvons utiliser ces informations pour déclencher le calcul de l'image à afficher et s'il doit afficher "En stock" ou "Out of stock", sur la variante survolée par la souris.

Nous sommes maintenant prêts à supprimer `image` et `inStock` de nos données, et remplacer celles-ci par des propriétés calculées.

- **main.js**

```
computed: {  
  image() {  
    return ??
```

```
},  
inStock() {  
  return ??  
}  
}
```

Alors, comment récupérer l'image et la quantité de la variante ? Cela ressemblera à ceci :

- **main.js**

```
image() {  
  return this.variants[this.selectedVariant].image  
}
```

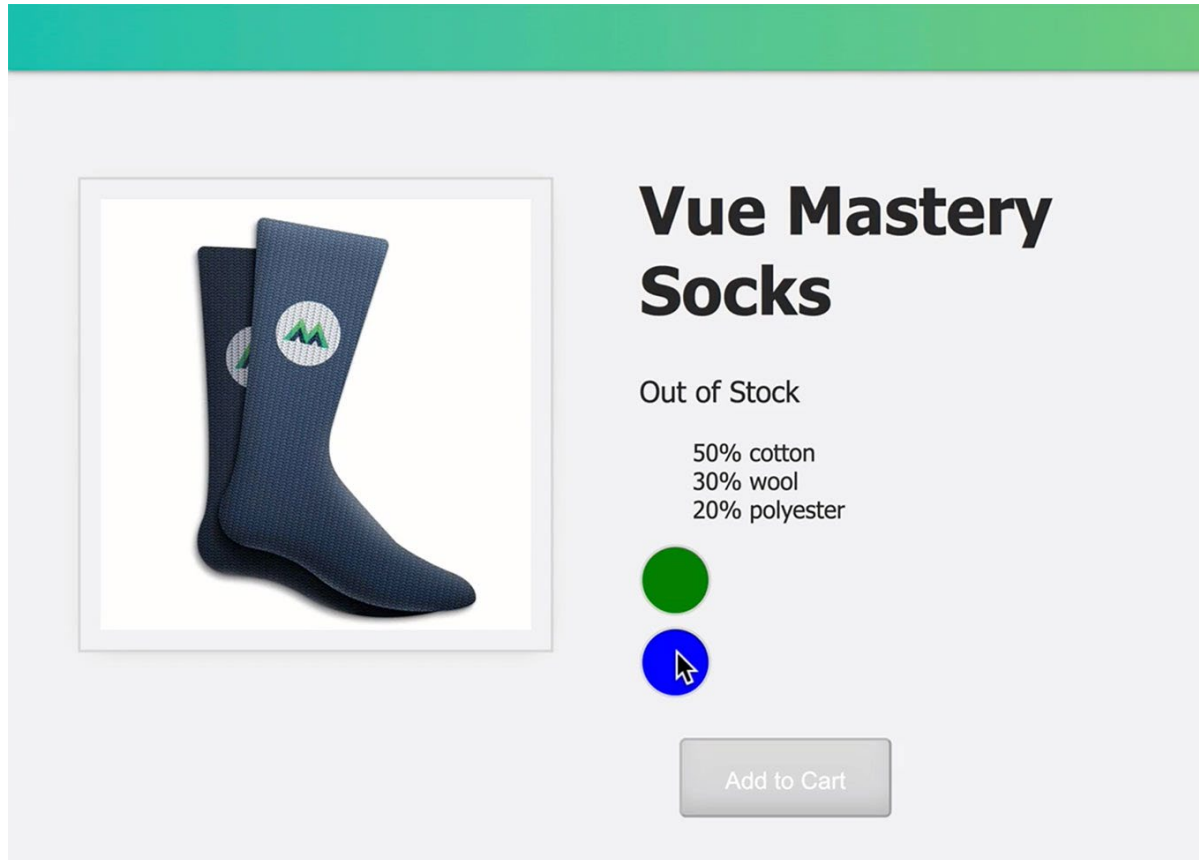
Nous ciblons le premier ou le deuxième élément de notre tableau variants grâce à selectedVariant qui vaut soit 0 ou 1, en fonction du cercle de couleur qui est survolé par la souris.

La logique de propriété calculée inStock est presque identique :

- **main.js**

```
inStock() {  
  return this.variants[this.selectedVariant].quantity  
}
```

Vérifiez cela dans le navigateur, quand nous survolons les cercles de couleur, non seulement nous mettons à jour l'image de la variante, mais nous affichons également si cette variante est en stock ou en rupture de stock, en utilisant sa quantité.



Remarquez comment le bouton est toujours automatiquement mis à jour pour nous, en s'activant ou en se désactivant. C'est parce que, dans notre modèle, nous utilisons toujours `inStock`.

index.html

```
<button
  class="button"
  :class="{ disabledButton: !inStock }"
  :disabled="!inStock"
  v-on:click="addToCart">
  Add to Cart
</button>
```

Maintenant `inStock` n'est plus une propriété des données : c'est la nouvelle propriété calculée.

8.5 A vous de coder !

Nous avons atteint la fin de la leçon et voici le nouveau défi :

Ajouter un booléen `onSale` (en français « en vente ») aux données.

Utiliser une propriété calculée pour afficher la chaîne : `brand + ' ' + product + ' ' + is on sale` dans une méthode `sale()`, chaque fois que `onSale` est true.

Afficher cela juste en dessous du `h1` dans une balise `p`

Pour rappel, vous pouvez vérifier votre code grâce à la [branche](#) L8-end.

9. Composants et Props

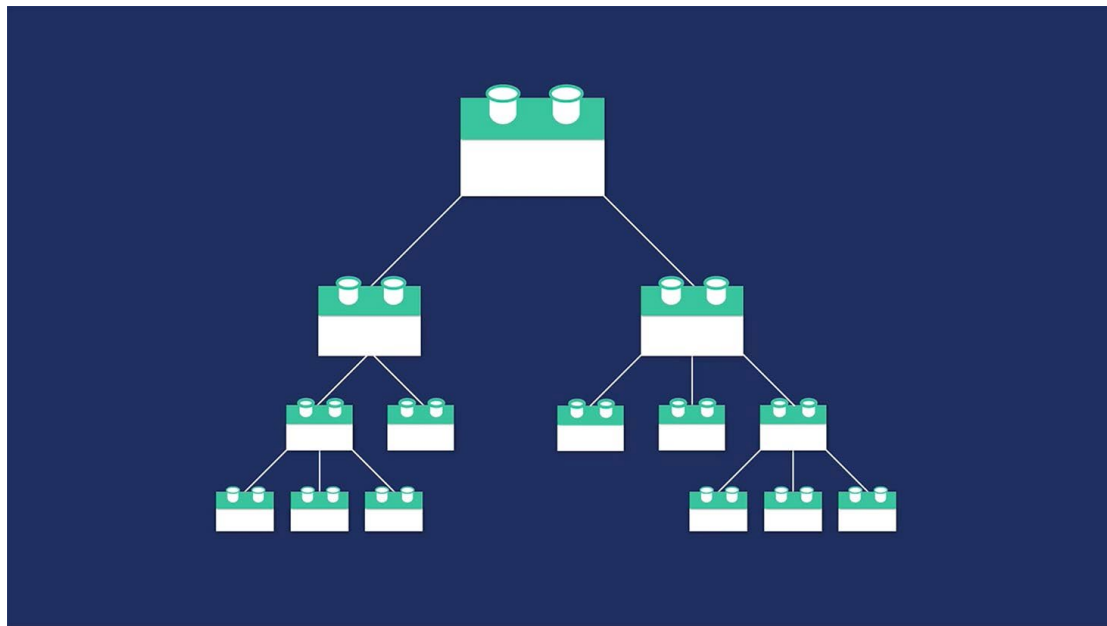
Dans cette leçon, nous allons étudier le concept de composant et de Props. Vous pouvez trouver le code de départ dans la [branche L9-start](#).

9.1 Notre objectif

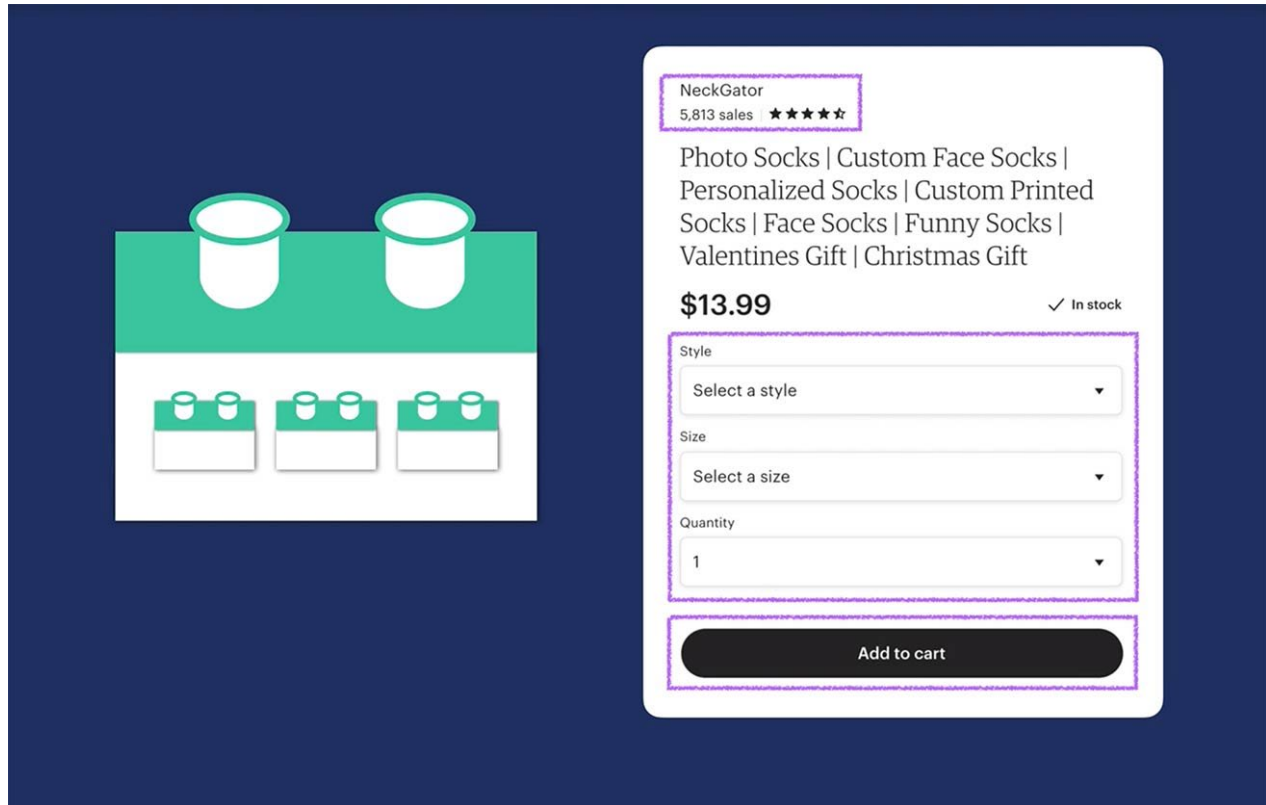
Refactoriser l'application d'exemple pour utiliser un composant produit, qui utilise un Props.

9.2 Blocs de construction d'une application Vue

Dans les Frameworks JavaScript modernes, les composants sont les éléments constitutifs d'une application, et c'est également le cas de Vue. Vous pouvez imaginer des composants comme des Legos que vous pouvez assembler les uns aux autres dans une hiérarchie d'arbres.

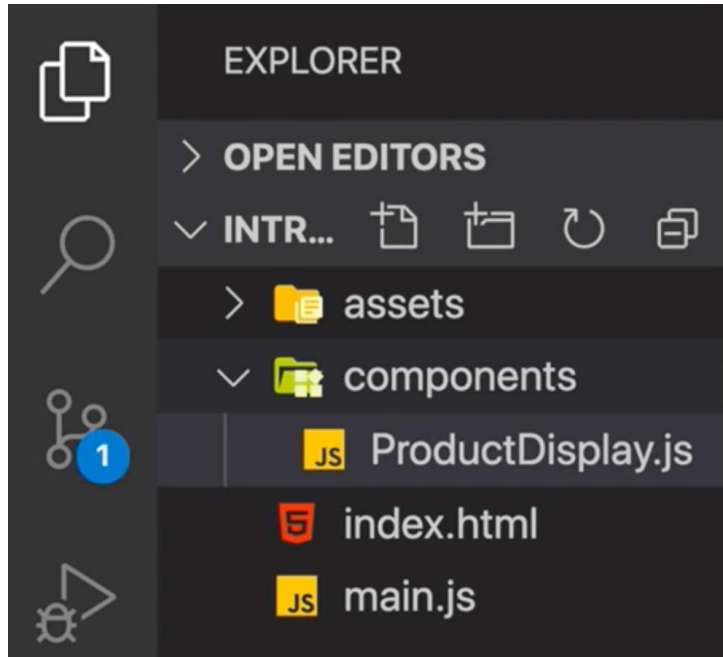


N'importe quelle page web donnée peut être composée de plusieurs composants, et il est courant que les composants soient des composants « parents » qui aient des composants « enfants » emboîtés en eux.



9.3 Création de notre premier composant

Reprenons notre application et créons notre premier composant. Comme notre application aura finalement plusieurs composants, nous allons créer un dossier **components**, à l'intérieur duquel nous allons créer notre premier composant, appelé **ProductDisplay.js**.



La syntaxe pour créer un composant ressemble à ceci :

Composants/ProductDisplay.js

```
app.component('product-display', {})
```

Le premier argument est le nom du composant, dans ce cas 'product-display', et le deuxième argument est un objet pour configurer notre composant (similaire à l'objet d'options utilisé pour configurer notre application racine Vue).

Template

Comme notre composant a besoin d'avoir une structure, nous allons ajouter la propriété `template` et coller tout le code HTML basé sur le produit dans un [template littéral](#).

Composants/ProductDisplay.js

```
app.component('product-display', {  
  template:  
    /*html*/
```

```
`<div class="product-display">
  <div class="product-container">
    <div class="product-image">
      
    </div>
    <div class="product-info">
      <h1>{{ title }}</h1>

      <p v-if="inStock">In Stock</p>
      <p v-else>Out of Stock</p>

      <div
        v-for="(variant, index) in variants"
        :key="variant.id"
        @mouseover="updateVariant(index)"
        class="color-circle"
        :style="{ backgroundColor: variant.color }">
      </div>

      <button
        class="button"
        :class="{ disabledButton: !inStock }"
        :disabled="!inStock"
        v-on:click="addToCart">
        Add to Cart
      </button>
    </div>
  </div>
</div>`
})
```

Notez que nous n'avons pas changé le code, nous le déplaçons simplement dans le Composant `product-display` pour qu'il y soit encapsulé. Si vous vous demandez ce que `/*html*/` vient faire là, ceci active l'extension VS Code [es6-string-html](#), qui nous donne une mise en évidence de la syntaxe HTML même si nous sommes dans un « template littéral ». Pour vous en assurer, vous pouvez

supprimer `/*html*/` et vous verrez que la coloration syntaxique du template disparaît.

Données et méthodes

Maintenant que nous avons donné à ce composant son template, ou sa structure, nous devons également définir ses données et ses méthodes, qui sont toujours dans **main.js**. Nous allons donc les coller maintenant :

Composants/ProductDisplay.js

```
app.component('product-display', {
  template:
    /*html*/
    `

Auteur : www.vuemastery.com  
Modifié par : Grégory Charmier



Page 54 sur 83



Création : 26.11.2023  
Impression : 07/12/2023  
C-294-ALL01-Introduction-vue-js-  
3.docx



Version : 1


```

```
    updateVariant(index) {
      this.selectedVariant = index
    },
    computed: {
      title() {
        return this.brand + ' ' + this.product
      },
      image() {
        return this.variants[this.selectedVariant].image
      },
      inStock() {
        return
this.variants[this.selectedVariant].quantity
      }
    }
  })
```

Nous avons supprimé `cart` de la `data` parce que nous n'avons pas besoin que chaque produit dispose de son propre panier.

Nettoyage de `main.js`

Maintenant que nous avons encapsulé tout ce code spécifique au produit dans notre Composant `product-display`, nous pouvons nettoyer notre fichier **`main.js`**.

- `main.js`

```
const app = Vue.createApp({
  data() {
    return {
      cart: 0,
    }
  },
  methods: {}
})
```

Nous avons conservé le `cart` et la `methods` car nous en aurons besoin plus tard.

Importation du composant

Afin d'utiliser `product-display`, nous devons l'importer dans notre **index.html**.

index.html

```
<!-- Import Components -->
<script src="./components/ProductDisplay.js"></script>
```

Attention ! vous devez placer le code ci-dessus à cet endroit :

```
<!-- Import App -->
<script src="./main.js"></script>

<!-- Import Components -->
<script src="./components/ProductDisplay.js"></script>

<!-- Mount App -->
<script>
  const mountedApp = app.mount("#app");
</script>
```

Maintenant qu'il est importé, nous pouvons l'utiliser dans notre modèle.

index.html

```
<div id="app">
  <div class="nav-bar"></div>

  <div class="cart">Cart({{ cart }})</div>
  <product-display></product-display>
</div>
```


Dans le navigateur, nous verrons que tout apparaît encore comme avant, mais maintenant que nous avons réorganisé les choses, le bouton « Ajouter au panier » n'incrémente plus le panier. Nous y remédierons dans la prochaine leçon.

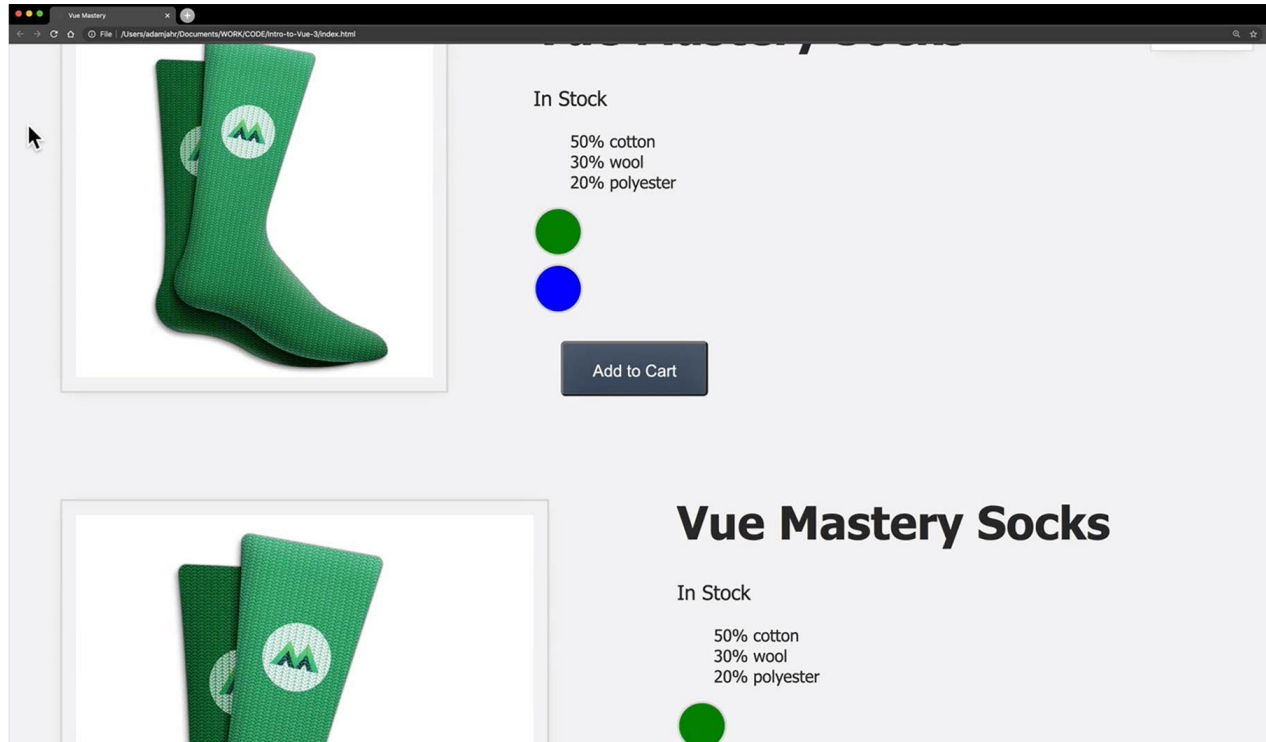
Pour l'instant, pour vous montrer à quel point ces blocs de code réutilisables peuvent être utiles, je vais ajouter deux autres composants `product-display`.

index.html

```
<div id="app">
  <div class="nav-bar"></div>

  <div class="cart">Cart({{ cart }})</div>
  <product-display></product-display>
  <product-display></product-display>
  <product-display></product-display>
</div>
```

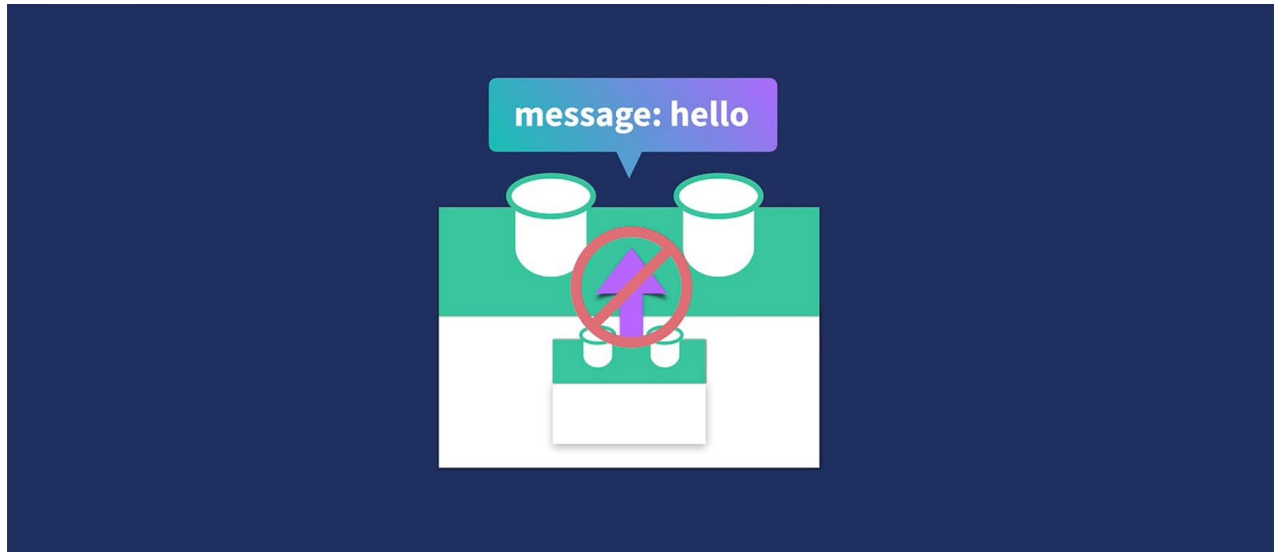
Lorsque nous rafraîchirons le navigateur, nous les verrons tous apparaître. Chacun d'eux est fonctionnel indépendamment.



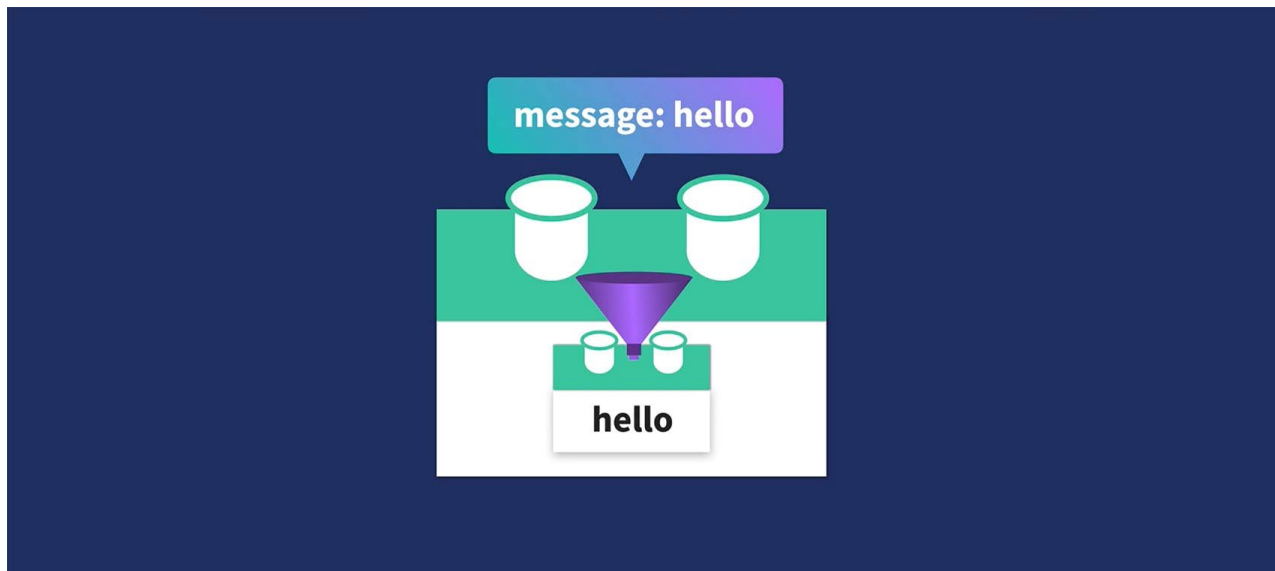
9.4 Props

Maintenant que nous commençons à comprendre comment encapsuler le code dans des composants, que se passe-t-il quand notre composant a besoin de quelque chose qui est à l'extérieur du composant ?

Par exemple, Imaginons que le parent a un message et que l'enfant en a besoin. Comme un composant a sa propre portée et que celle-ci est isolée, il ne peut pas atteindre quelque chose qui à l'extérieur de lui-même.



La réponse ici est **les Props**. Ce sont des attributs personnalisés pour transmettre des données dans un composant. Ils fonctionnent un peu comme un entonnoir, dans lequel vous pouvez passer les données dont le composant a besoin.



Ajoutons la capacité à notre composant product-display d'utiliser un Props.

9.5 Donner un Props à notre composant

Donnons à notre application Vue racine, située dans **main.js**, une nouvelle propriété de données, qui indique si l'utilisateur est un utilisateur premium ou non.

- **main.js**

```
const app = Vue.createApp({
  data() {
    return {
      cart: 0,
      premium: true
    }
  }
})
```

Si un utilisateur est premium, les frais de port sont offerts. Donc notre composant product-display doit avoir accès à ces données. En d'autres termes, il a besoin d'un attribut personnalisé dans lequel nous pouvons alimenter ces données.

Nous allons donc ajouter à notre composant une option props et ajouter premium au props.

Composants/ProductDisplay.js

```
app.component('product-display', {
  props: {
    premium: {
      type: Boolean,
      required: true
    }
  },
  ...
})
```

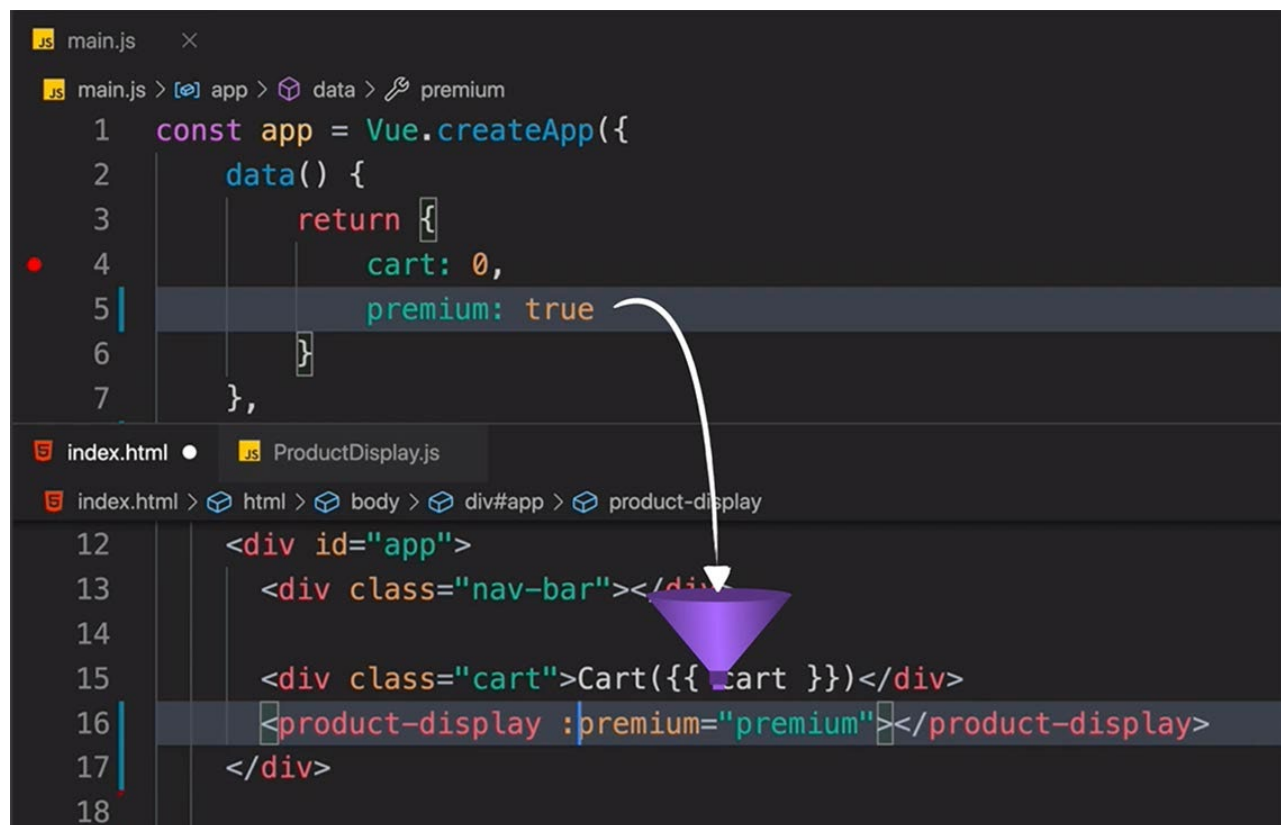
Remarquez comme la fonctionnalité Props de Vue a une validation intégrée, de sorte que nous pouvons spécifier des choses comme le type de Props et si premium est required, etc.

Maintenant que nous avons configuré ceci, nous pouvons ajouter cet attribut personnalisé sur le composant product-display où nous l'utilisons.

index.html

```
<div id="app">
  <div class="nav-bar"></div>

  <div class="cart">Cart({{ cart }})</div>
  <product-display :premium="premium"></product-display>
</div>
```



Il est important de noter comment nous utilisons le raccourci v-bind afin que nous puissions recevoir de manière réactive la nouvelle valeur de premium si ce dernier se met à jour (de true à false).

9.6 Utilisation de Props

Maintenant que notre composant product-display a un Props premium nous pouvons l'utiliser au sein du composant. Rappelez-vous, nous voulons utiliser le fait qu'un utilisateur est premium afin de déterminer ce qu'il doit payer comme frais de port.

Dans le template du composant, nous ajouterons :

Composants/ProductDisplay.js

```
template:
  /*html*/
  `<div class="product-display">
    ...
    <p>Shipping: {{ shipping }}</p>
    ...
  </div>`,
```

Ici, shipping est le nom d'une nouvelle propriété calculée sur le composant product-display qui ressemble à ceci :

Composants/ProductDisplay.js

```
computed: {
  ...
  shipping() {
    if (this.premium) {
```

```
    return 'Free'  
  }  
  return 2.99  
}  
}
```

La propriété calculée vérifie si premium est true, et si oui, retourne 'Free'. Sinon, il revient 2.99

9.7 A vous de coder !

Nous avons atteint la fin de la leçon et voici le nouveau défi :

Créer un nouveau composant appelé 'product-details', qui reçoit les détails du produit par l'intermédiaire d'un Props appelé details. A noter que le composant ProductDisplay.js a déjà dans une donnée details et que vous devez utiliser

Pour rappel, vous pouvez vérifier votre code grâce à la [branche](#) L9-end.

10. Communiquer grâce aux événements

Dans cette leçon, nous allons étudier le concept d'événements permettant la communication entre composants.

Vous pouvez trouver le code de départ dans la [branche](#) L10-start.

10.1 Notre objectif

Notre objectif est d'offrir à notre composant la possibilité de transmettre à ses parents un événement qui s'est produit en son sein.

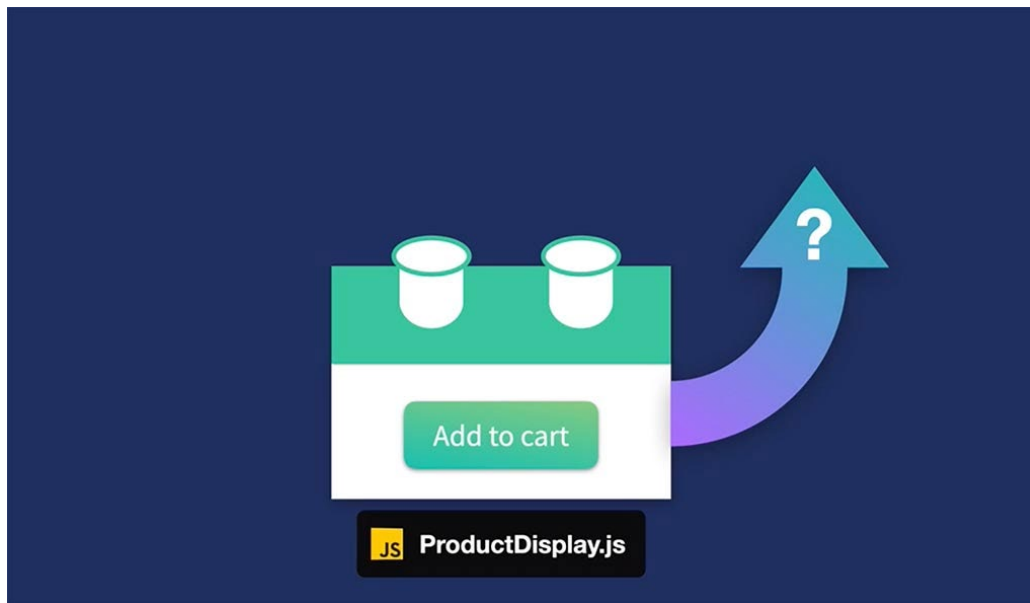
10.2 Émettre l'événement

Quand nous avons refactorisé le code dans notre dernière leçon, nous avons déplacé le code lié au produit dans le nouveau composant product-display. En faisant cela, nous avons cassé le fonctionnement du bouton « Ajouter au panier » qui permet d'incrémenter la valeur de cart.

Pourquoi ? Car `cart` se trouve à l'intérieur de l'application racine `Vue` dans **`main.js`** et qu'il est donc situé **en dehors** du « scope » du composant `product-display`.

Nous devons donner au composant `product-display` un moyen d'annoncer que son bouton est cliqué. Comment faire en sorte que cela se produise ?

Nous savons déjà que les **Props** sont un moyen de transmettre les données dans un composant, mais qu'en est-il quand quelque chose se passe dans ce composant, comme un clic de bouton ? Comment faire savoir à d'autres parties de notre application que cet événement s'est produit ?



La réponse est d'émettre cet événement en prévenant le composant parent que cela s'est produit. Ajoutons cela dans notre composant `product-display` en modifiant la méthode `addToCart()`.

Composants/ProductDisplay.js

```
methods: {
  addToCart() {
    this.$emit('add-to-cart')
  }
  ...
}
```

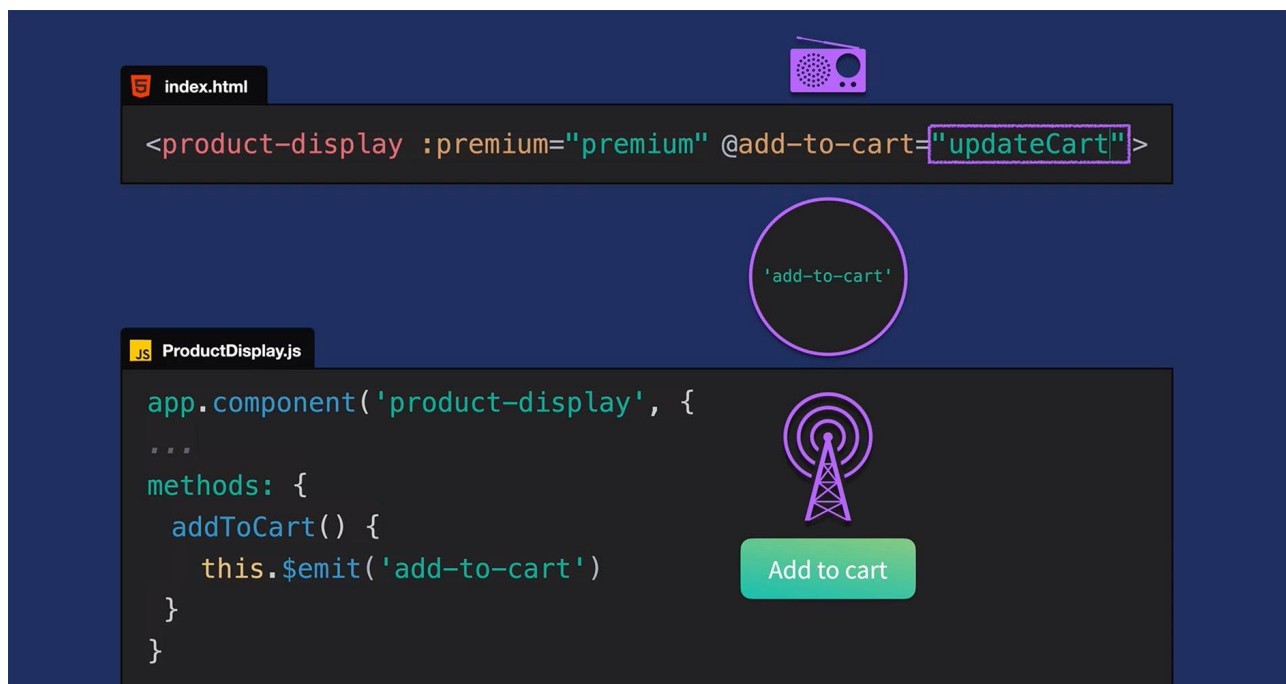

Nous allons grâce à `this.$emit()` émettre un événement appelé 'add-to-cart'. Donc, quand le bouton est cliqué, nous allons émettre cet événement.

Ensuite, nous pouvons écouter cet événement à partir du « scope » parent, où nous utilisons `product-display`, en ajoutant un listener : `@add-to-cart`.

index.html

```
<product-display :premium="premium" @add-to-cart="updateCart"></product-display>
```

Lorsque cet événement est « entendu » par le parent, il déclenche une nouvelle méthode ayant pour nom `updateCart`, que nous ajouterons dans **main.js**.



- main.js

```
const app = Vue.createApp({
  data() {
    return {
      cart: [],
      ...
    }
  }
})
```

```
    }  
  },  
  methods: {  
    updateCart() {  
      this.cart += 1  
    }  
  }  
})
```

Si nous vérifions cela dans le navigateur, nous devrions maintenant pouvoir cliquer sur le bouton « Ajouter au panier », ce qui permet aux parents de savoir que l'événement `add-to-cart` s'est produit, ceci déclenchant la méthode `updateCart()`.

10.3 Ajouter l'identifiant du produit au panier

Pour rendre notre application plus réaliste, notre `cart` ne devrait pas être un nombre. Il devrait s'agir d'un tableau qui contient les ids des produits qui y sont ajoutés. Nous allons donc faire un peu de refactoring.

- **main.js**

```
const app = Vue.createApp({  
  data() {  
    return {  
      cart: [],  
      ...  
    }  
  },  
  methods: {  
    updateCart(id) {  
      this.cart.push(id)  
    }  
  }  
})
```

Maintenant, `cart` est un tableau et `updateCart(id)` « push » l'id produit. Nous avons juste besoin d'ajouter une donnée (payload) lors de l'émission de l'événement `add-to-cart`, de sorte que `updateCart` ait accès à cet id.

Composants/ProductDisplay.js

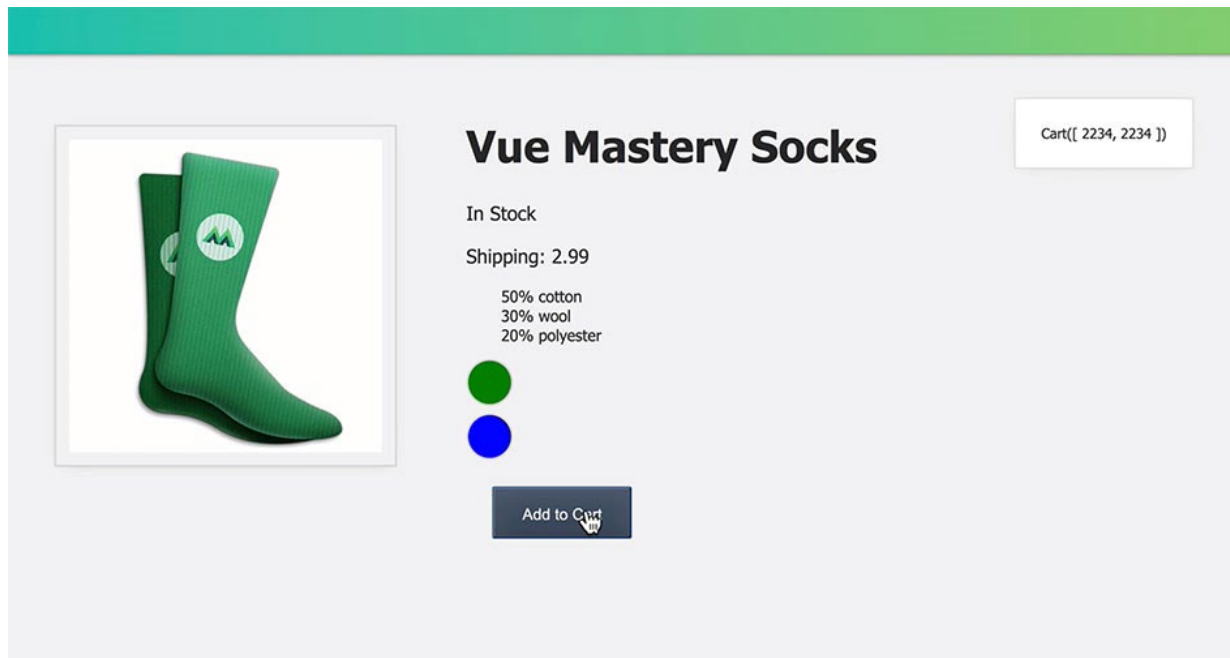
```
methods: {  
  addToCart() {  
    this.$emit('add-to-cart',  
this.variants[this.selectedVariant].id)  
  }  
  ...  
}
```

Ici, nous avons ajouté un deuxième paramètre à savoir l'id du produit. Un peu comme nous avons fait pour récupérer l'image et la quantity précédemment :

Composants/ProductDisplay.js

```
computed: {  
  image() {  
    return this.variants[this.selectedVariant].image  
  },  
  inStock() {  
    return this.variants[this.selectedVariant].quantity  
  }  
}
```

Maintenant, dans le navigateur, vous pouvez voir que nous ajoutons les ids de produit dans le panier (ce panier qui est maintenant un tableau).



Mais nous n'avons pas besoin de montrer ces ids. Nous voulons juste afficher le nombre d'articles dans le panier. Heureusement, c'est simple à faire.

index.html

```
<div id="app">
  ...
  <div class="cart">Cart({{ cart.length }})</div>
  ...
</div>
```

En ajoutant `cart.length` nous n'afficherons plus que la *quantité* d'éléments dans le cart.

10.4 A vous de coder !

Nous avons atteint la fin de la leçon et voici le nouveau défi :

Ajouter un nouveau bouton au composant `product-display`, qui retire le produit du panier `cart`.

Pour rappel, vous pouvez vérifier votre code grâce à la [branche](#) L10-end.

Remarque, dans la solution, on ne cherche à définir proprement la condition permettant de déterminer s'il faut activer le bouton « Remove Item ».

11. Formulaires et modèle v

Dans cette leçon, nous allons étudier le concept de liaison des attributs. Vous pouvez trouver le code de départ dans la [branche](#) L11-start.

11.1 Notre objectif

Créer un formulaire permettant aux utilisateurs d'ajouter des critiques/avis sur les produits.

11.2 Introduction à la directive v-model

Au début de ce cours, nous avons appris que `v-bind`, permet de créer une liaison unidirectionnelle, des données au template. Cependant, lorsque l'on travaille avec des formulaires, cette liaison unidirectionnelle n'est pas suffisante. Nous devons également créer une liaison du template aux données.

Par exemple, lorsqu'un utilisateur saisit son nom dans un champ d'entrée, nous voulons enregistrer et stocker cette valeur dans nos données. La directive `v-model` nous aide à atteindre cet objectif, en créant une **liaison** de données **bidirectionnelle**.



Pour voir tout cela en action, nous allons créer un nouveau composant review-form.

11.3 Le composant ReviewForm

Nous ajoutons un nouveau fichier **ReviewForm.js** dans notre dossier de composants, et nous allons mettre en place la structure de ce composant.

Composants/ReviewForm.js

```
app.component('review-form', {  
  template:  
    /*html*/  
    `<form class="review-form">  
      <h3>Leave a review</h3>  
      <label for="name">Name:</label>  
      <input id="name">  
  
      <label for="review">Review:</label>  
      <textarea id="review"></textarea>  
  
      <label for="rating">Rating:</label>
```

```
<select id="rating">
  <option>5</option>
  <option>4</option>
  <option>3</option>
  <option>2</option>
  <option>1</option>
</select>

<input class="button" type="submit" value="Submit">
</form>`,
data() {
  return {
    name: '',
    review: '',
    rating: null
  }
}
```

À l'intérieur de notre template, on peut noter les éléments suivants :

- `<input id="name">`
- `<textarea id="review">`
- `<select id="rating">`

Nous voulons lier ces champs aux données de sorte que lorsque l'utilisateur remplit le formulaire, nous stockons leurs données localement.

```
data() {
  return {
    name: '',
    review: '',
    rating: null
  }
}
```

Nous y parviendrons en ajoutant la directive **v-model** à chacun des éléments de saisie.

Composants/ReviewForm.js

```
app.component('review-form', {
  template:
    /*html*/
    `<form class="review-form">
      <h3>Leave a review</h3>
      <label for="name">Name:</label>
      <input id="name" v-model="name">

      <label for="review">Review:</label>
      <textarea id="review" v-model="review"></textarea>

      <label for="rating">Rating:</label>
      <select id="rating" v-model.number="rating">
        <option>5</option>
        <option>4</option>
        <option>3</option>
        <option>2</option>
        <option>1</option>
      </select>

      <input class="button" type="submit" value="Submit">
    </form>`,
  data() {
    return {
      name: '',
      review: '',
      rating: null
    }
  }
})
```


Remarquez comment sur le tag HTML `<select>`, nous avons utilisé `v-model.number`, il s'agit d'un modificateur qui convertit la valeur en nombre.

Soumettre le formulaire ReviewForm

Pour soumettre ce formulaire, nous ajouterons un **listener** en haut au niveau du tag HTML form :

Composants/ReviewForm.js

```
app.component('review-form', {
  template:
    /*html*/
    `<form class="review-form" @submit.prevent="onSubmit">
      ...
      <input class="button" type="submit" value="Submit">
    </form>`
  ...
})
```

Nous utilisons un autre modificateur `@submit.prevent="onSubmit"` afin d'éviter le comportement par défaut (un rafraîchissement du navigateur). Lorsque ce formulaire est soumis, il déclenche la méthode `onSubmit()`, que nous allons écrire maintenant :

Composants/ReviewForm.js

```
...
data() {
  return {
    name: '',
    review: '',
    rating: null
  }
},
methods: {
  onSubmit() {
```

```
let productReview = {
  name: this.name,
  review: this.review,
  rating: this.rating,
}
this.$emit('review-submitted', productReview)

this.name = ''
this.review = ''
this.rating = null
}
}
...
```

Cette méthode créera un objet `productReview` contenant `name`, `review` et `rating` provenant de nos `data`. Cet objet sera alors `$emit` avec l'événement `review-submitted`, en passant en paramètre l'objet `productReview`.

Enfin, nous vidons les champs de données.

Utilisation du formulaire de ReviewForm

Maintenant que notre formulaire pour les avis est créé, nous pouvons l'importer dans **index.html**.

index.html

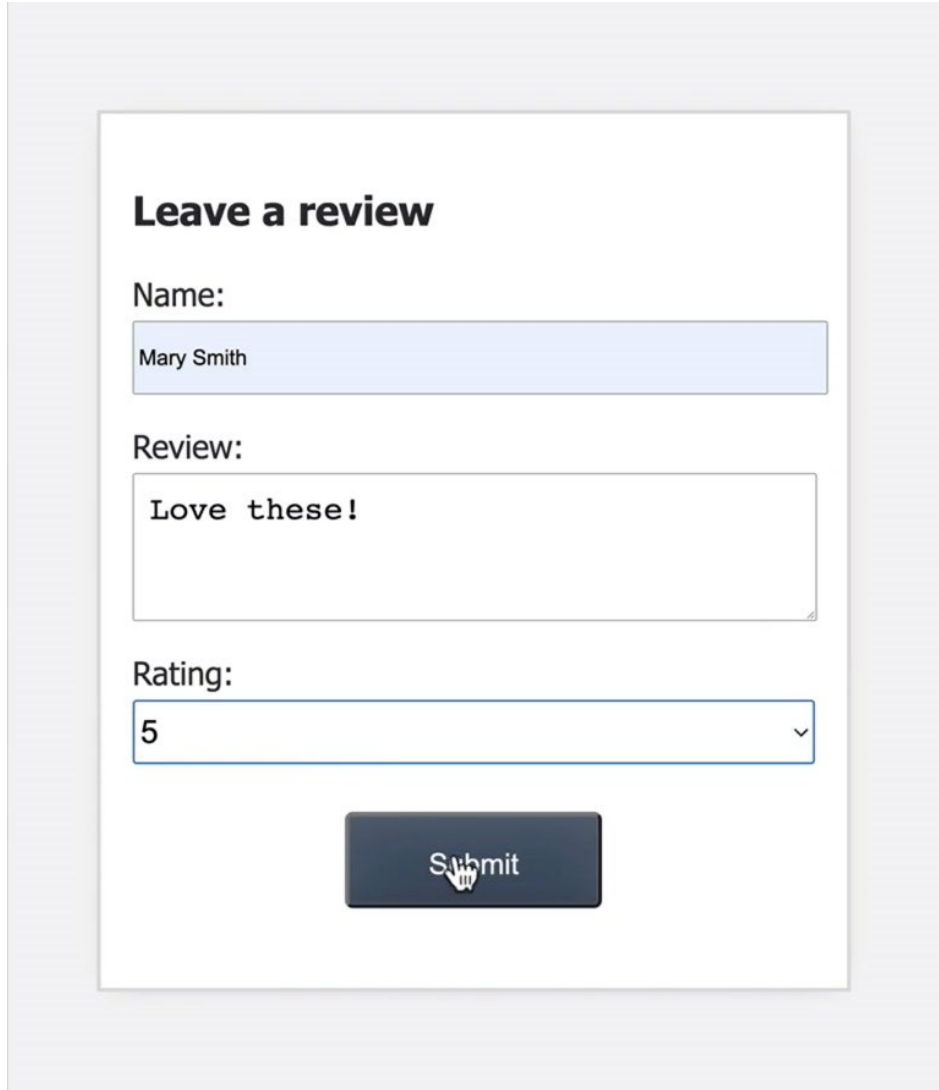
```
<!-- Import Components -->
...
<script src="./components/ReviewForm.js"></script>
...
```

Maintenant dans le composant `product-display`, nous pouvons utiliser le composant `ReviewForm`.

Composants/ProductDisplay.js

```
template:
  /*html*/
  `<div class="product-display">
    <div class="product-container">
      ...
    </div>
    <review-form></review-form>
  </div>`
  `)
```

Dans le navigateur, nous pouvons normalement voir le formulaire d'avis.



Il semble que ça marche... sauf que quand on clique sur le bouton de soumission, on émet l'événement, mais on n'a pas mis en place le code pour écouter cet événement.

Comme nous l'avons appris dans la leçon précédente, nous devons écouter l'évènement `review-submitted` dans le scope du parent (à savoir `product-display`).

Quand l'événement sera « entendu », nous ajouterons la donnée (payload) `productReview` aux données du composant `product-display`.

Ajout d'avis des produits

Nous ajouterons l'événement sur review-form, où il est utilisé :

Composants/ProductDisplay.js

```
template:
  /*html*/
  `<div class="product-display">
    <div class="product-container">
      ...
    </div>
    <review-form @review-submitted="addReview"></review-form>
  </div>`
  })
```

Quand l'événement se produira, nous déclencherons une nouvelle méthode `addReview()`. Cela ajoutera les avis de produits à notre composant `product-display`, ce qui signifie que le composant a besoin d'un nouveau tableau `reviews` dans ses données.

Composants/ProductDisplay.js

```
...
data() {
  return {
    ...
    reviews: []
  }
}
...
```

Maintenant, occupons nous de la méthode `addReview()` :

Composants/ProductDisplay.js

```
...  
data() {  
  return {  
    ...  
    reviews: []  
  }  
},  
methods: {  
  ...  
  addReview(review) {  
    this.reviews.push(review)  
  }  
},  
...
```

Comme vous pouvez le voir, nous prenons le paramètre `review` que nous avons obtenu grâce aux données (payload) de l'événement `review-submitted`, et on le « push » dans le tableau `reviews`.

11.4 Affichage des avis/critiques

Maintenant que nous avons mis en œuvre la possibilité d'ajouter des critiques/avis, nous devons les afficher. Créons un nouveau composant pour cela. Ce composant sera appelé `review-list` et nous allons le construire ainsi :

Composants/ReviewList.js

```
app.component('review-list', {  
  props: {  
    reviews: {  
      type: Array,  
      required: true  
    }  
  },  
  template:
```

```
/*html*/
<div class="review-container">
  <h3>Reviews:</h3>
  <ul>
    <li v-for="(review, index) in reviews" :key="index">
      {{ review.name }} gave this {{ review.rating }}
stars
      <br/>
      "{{ review.review }}"
      <br/>
    </li>
  </ul>
</div>
})
})
```

Il aura un **props** de sorte qu'il puisse recevoir les reviews et les afficher dans le template en utilisant v-for avec index afin que nous puissions lier l'attribut :key à celui-ci.

Maintenant nous pouvons importer ce composant à l'intérieur de **index.html** :

index.html

```
<!-- Import Components -->
...
<script src="./components/ReviewList.js"></script>
...
```

Puis il faut ajouter le code suivant à l'intérieur du composant product-display, juste au-dessus de review-form :

Composants/ProductDisplay.js

```
template:
  /*html*/
  `<div class="product-display">
    <div class="product-container">
      ...
    </div>
    <review-list :reviews="reviews"></review-list>
    <review-form @review submitted="addReview"></review-form>
  </div>`
  `)
```

Il est important de noter la manière dont nous avons ajouté `:reviews="reviews"` afin de passer les reviews à afficher sur le composant `product-display` dans la `review-list`.

Vous pouvez vérifier que tout est ok dans le navigateur. C'est-à-dire, nous pouvons ajouter un nouvel avis, et voir l'avis s'afficher.

Reviews:

Mary Smith gave this 5 stars
"Love these!"

Leave a review

Name:

Review:

Rating:

Quand nous rafraîchissons la page (et qu'il n'y a pas d'avis/critiques), nous voyons toujours une boîte vide parce que le composant `review-list` est encore en cours d'affichage sans avis à afficher. Réparons cela, et n'affichons ce composant que lorsque nous avons des reviews à afficher.

Composants/ProductDisplay.js

```
template:
  /*html*/
  `<div class="product-display">
    ...`
```

```
<review-list v-if="reviews.length"
:reviews="reviews"></review-list>
...
</div>`
})
```

En d'autres termes, si le tableau `reviews` est vide, nous ne montrerons pas le composant `review-list`.

Après avoir rafraîchi la page, il semble que ça marche, et que le composant ne s'affiche qu'après avoir ajouté un avis.

11.5 Validation du formulaire

Pour terminer cette leçon, nous allons ajouter une validation très basique à notre `review-form`.

Composants/ReviewForm.js

```
methods: {
  onSubmit() {
    if (this.name === '' || this.review === '' ||
this.rating === null) {
      alert('Review is incomplete. Please fill out every
field.')
      return
    }
    ...
  }
}
```

Avant de créer un `productReview`, nous vérifierons si `this.name` ou `this.review` ou `this.rating` sont vides. Si c'est le cas, nous afficherons un alert, qui affiche : « Review is incomplete. Please fill out every field. ».

Il s'agit d'une méthode de validation extrêmement simplifiée. D'autres méthodes plus complètes et robustes existent bien sûr en `Vue.js`.

11.6 A vous de coder !

Nous avons atteint la fin de la leçon et voici le nouveau défi :

Ajouter une question au review-form: « Recommanderiez-vous ce produit ? »

Il s'agit d'un tag HTML **select** avec les options **Yes** et **No**.

Enregistrer et émettre la réponse, et l'afficher à l'intérieur de review-list

Pour rappel, vous pouvez vérifier votre code grâce à la [branche](#) L11-end.

12. Conclusion

Bravo! vous êtes arrivés au bout de cette introduction à Vue.js. Vous allez pouvoir mettre en application toutes ces nouvelles compétences dans le prochain exercice.