

```
src\vue-pratique\src\components\Calculator.vue
4 <template>
5   <input type="text" v-model="n1" placeholder="Enter first number"/>
6   <input type="text" v-model="n2" placeholder="Enter second number"/>
7   <br>
8   <script>
9     import { ref } from 'vue';
10    import { defineProps, defineEmits } from 'vue';
11    import { onMounted } from 'vue';
12    import { watch } from 'vue';
13    export default {
14      setup() {
15        let n1 = ref('');
16        let n2 = ref('');
17        let result = ref('');
18        function calc() {
19          result.value = `The sum is ${n1.value + n2.value}`;
```

3

Manuel pratique

Table des matières

[Un cadeau avant de commencer la lecture](#)

[Partie 1 – Premiers pas en Vue.js 3](#)

[Pourquoi Vue.js 3](#)

[Préparation de l'environnement de développement](#)

[L'éditeur de code Visual Studio Code](#)

[Les extensions que vous devez utiliser dans VS Code](#)

[L'extension Emmet](#)

[L'extension Vetur](#)

[L'extension Live Server](#)

[Une invite de commande dans VS Code](#)

[Le modèle MVVM de Vue.js](#)

[Un premier projet Vue.js](#)

[Un deuxième exemple \(projet3\)](#)

[Propriétés calculées](#)

[Méthodes ou propriétés calculées ?](#)

[Data binding](#)

[innerHTML et nœud texte](#)

[Binding bidirectionnel](#)

[Utilisation conjointe des directives v-model et v-bind](#)

[Gestion évènementielle](#)

[L'objet \\$event dans la gestion évènementielle](#)

[Les fonctions stopPropagation\(\) et preventDefault\(\) de l'objet \\$event](#)

[Suffixes stop et prevent de Vue](#)

[Affichage des langages dans une liste à puces](#)

[Deux paramètres dans la directive v-for](#)

[Arguments dynamiques](#)

[Directive v-for pour parcourir des objets JSON](#)

[Gestion des évènements clavier](#)

[La directive v-cloak](#)

[La directive v-once](#)

[Classes conditionnelles](#)

[Gestion des styles inline](#)

[Afficher du code HTML conditionnellement](#)

[Afficher du code HTML conditionnellement](#)

[Partie 2 – Les composants Vue.js](#)

[Un premier composant global](#)

[Un premier composant local](#)

[Ajouter des données dans un composant](#)

[Ajouter des méthodes dans un composant](#)

[Passer des données aux composants enfants avec des props](#)

[Accès à une prop dans le modèle du composant](#)

[Passer des données de l'application aux composants enfants](#)

[Custom-events - Envoi de messages au parent par un composant avec \\$emit](#)

[Lifecycle Hooks](#)

[Organiser une application en hiérarchie de composants](#)

[Passer des données à un composant avec des slots](#)

[Slots simples](#)

[Slots nommés](#)

[Slots avec portée \(scoped slots\)](#)

[Filtres](#)

[Les limites des expressions calculées](#)

[Observateurs \(watchers\)](#)

[Requêtes asynchrones dans un observateur](#)

[Animations et transitions](#)

[Transitions d'entrée et de sortie](#)

[Transitions d'entrée et de sortie avec Vue.js](#)

[Transitions d'entrée et de sortie non simultanées](#)

[Transitions entre éléments](#)

[Transitions de listes](#)

[Transitions d'entrée et de sortie avec animate.css](#)

[Animations en JavaScript](#)

[Partie 3 - Les applications SPA \(*Single Page Applications*\)](#)

[Création d'une application SPA avec vue-router](#)

[Allure du lien actif](#)

[Indicateurs de navigation - Vue Router Navigation Guards](#)

[Indicateurs de navigation globaux](#)

[Indicateurs de navigation par route](#)

[Indicateurs de navigation par composant](#)

[Routage – Transition entre les pages](#)

[Partie 4 - L'interface en ligne de commande Vue-Cli](#)

[L'outil Vue-Cli](#)

[Architecture de l'application](#)

[Le fichier public/index.html](#)

[Les fichiers .vue](#)

[Le fichier App.vue](#)

[Le fichier Main.js](#)

[Le fichier components/HelloWorld.vue](#)

[Création et modification d'une application avec Vue-Cli](#)

[Une deuxième application avec Vue-Cli](#)

[Fonctions de rendu \(*Render Functions*\)](#)

[Templates vs render functions & JSX](#)

[Composants fonctionnels](#)

[Créer un composant fonctionnel](#)

Utilisation des paramètres props et context

Partie 5 - Le gestionnaire d'état Vuex

Installation de Vuex

Une première application Vuex

Accès aux méthodes du store dans les composants

Accès à l'état du store dans les composants

L'option plugins de Vuex

Alternative à Vuex

Vuex et les formulaires

Validation d'un formulaire dans Vue.js

Partie 6 - Plus loin avec Vue.js

Mixins

Mixin local

Mixin global

Plugins

Directives personnalisées

Directives personnalisées – Un premier exemple

Directive personnalisée avec arguments

Rendre son code robuste grâce aux props typées

NuxtJS

Développement d'applications mobiles

Le composant <teleport></teleport>

L'API de composition

Première approche – Utilisation de la fonction ref()

Deuxième approche – Utilisation de la fonction reactive()

Troisième approche – Utilisation de la fonction computed()

Quatrième approche – Externalisation du calcul réactif

Vue 3 et TypeScript

Le mot de la fin

Bonjour et bienvenue dans cette formation dédiée à Vue.js 3, le framework JavaScript qui a le vent en poupe. Je m'appelle Michel Martin et je vais vous accompagner tout au long de ce manuel très complet pour débutants. Pour être en mesure de la suivre, vous devez avoir des notions de HTML, de CSS et de JavaScript. Rien d'autre !

Vous avez pris une excellente décision en décidant de vous former à Vue 3 et je suis vraiment content de vous retrouver ici.

Dans les dix dernières années, les pages Web sont devenues de plus en plus puissantes et dynamiques grâce au langage JavaScript. Beaucoup de code qui s'exécutait côté serveur s'exécute désormais côté client. Le code JavaScript devient de plus en plus conséquent et difficile à gérer sur de gros projets.

C'est là qu'interviennent les frameworks JavaScript, tels que Vue, React et Angular.

De nos jours, les applications Web sont de plus en plus courantes et de plus en plus complexes. À un tel point que JavaScript aussi montre ses limites. Aujourd'hui, les frameworks modernes, tels que Vue, React et Angular offrent la possibilité de coder en TypeScript. Cette formation va vous montrer comment coder des projets Vue.js 3 en *Vanilla JavaScript* (c'est-à-dire en pur JavaScript) et en *TypeScript*.

Comme vous allez le voir tout au long de ce manuel, Vue 3 est performant et relativement facile à prendre en main pour qui connaît déjà les langages JavaScript, HTML et CSS.

Contrairement à Angular et React, Vue.js vous permet d'incorporer du code Vue.js dans des projets existants, sans tout devoir réimaginer. Bien entendu, vous pouvez créer des projets complets avec Vue.js, en partant d'une feuille blanche.

Vue.js permet de découper vos applications Web en composants réutilisables, chacun possédant son propre code HTML, CSS et JavaScript/TypeScript. Vous pouvez même créer des sites Web dynamiques de type SPA (Single Page Application), ou des widgets que vous pourrez facilement intégrer à des projets Web existants.

Tout au long de cet ouvrage, vous trouverez des challenges dans lesquels vous devrez coder de petites applications en Vue.js 3. Jouez le jeu pour progresser encore plus vite dans votre apprentissage du framework. Une correction est systématiquement proposée à la suite de chaque challenge.

Happy Coding!

Un cadeau avant de commencer la lecture

Vous voulez compléter ce manuel par une formation vidéo de plus de 10 heures sur Vue.js 3 ?

Contactez-moi [en cliquant sur ce lien](#). Je vous enverrai un bon de réduction.

La formation sera accessible à 9,99 € ~~au lieu de 109,99 €~~ sur la plateforme Udemy.

Partie 1 – Premiers pas en Vue.js 3

Dans cette première partie, vous allez découvrir les principes de base de Vue.js 3 et vous allez commencer à programmer dans Visual Studio Code.

Pourquoi Vue.js 3

La version 3 de Vue.js a été publiée le 18 Septembre 2020. Une question se pose : devez-vous passer à la version 3 ou rester en version 2 ? Eh bien, plusieurs arguments font pencher la balance en faveur de la version 3 :

- Vue 3 est plus rapide que Vue 2 : la taille du bundle créé par Webpack est réduite de plus de 40%, et le temps de chargement de la page est réduit jusqu'à 55%.
- Le code source de Vue 3 a été complètement réécrit en TypeScript. Le langage TypeScript est donc pris en charge de façon native dans Vue 3.
- Une nouvelle API (la composition API) est proposée en alternative à l'API historique (Options API). Utilisée dans les projets de grande envergure, elle facilite l'organisation du code et le partage des éléments réactifs à tous les niveaux de l'application.

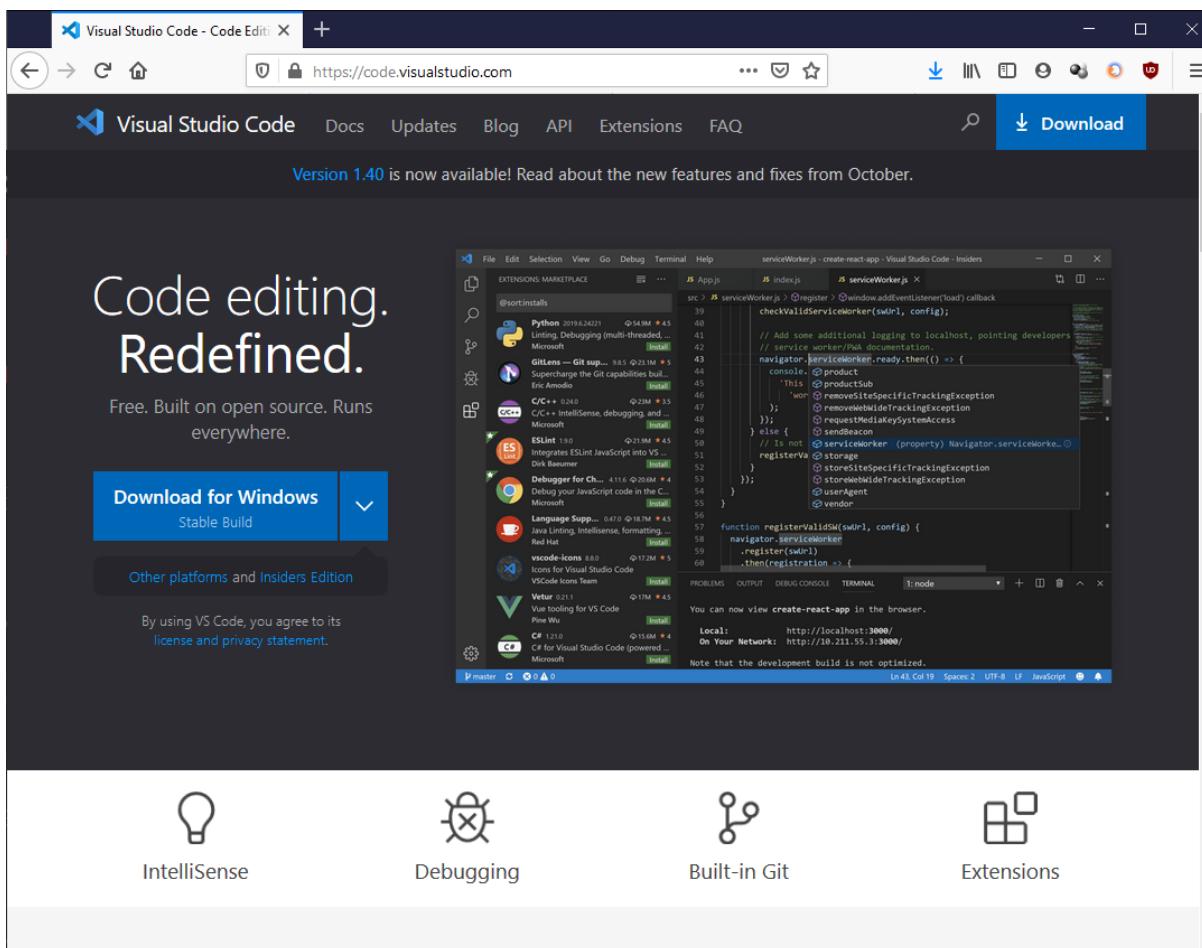
Préparation de l'environnement de développement

Après cette brève introduction, je vous propose d'installer l'environnement de développement que vous utiliserez tout au long de ce manuel.

L'éditeur de code Visual Studio Code

Allez sur <https://code.visualstudio.com/> et installez Visual Studio Code en cliquant sur le bouton :

- **Download for Windows** si votre ordinateur fonctionne sous Windows 10
- **Download Mac Universal** si votre ordinateur fonctionne sous Mac OSX
- **.deb** si votre ordinateur fonctionne sous une distribution Debian ou Ubuntu de Linux
- ou **.rpm** si votre ordinateur fonctionne sous une distribution Red Hat ou Fedora de Linux



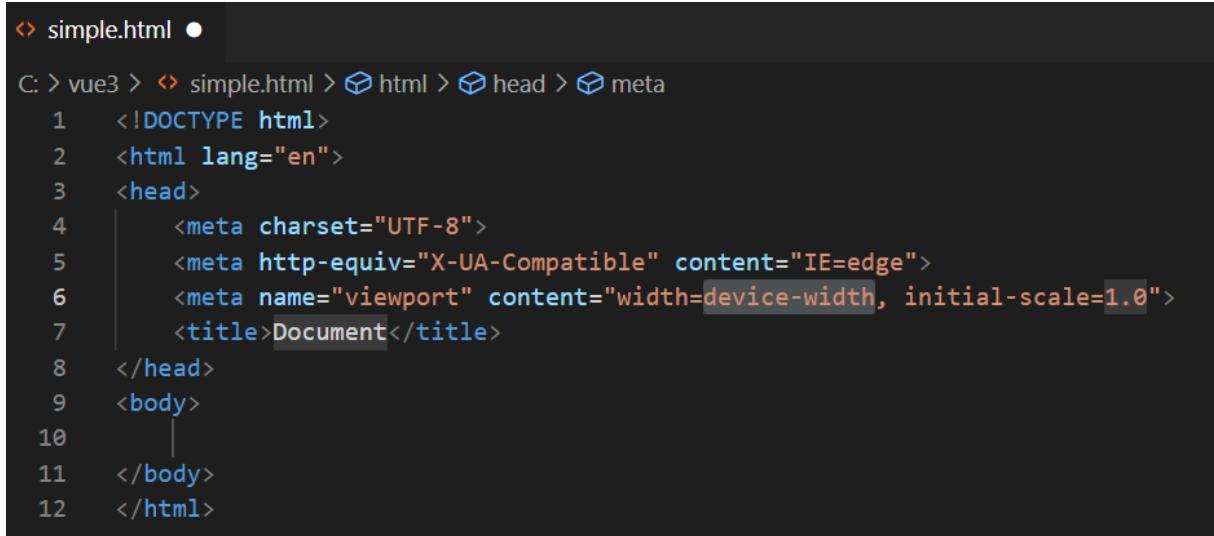
Les extensions que vous devez utiliser dans VS Code

Plusieurs extensions de Visual Studio Code sont vraiment intéressantes lorsque vous développez en Vue.js. Cette section va vous montrer comment les installer.

L'extension Emmet

L'extension **Emmet** est intégré dans VS Code. Vous l'utiliserez pour créer un squelette standard HTML5. Créez un nouveau fichier HTML. Pour cela, cliquez du bouton droit dans le volet gauche de VS Code et sélectionnez **New File** dans le menu contextuel. Donnez un nom quelconque au nouveau fichier et une extension **html**. Par exemple **test.html**.

Pour créer un squelette standard HTML5, tapez *html :5* et appuyez sur la touche *Entrée* du clavier. Voici le résultat :

A screenshot of the Visual Studio Code interface. The title bar shows 'simple.html'. The left sidebar shows a file tree with 'simple.html' selected. The main editor area displays the following HTML code:

```
C: > vue3 > <> simple.html > ⚡ html > ⚡ head > ⚡ meta
1   <!DOCTYPE html>
2   <html lang="en">
3     <head>
4       <meta charset="UTF-8">
5       <meta http-equiv="X-UA-Compatible" content="IE=edge">
6       <meta name="viewport" content="width=device-width, initial-scale=1.0">
7       <title>Document</title>
8     </head>
9     <body>
10    |
11     </body>
12   </html>
```

The code is syntax-highlighted, with tags in blue and attributes in orange. The Emmet abbreviation ':5' is expanded into the full HTML5 skeleton.

Les possibilités du plugin Emmet ne s'arrêtent pas là. Vous utiliserez plusieurs autres abréviations pour faciliter l'écriture de code HTML. Voyons les plus courantes.

Le signe **>** permet de générer une structure de balises imbriquées. Par exemple **nav>ul>li** crée une balise **** dans une balise **** dans une balise **<nav></nav>**.

Le signe * permet de créer plusieurs occurrences d'une même balise. Par exemple **li*4** crée quatre balises ****.

Le signe + permet de créer plusieurs balises non imbriquées. Par exemple **h1+h2+div** crée une balise **<h1></h1>** suivie d'une balise **<h2></h2>** suivie d'une balise **<div></div>**.

Les parenthèses permettent de regrouper plusieurs abréviations. Par exemple :

(nav>ul>li*3)+(p*2)+(h1+h2+div)

Crée les balises suivantes :

```
<nav>
  <ul>
    <li></li>
    <li></li>
    <li></li>
  </ul>
</nav>
<p></p>
<p></p>
<h1></h1>
<h2></h2>
<div></div>
```

Les accolades permettent d'insérer un contenu dans le innerHTML d'une balise. Par exemple :

div{contenu}

Crée la balise **<div></div>** et insère le texte contenu dans son innerHTML :

<div>contenu</div>

Le signe \$ permet de créer une incrémentation automatique dans une énumération. Par exemple,

ul>li{contenu \$}*5

Crée ces balises :

```
<ul>
    <li>contenu 1</li>
    <li>contenu 2</li>
    <li>contenu 3</li>
    <li>contenu 4</li>
    <li>contenu 5</li>
</ul>
```

Le signe **#** permet d'affecter un id à une balise. Par exemple :

div#perso

Crée la balise **<div id="perso"></div>**

Le signe **.** permet d'affecter une classe à une balise. Par exemple :

div.perso

Crée la balise **<div class="perso"></div>**

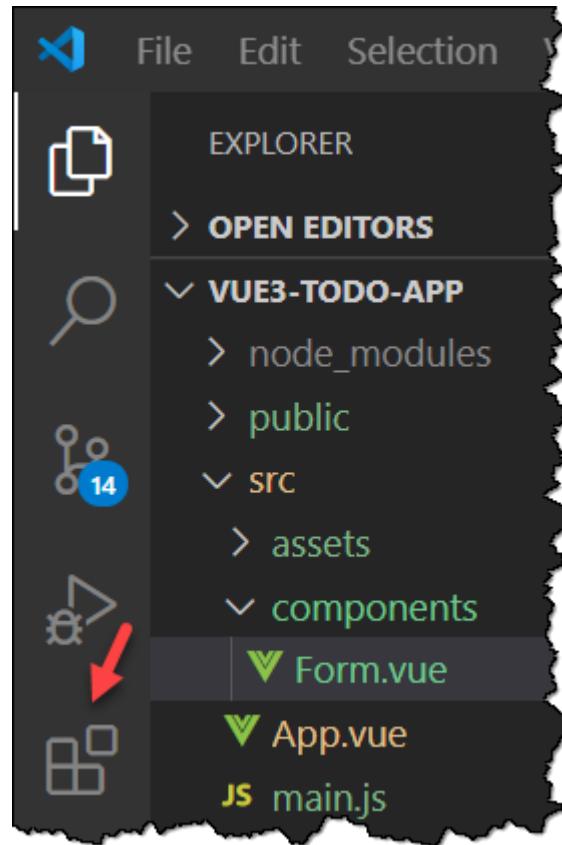
L'extension Vetur

L'extension **Vetur** (*Vue Tooling for VS Code*) ajoute de nombreuses fonctionnalités pour faciliter le développement Vue.js. Entre autres :

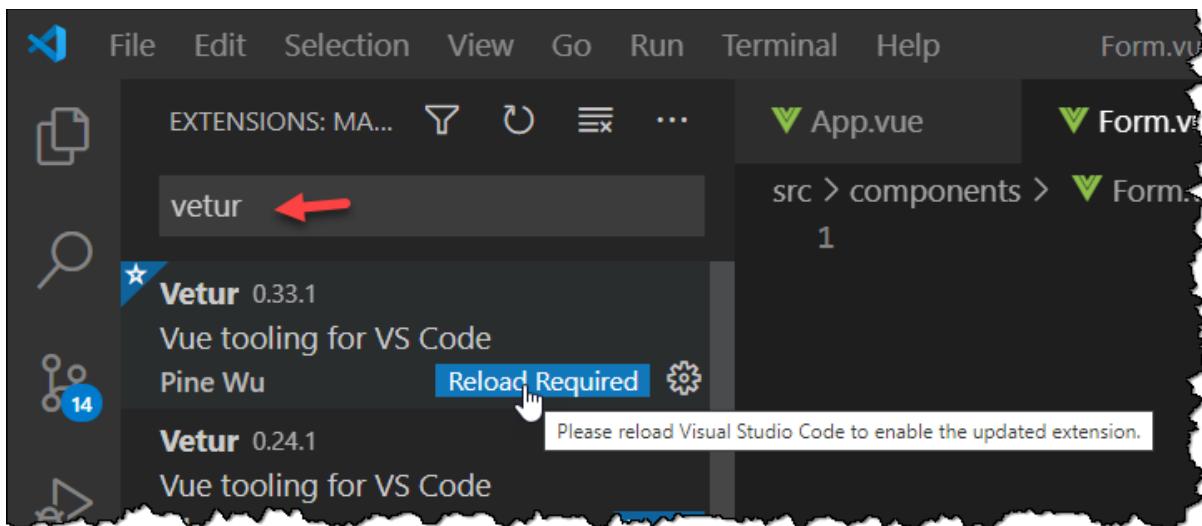
- Coloration syntaxique des instructions dans les fichiers d'extension **.vue**.
- Mise en concordance des accolades.
- Autocomplétion sur les attributs spécifiques à Vue.js dans la section template (**v-bind**, **v-show**, etc.) et sur les termes spécifique Vue.js dans la section **<script></script>** (**components**, **methods**, etc.).
- Création de snippets avec une intégration de Emmet.

Voyons comment installer l'extension Vetur.

Cliquez sur l'icône **Extensions** dans la partie supérieure gauche de la fenêtre :



Tapez **vetur** et cliquez sur **Reload Required** sous **Vetur** :



VS Code est rechargé.

Désormais, dans un fichier d'extension .vue, on peut taper :

<vue

Et appuyer sur la touche *Entrée* pour que le squelette du composant soit inséré dans le fichier vue :

```
<template>
```

```
</template>
```

```
<script>
```

```
export default {
```

```
}
```

```
</script>
```

```
<style>
```

```
</style>
```

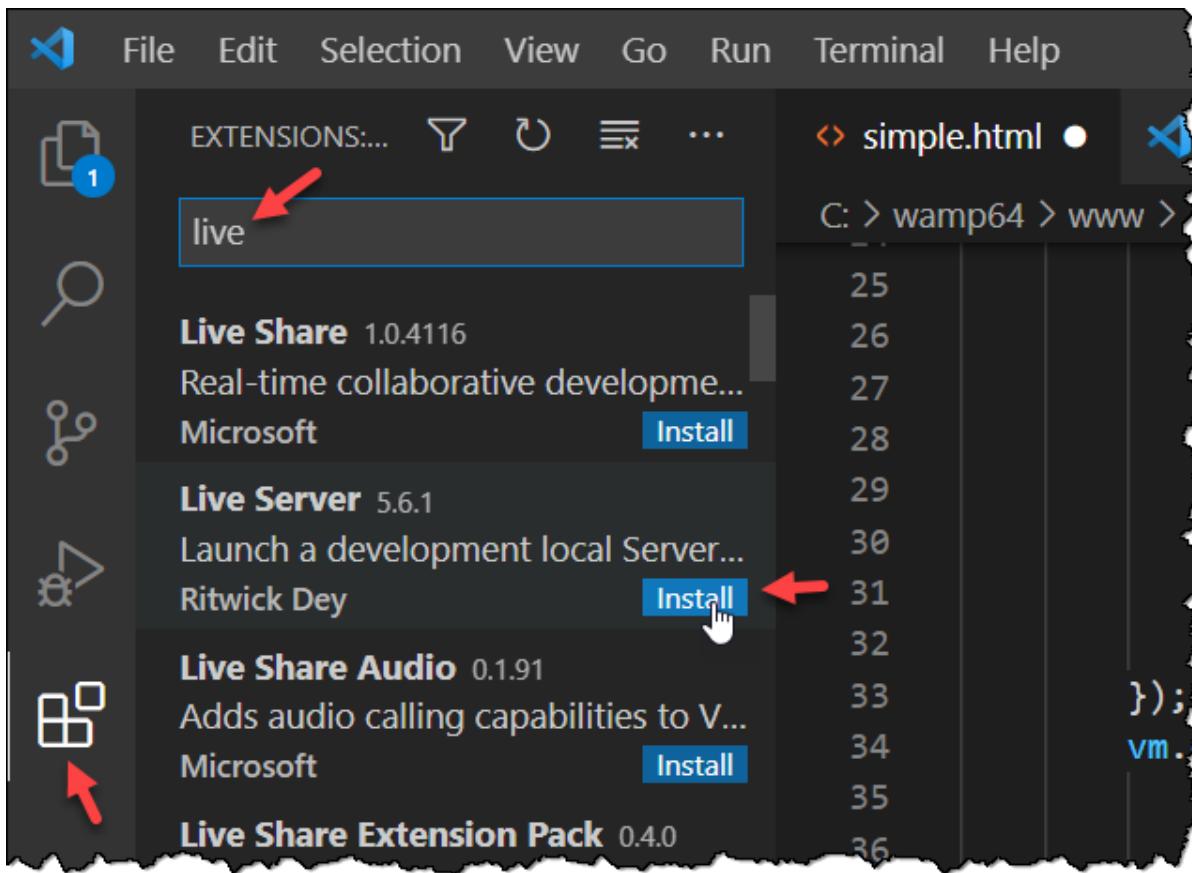
Remarquez la coloration syntaxique des balises, qui n'aurait pas lieu sans l'extension Vetur.

L'extension Live Server

L'extension **Live Server** est très utile. Elle exécute votre code dans un serveur local. Chaque fois que le code est modifié dans VS Code, l'affichage dans le navigateur est automatiquement mis à jour.

Voyons comment installer l'extension Live Server.

Cliquez sur l'icône **Extensions** dans la partie supérieure gauche de la fenêtre. Tapez *Live Server* et cliquez sur **Install** sous **Live Server** :

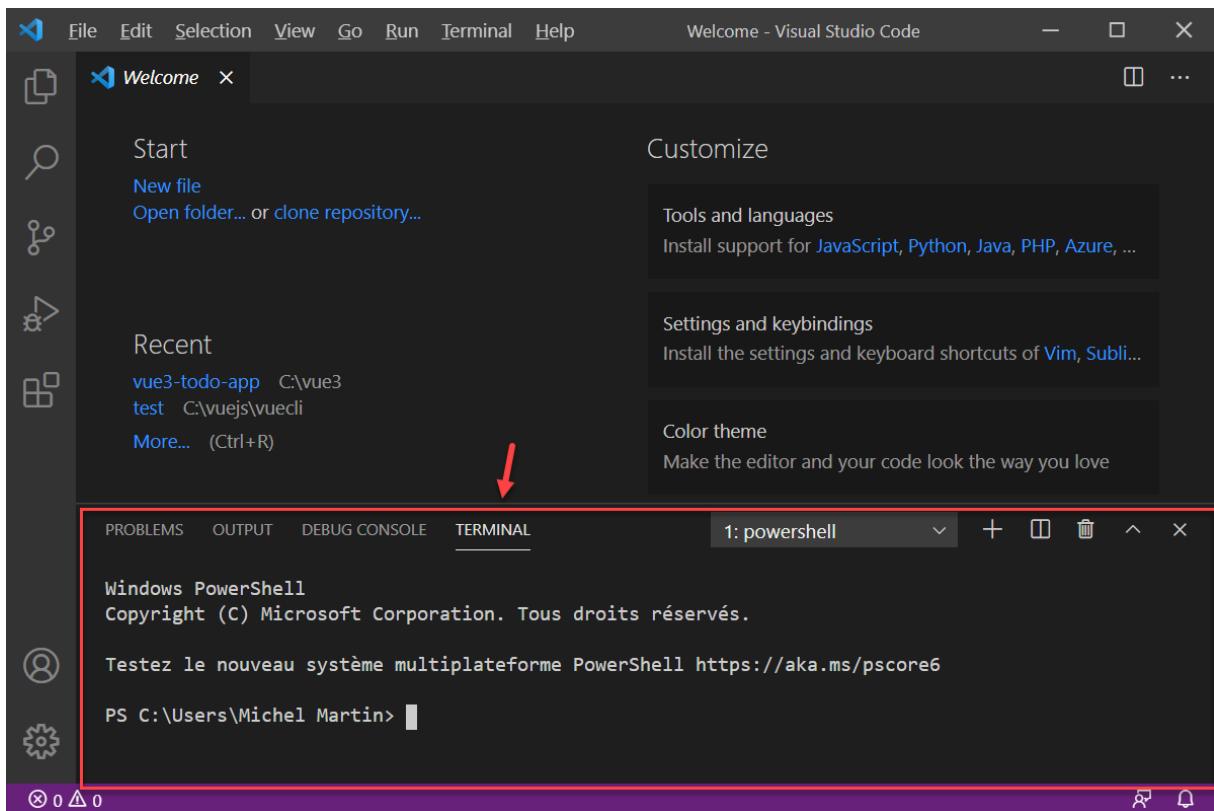


Pour ouvrir un fichier HTML en cours d'édition dans le serveur local Live Server, cliquez du bouton droit sur le code de ce fichier dans VSCode et cliquez sur **Open with Liver Server** dans le menu contextuel. Le code s'exécute dans votre navigateur par défaut. L'affichage sera mis à jour chaque fois que vous sauvegarderez votre code dans VS Code.

Une invite de commande dans VS Code

Lors de vos développements en Vue 3, vous devrez taper des commandes textuelles. Par exemple pour créer un nouveau projet avec Vue Cli. Pour cela, vous pouvez utiliser une fenêtre Invite de commandes sous Windows, une fenêtre Terminal sous OSX ou ... le terminal intégré dans VSCode.

Pour accéder au terminal intégré dans VS Code, lancez la commande **Terminal** dans le menu **View**. Un volet contenant plusieurs onglets s'affiche dans la partie inférieure de la fenêtre de VS Code. L'onglet **TERMINAL** est sélectionné dans cet onglet. C'est là que vous taperez les commandes :

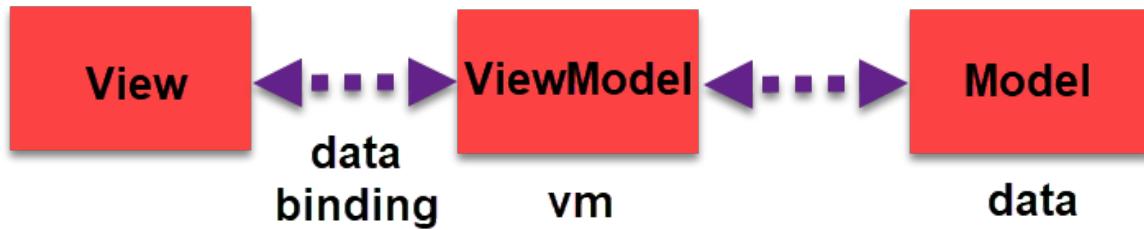


Le modèle MVVM de Vue.js

Vue.js utilise le modèle **MVVM** (*Model-View-ViewModel*). C'est un modèle assez proche du modèle **MVC** (*Model-View-Controller*). Le but est de séparer ce qui est affiché sur l'écran (l'interface utilisateur) de la logique programmatique :

- Le **model** représente les données nécessaires à l'application : nom, âge, numéro de sécurité sociale
- La **view** représente l'interface accessible à l'utilisateur. Si l'utilisateur agit sur la **view**, elle s'adapte. Mais elle ne contient pas la logique JavaScript nécessaire.
- La **viewmodel** est un intermédiaire bidirectionnel entre la vue et le modèle :
 - Les changements dans le model sont transmis à la **view** via la **ViewModel** par le processus de **data binding**

- Les actions dans la **view** sont transmises au **model**



Création du viewmodel

```
const app = Vue.createApp({
);
app.mount('#app');
```

Création du model

```
const app = Vue.createApp({
  data : function() {
    return {
    }
  }
});
```

Création de la view

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

Un premier projet Vue.js

Vous allez enfin écrire votre premier code Vue.js 2. Ici, vous utiliserez :

- Un simple fichier HTML pour définir le code HTML et JS.
- L'extension Emmet dans VS Code pour faciliter l'écriture du code HTML.
- Le CDN de Vue.js 3.

- Le serveur local Live Server pour exécuter le code.

Allons-y.

Je vous suggère de stocker tous vos premiers projets dans le dossier **vue3** qui se trouvera à la racine de votre disque principal **c :** (si vous travaillez sur un PC sous Windows 10 bien entendu).

Ouvrez VS Code. Lancez la commande **Terminal** dans le menu **View** pour afficher le terminal. Tapez ces commandes :

```
cd \
```

```
md vue3
```

Lancez la commande **Open folder** dans le menu **File** et désignez le dossier **c :\vue3** que vous venez de définir. Vous allez maintenant créer le fichier **projet1.html**. Dans le volet gauche, sous **VUE3**, cliquez sur l'icône **New File** et créez le fichier **projet1.html**.

Tapez *doc* ou *html :5* et appuyer sur la touche *Entrée* pour écrire un squelette standard avec l'extension Emmet intégré à VS Code :

```
index.html •
C: > vue3 > index.html
1 doc
  ↗ doc Emmet Abbreviation
```

Voici ce que vous devez obtenir :

```
File Edit Selection View Go Run Terminal Help • projet1.html - vue3 - Visual Studio Code
EXPLORER OPEN EDITORS 1 UNSAVED ...
Welcome projet1.html ...
VUE3 projet1.html ...
OUTLINE
Ln 5, Col 54 (12 selected) Spaces: 4 UTR-8 CRLF HTML ⚡ ⚡
```

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
  </head>
  <body>
    <h1>Hello Vue.js!</h1>
  </body>
</html>
```

Vous allez maintenant intégrer le CDN de Vue 3 dans l'en-tête du document HTML. Allez sur la page <https://v3.Vue.js.org/>, cliquez sur **GET**

STARTED, puis sur **Installation**, dans le volet gauche. Récupérez le CDN qui pointe vers la dernière version de Vue et collez-le dans la section **<head>** **</head>** de votre document :

CDN

For prototyping or learning purposes, you can use the latest version with:

1

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

html



For production, we recommend linking to a specific version number and build to avoid unexpected breakage from newer versions.

Avec ce code, vous aurez systématiquement accès à la dernière version de Vue.js. Si vous voulez utiliser une version spécifique de Vue, par ex la **3.0.2**, utilisez ce code :

```
<script src="https://unpkg.com/vue@3.0.2"></script>
```

Vous allez maintenant créer une balise **<div></div>** d'id **app** dans la section **<body></body>** du document pour héberger l'application. Tapez :

```
div#app
```

Et appuyez sur la touche *Entrée* du clavier. Emmet remplace ce raccourci par une balise div d'id app :

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

Tout ce qui se trouve à l'intérieur de cette balise **<div></div>** sera contrôlé par Vue.js.

Vous allez maintenant insérer la section **<script></script>** à la suite de la balise **<div></div>**.

Pour créer une application Vue 3, deux instructions sont nécessaires :

- Une première qui crée un objet **app** en exécutant la méthode **createApp()** de la classe Vue.
- Une deuxième qui exécute la fonction **mount()** sur l'objet créé par la première instruction et qui précise **l'id** de l'élément qui doit être contrôlé par Vue.js.

```
<body>
```

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

```
<script>  
    const app = Vue.createApp();  
    app.mount('#app');  
</script>  
</body>
```

Vous allez maintenant compléter ce code en fournissant un objet en argument de la méthode **createApp()**. Cet objet contiendra au minimum un template HTML. Par exemple, un titre **<h2></h2>** :

```
const app = Vue.createApp({  
    template : '<h2>Ma première application Vue 3</h2>'  
});
```

Remarque

Contrairement à Vue 2, Vue 3 permet d'avoir plusieurs éléments à la racine d'un template.

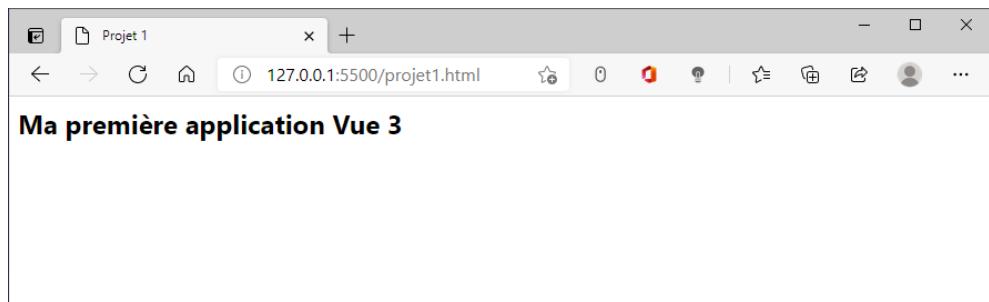
Vous pourriez très bien avoir quelque chose comme ceci :

```
template : `<div>  
    <p>Hello</p>  
</div>  
<div>  
    <p>Hello Again</p>  
</div>`
```

Remarquez les apostrophes penchées qui permettent de répartir le template sur plusieurs lignes pour améliorer sa lisibilité. Pour insérer une apostrophe penchée sur un PC, appuyez simultanément sur les touches *Alt Gr* et *7*, puis sur *Espace*.

Si nécessaire, appuyez sur *Alt + Maj + F* pour ajuster l'indentation du code.

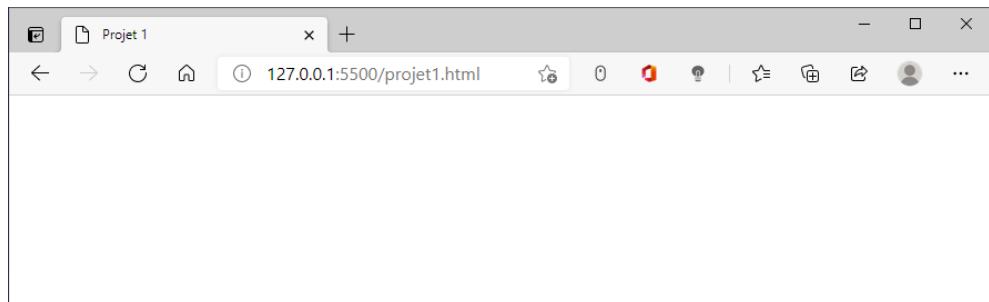
Sauvegardez ce code avec la commande **Save** dans le menu **File** ou le raccourci clavier *Contrôle + S*. Exécutez ce code dans le serveur local Live Server. Pour cela, cliquez avec le bouton droit de la souris dans le code et choisissez **Open With Live Server** dans le menu contextuel. Le code s'exécute dans le navigateur par défaut :



Ajoutez un paragraphe dans le template :

```
template : '<h2>Ma première application Vue 3</h2>
<p>... en 3 minutes chrono !</p>'
```

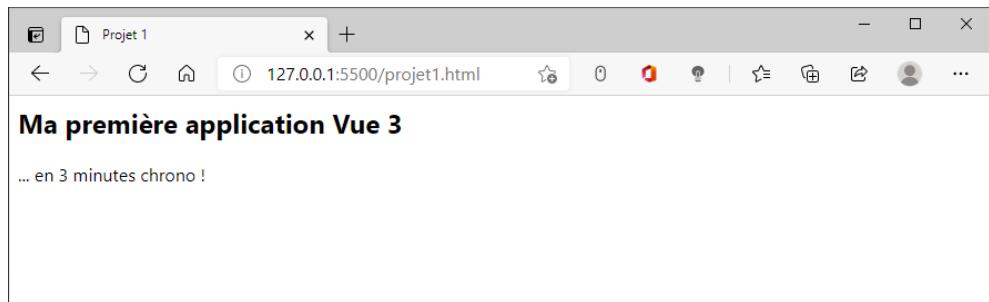
Sauvegardez le code et basculez sur le navigateur. Comme vous le voyez, plus rien n'est affiché dans le navigateur :



Pour corriger ce problème, ajoutez un antislash à la fin de la première ligne du template :

```
template : '<h2>Ma première application Vue 3</h2> \
<p>... en 3 minutes chrono !</p>'
```

Ça y est, le code est fonctionnel :



Retenez que lorsqu'un template est défini sur plusieurs lignes, vous devez ajouter un "\ " à la fin de toutes les lignes, sauf de la dernière.

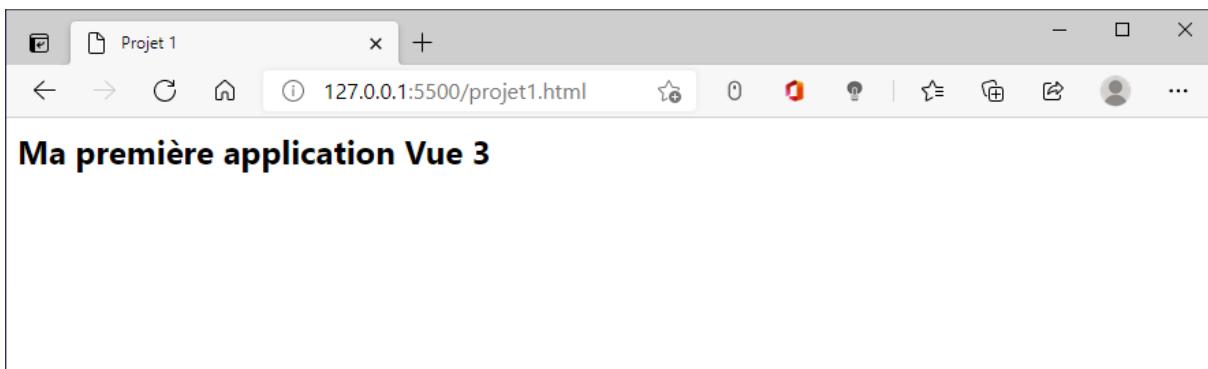
Si vous préférez, vous pouvez également encadrer les instructions du template par des apostrophes penchées.

Le template peut être défini dans la vue et pas dans le script.

Pour cela, on définit la fonction anonyme **data()** dans le script. Cette fonction retourne un objet. Les propriétés définies dans cet objet seront accessibles dans le template défini dans le HTML avec des interpolations, c'est-à-dire avec des accolades doubles.

```
<div id="app">  
  <h2>{{titre}}</h2>  
</div>  
  
<script>  
  const app = Vue.createApp({  
    data : function() {  
      return {  
        titre : 'Ma première application Vue 3'  
      }  
    }  
  }) ;  
  app.mount('#app')  
</script>
```

Voici le résultat :



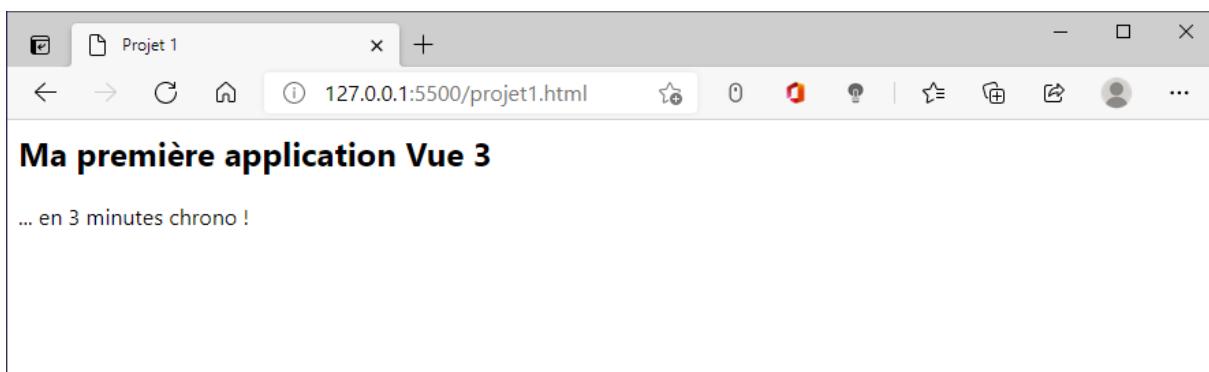
La définition de la fonction anonyme `data` peut être simplifiée en utilisant une syntaxe ES6 :

```
data() {  
    titre : 'Ma première application Vue 3'  
}
```

Pour ajouter le sous-titre, il suffit de définir une deuxième propriété dans le script et de l'afficher avec une interpolation dans le HTML :

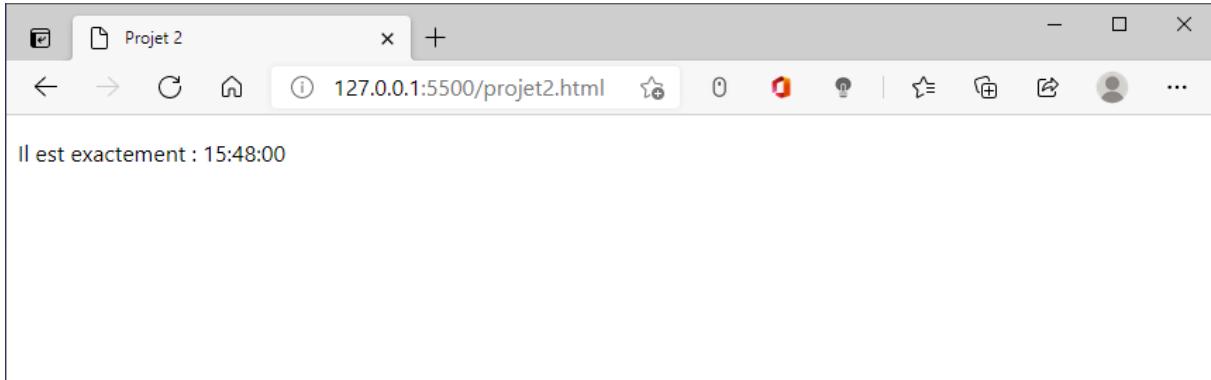
```
<div id="app">  
    <h2>{{titre}}</h2>  
    <p>{{sousTitre}}</p>  
</div>  
<script>  
    const app = Vue.createApp({  
        data() {  
            return {  
                titre : 'Ma première application Vue 3',  
                sousTitre : '... en 3 minutes chrono !'  
            }  
        }  
    }) ;  
    app.mount('#app')  
</script>
```

Voici le résultat (projet1) :



Challenge

Voyons si vous avez compris. Définissez une nouvelle application Vue.js qui affiche l'heure système :



Solution (projet2)

Définissez une nouvelle application.

Insérez-y un squelette standard HTML 5 avec l'extension Emmet de VS Code.

Faites référence au CDN de Vue 3.

Définissez un div d'id **app** pour représenter la vue.

Définissez l'application **app** avec la méthode **createApp()** de la classe Vue et reliez l'objet créé à la balise d'id **app** avec la méthode **mount()**.

Définissez la fonction anonyme **data()**. Cette fonction retourne l'heure locale via la fonction **toLocaleTimeString()** d'un objet **Date**.

L'heure est affichée avec une interpolation dans la vue :

```
<body>
  <div id="app">
    <p>Il est exactement : {{date}}</p>
  </div>
<script>
  const app = Vue.createApp({
    data() {
      return {
        date : new Date().toLocaleTimeString()
      }
    }
  })
  app.mount('#app')
</script>
```

```
        }
    }
}) ;
app.mount('#app') ;
</script>
</body>
```

Dans cette section, vous avez appris plusieurs choses :

- 1)** À utiliser l'extension Emmet dans VS Code pour faciliter l'écriture du code HTML.
- 2)** À créer une balise HTML contrôlée par Vue.js
- 3)** À définir la partie JavaScript de l'application avec les méthodes **createApp()** et **mount()**.
- 4)** À créer des données dans la partie JavaScript de l'application et à les utiliser dans la vue avec des interpolations.

Un deuxième exemple (projet3)

Dans cette section, vous allez voir que les interpolations peuvent contenir des expressions JavaScript.

Vous avez vu qu'une interpolation peut faire référence aux données définies dans le modèle de l'application. Eh bien, n'importe quelle expression JavaScript valide peut être utilisée dans une interpolation. Nous allons dupliquer le code du projet 1 et le renommer en projet 3.

Ici par exemple, je vais :

- 1)** Inverser le titre. C'est-à-dire afficher le caractères de droite à gauche et non de gauche à droite.
- 2)** Afficher le sous-titre en majuscules.

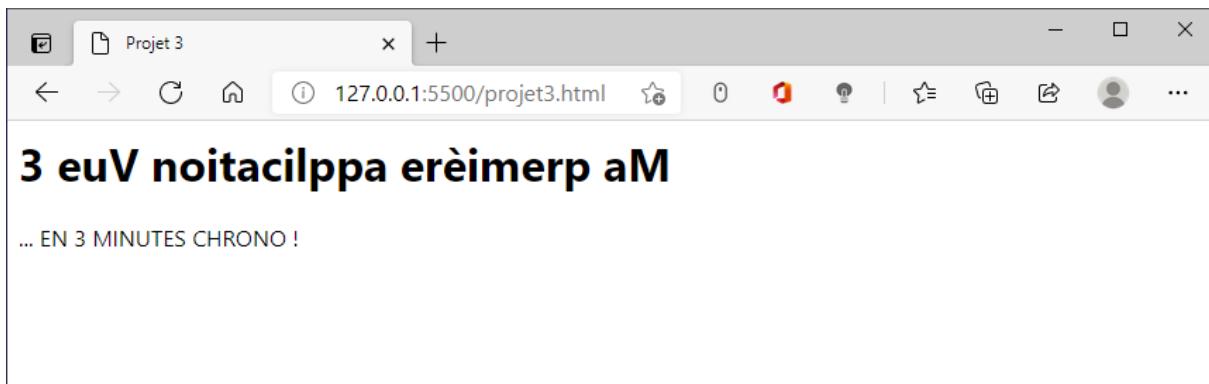
Pour inverser le titre, j'applique la fonction **split()** à la variable **titre**. En lui passant une chaîne vide, les caractères de la chaîne sont placés dans un tableau à raison d'un caractère par cellule. J'applique ensuite la fonction **reverse()** pour inverser le tableau et la fonction **join()** avec une chaîne vide

en paramètre pour transformer le tableau en une chaîne sans aucun séparateur entre les caractères.

Pour afficher le sous-titre en majuscules, j'applique la fonction **toUpperCase()** à la variable **sousTitre** :

```
<h1>{{titre.split('').reverse().join('')}}</h1>
<p>{{sousTitre.toUpperCase()}}</p>
```

Voici le résultat :



Challenge

Supposons qu'une propriété **inter** soit initialisée à **true** ou à **false** dans le modèle. Comment afficheriez-vous **Vrai** ou **Faux** à l'aide d'une interpolation ?

Solution (projet4) :

Avec une expression ternaire :

```
<body>
  <div id="app">
    <p>{{inter ? 'Vrai' : 'Faux'}}</p>
  </div>
  <script>
    const app = Vue.createApp({
      data() {
        return {

```

```

        inter : true
    }
}

}) ;

app.mount('#app');

</script>

</body>

```

Propriétés calculées

Dans les sections précédentes, vous avez vu qu'il était possible d'utiliser du code JavaScript dans une interpolation. Si le code JavaScript est long et/ou complexe, il est préférable de le placer dans le ViewModel en utilisant des propriétés calculées.

Les propriétés calculées sont définies dans la propriété **computed** de l'objet passé en paramètre à la méthode **createApp()** :

```

const app = Vue.createApp({
  data() {
    return {
      d : 15
    }
  },
  computed : {
    calcul() {
      return uneProprieteCalculee
    }
  }
}) ;
app.mount('#app');

```

Pour interpoler une propriété calculée, écrivez ceci :

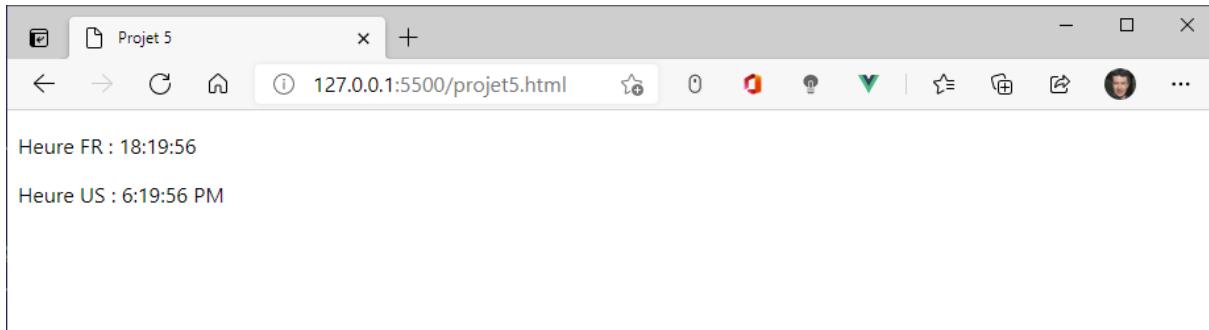
```
{ {calcul} }
```

Remarque

Les données définies dans le modèle sont accessibles dans les propriétés calculées en les préfixant par "this". Ici par exemple, **d** est accessible via **this.d**.

Challenge

Définissez deux interpolations pour afficher l'heure courante au format français et américain.



Solution :

```
<body>
    <div id="app">
        <p>Heure FR : {{dateFr}}</p>
        <p>Heure US : {{dateUs}}</p>
    </div>
    <script>
        const app = Vue.createApp({
            data() {
                return {
                    d : new Date()
                }
            },
            computed : {
                dateFr() {
                    return this.d.toLocaleTimeString();
                },
                dateUs() {
                    return this.d.toLocaleTimeString('en-US');
                }
            }
        })
    </script>

```

```

        }
    }) ;
    app.mount('#app') ;
</script>
</body>

```

Méthodes ou propriétés calculées ?

La version avec methods :

```

<body>

<div id="app">

    <p>Heure FR : {{dateFr}}</p>
    <p>Heure US : {{dateUs}}</p>

</div>

<script>

    const app = Vue.createApp({


        data() {
            return {
                dateFr : new Date().toLocaleTimeString(),
                dateUs : new Date().toLocaleTimeString('en-US')
            }
        }
    });

    app.mount('#app') ;

</script>

```

La version avec computed :

```

<body>

<div id="app">

    <p>Heure FR : {{dateFr}}</p>

```

```

<p>Heure US : {{dateUs}}</p>
</div>
<script>
  const app = Vue.createApp({
    data() {
      return {
        d : new Date()
      }
    },
    computed : {
      dateFr() {
        return this.d.toLocaleTimeString();
      },
      dateUs() {
        return this.d.toLocaleTimeString('en-US');
      }
    }
  });

  app.mount('#app');
</script>
</body>

```

Ces deux codes produisent le même effet.

Cependant, le code d'une fonction calculée n'est réévalué que si ses dépendances ont changé.

Il est donc moins consommateur que le même code défini dans la propriété **methods**.

Data binding

Vous savez insérer une propriété du modèle dans l'innerHTML d'une balise en utilisant une interpolation. Ce cours va vous montrer comment affecter à un attribut HTML la valeur contenue dans une propriété du modèle. Cette technique s'appelle le *data binding*.

Pour binder une propriété du modèle à un attribut de la vue, il suffit de préfixer cet attribut par "**v-bind** :" , ou en version abrégée par " ":".

Voyons un exemple de data binding.

Nous allons afficher le logo de Vue.js en utilisant un data-binding sur la propriété src d'une balise img. Le logo se trouve à l'adresse '<https://Vue.js.org/images/logo.png>'.

Data binding in extenso (projet6) :

```
<body>
    <div id="app">
        
    </div>
    <script>
        const app = Vue.createApp({
            data() {
                return {
                    url : 'https://Vue.js.org/images/logo.png'
                }
            }
        });
        app.mount('#app');
    </script>
</body>
```

Data binding abrégé

```
<body>
    <div id="app">
        
    </div>
    <script>
        const app = Vue.createApp({
            data() {
```

```

        return {

            url : 'https ://Vue.js.org/images/logo.png'

        }

    }

);

app.mount('#app');

</script>

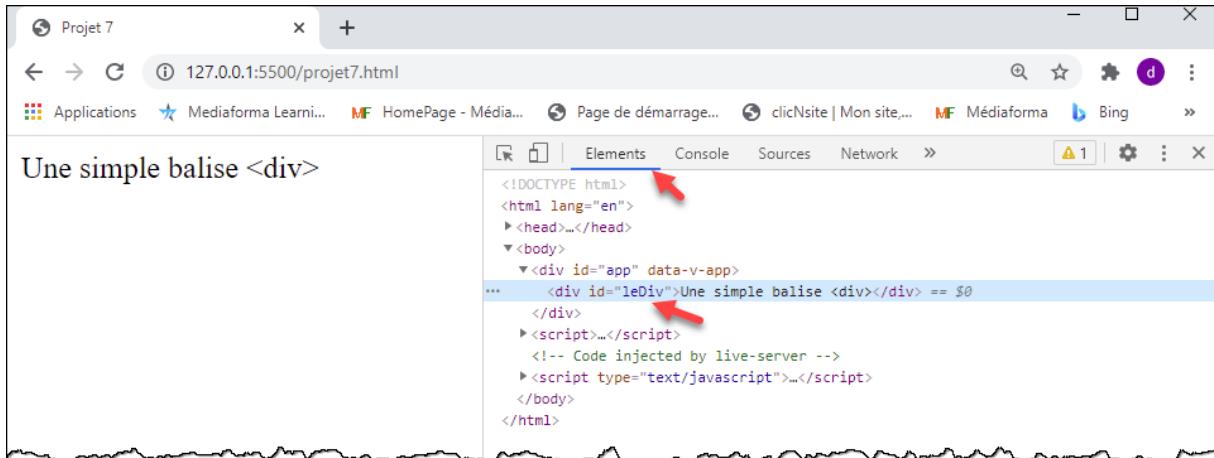
</body>

```

Challenge (projet 7)

Affectez un id dynamique à un **<div></div>** en utilisant un data-binding (l'id sera défini dans les **data** du modèle).

Vérifiez dans la console que l'id est bien celui que vous avez défini dans les **data**.



Solution :

```

<body>

<div id="app">

    <div v-bind :id="idDuDiv">Une simple balise &lt;div&gt;
</div>

</div>

<script>

    const app = Vue.createApp({

        data() {

```

```

        return {
            idDuDiv : 'leDiv'
        }
    });
app.mount('#app');
</script>
</body>
```

innerHTML et nœud texte

Vous savez insérer une propriété du modèle dans l'innerHTML d'une balise en utilisant une interpolation. Cette propriété doit être au format texte brut. Si vous essayez d'afficher un texte HTML avec une interpolation, vous verrez les balises HTML et non leur interprétation.

Il existe une possibilité complémentaire pour afficher les propriétés du modèle dans la vue : vous pouvez utiliser les directives **v-text** et **v-html** pour affecter une donnée (respectivement) texte ou HTML au innerHTML d'un élément :

```

<body>
    <div id="app">
        <p v-text="texteBrut"></p>
        <p v-html="texteHTML"></p>
    </div>
    <script>
        const app = Vue.createApp({
            data() {
                return {
                    texteBrut : 'Un simple texte',
                    texteHTML : '<font color="red"><u>Un texte
rouge souligné</u></font>'
                }
            }
        });
        app.mount('#app');
```

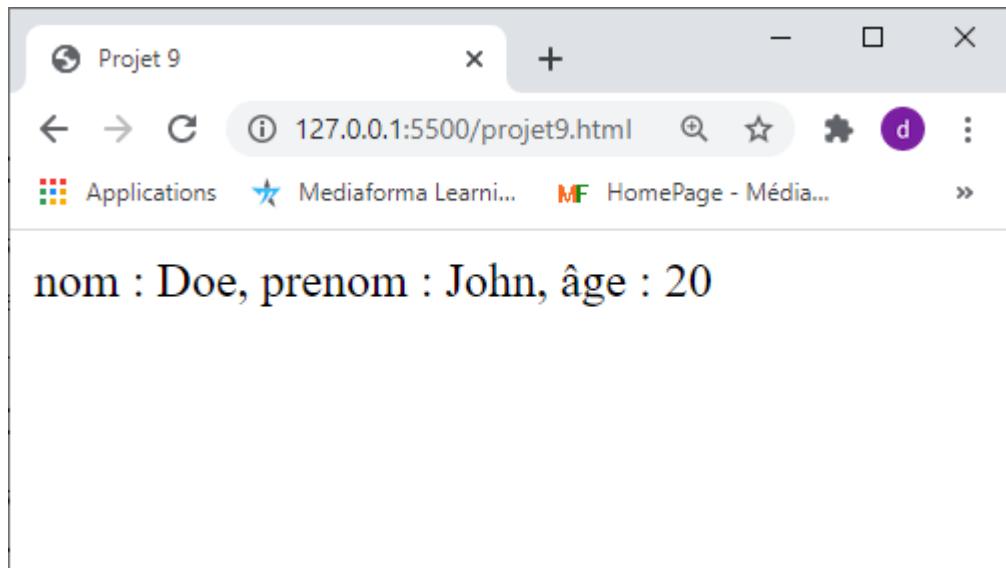
```
</script>  
</body>
```

Voici le rendu :



Challenge

Définissez l'objet JSON **personne** dans le modèle. Définissez le nom, le prénom et l'âge d'une personne. Utilisez des directives **v-text** ou **v-html** pour obtenir ceci :



Solution (projet 9) :

```
<body>
    <div id="app">
        nom : <span v-text="personne.nom"></span>,
        prenom : <span v-text="personne.prenom"></span>,
        âge : <span v-text="personne.age"></span>
    </div>
    <script>
        const app = Vue.createApp({
            data() {
                return {
                    personne : {
                        nom : 'John',
                        prenom : 'Doe',
                        age : 20
                    }
                }
            }
        });
    </script>
```

```
    app.mount('#app');
</script>
</body>
```

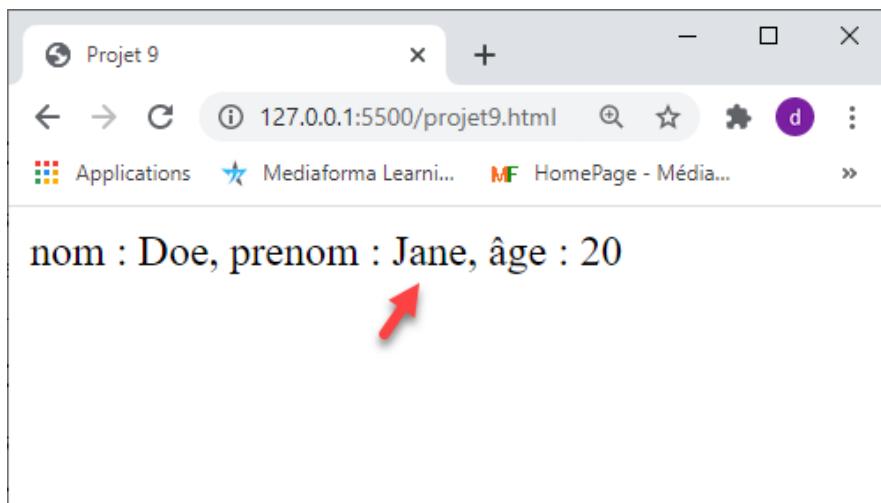
La méthode **mount()** retourne l'instance de la vue. En stockant cette instance dans une variable, vous pouvez accéder aux propriétés définies dans le modèle.

L'instance de la vue est stockée dans la variable **vm** (pour **ViewModel**). Cette variable donne accès aux propriétés définies dans le modèle. Ici par exemple, nous modifions le prénom de la propriété personne :

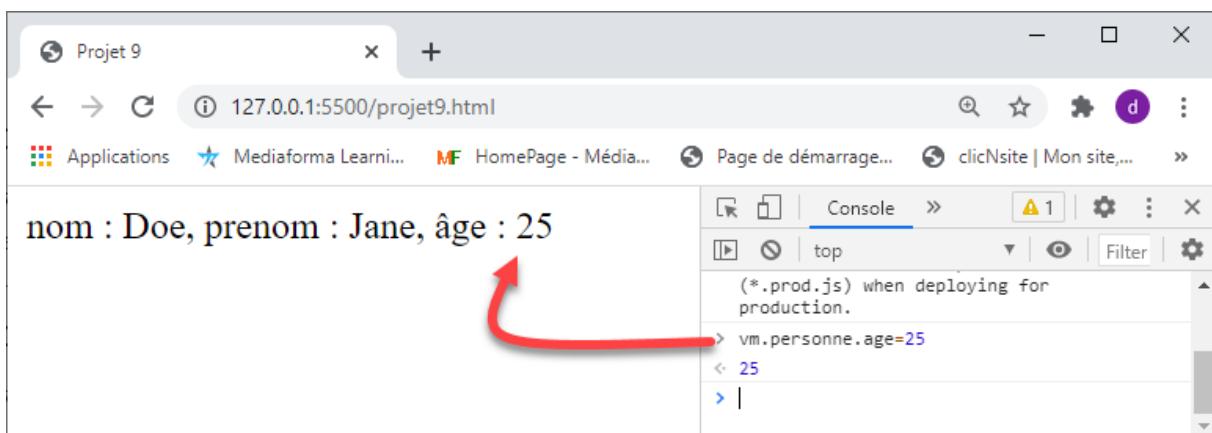
```
const vm = app.mount('#app');

vm.personne.prenom='Jane';
```

Voici le résultat :



Les propriétés du modèle peuvent également être modifiées dans la console du navigateur :



Challenge

Vous avez vu que si le modèle change, la vue change également.

Appelez la fonction **setTimeout()** après la définition de la vue et modifiez l'âge de l'objet personne en lui affectant la valeur **25** au bout de **3 secondes**.

Solution (projet 10) :

```
let vm = app.mount('#app');
setTimeout(()=>{
    vm.personne.age = 25;
}, 3000);
```



Puis, trois secondes plus tard :

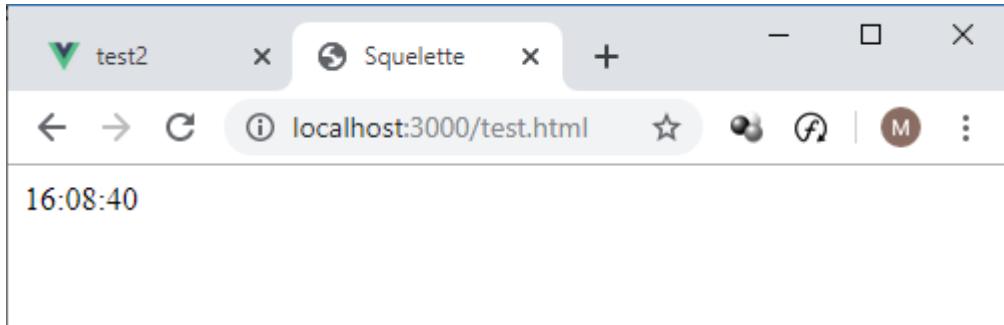


Challenge

Vous savez agir sur les propriétés du **model** en passant par le **viewModel** (**vm**).

Vous savez également afficher des interpolations dans la vue avec des accolades doubles.

Définissez le code nécessaire pour afficher une horloge au format **hh :mm :ss** en Vue.js.



Solution (projet11) :

```
<body>
    <div id="app">
        <span>{ { hms } }</span>
    </div>
    <script>
        const app = Vue.createApp({
            data() {
                return {
                    hms : ''
                }
            }
        });
        let vm = app.mount('#app');
        setInterval(() => {
            let d = new Date();
            vm.hms = d.toLocaleTimeString();
        }, 1000);
    </script>
</body>
```

Binding bidirectionnel

Vous savez binder une propriété du modèle avec un attribut HTML en ajoutant la directive "**v-bind** :" ou l'alias " :" devant cet attribut. On appelle cela le *one-way binding*.

Vous allez maintenant apprendre à effectuer un binding bidirectionnel (*two-way binding*). Ceci permettra au DOM d'injecter des données dans Vue.

Pour cela, vous utiliserez l'attribut **v-model** dans une balise HTML de formulaire (**<input>**, **<select>** ou **<textarea>**). Les autres balises HTML ne sont pas concernées par **v-model**.

Par exemple dans une balise **<input>** de type **text** :

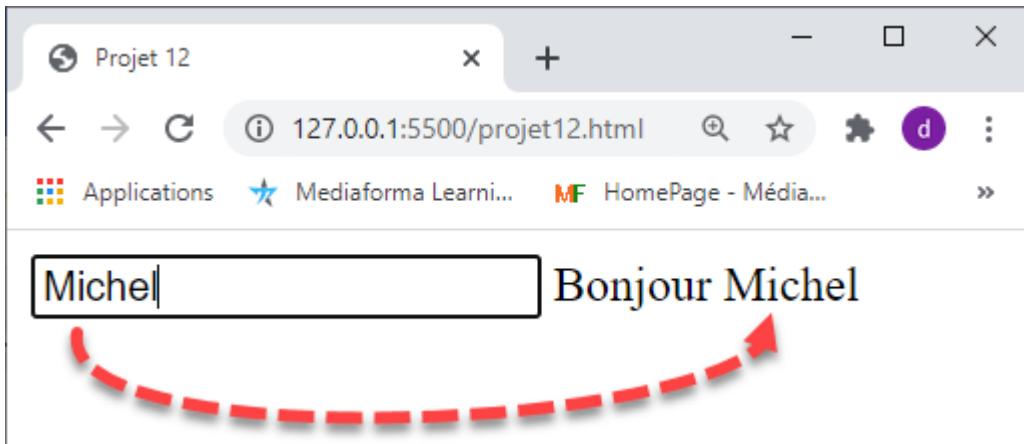
```
<input type="text" placeholder="Nom" v-model="nom"> {{ nom }}
```

Pour récupérer le nom dans les data de Vue, écrivez simplement :

```
data() {  
  return {  
    nom : ''  
  }  
}
```

Challenge

Ecrivez le code nécessaire pour obtenir ce comportement :



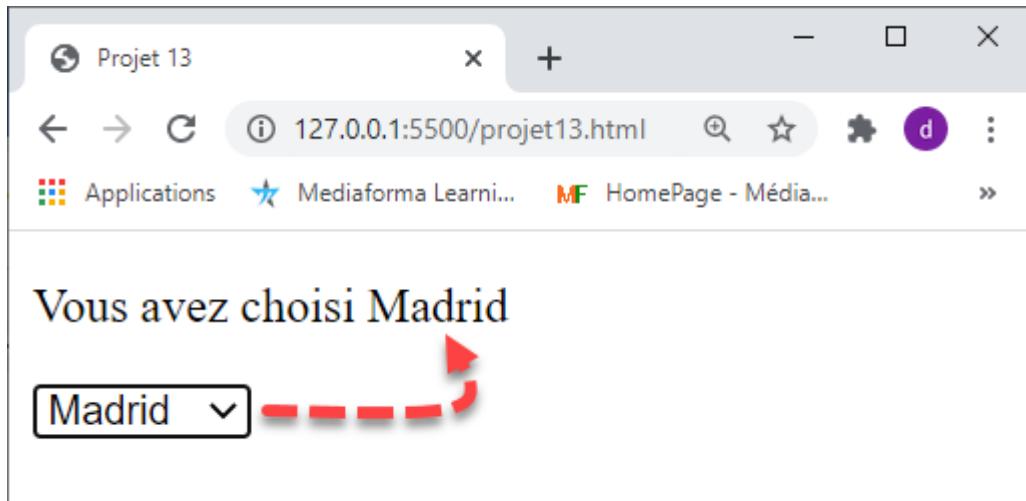
Solution (projet12) :

```
<body>
  <div id="app">
    <input type="text" placeholder="Entrez votre nom" v-
model="nom"> Bonjour {{nom}}
  </div>
  <script>
    const app = Vue.createApp({
      data() {
        return {
          nom : ''
        }
      }
    });
    let vm = app.mount('#app');
  </script>
</body>
```

Challenge

Définissez une liste déroulante qui contient les trois capitales Paris, Londres et Madrid.

Au-dessus de la liste déroulante, affichez le message "Vous avez choisi " suivi du nom de la ville sélectionnée.



Solution (projet13) :

```
<body>
  <div id="app">
    <p>Vous avez choisi {{selection}}</p>
    <select v-model="selection">
      <option>Paris</option>
      <option>Londres</option>
      <option>Madrid</option>
    </select>
  </div>
  <script>
    const app = Vue.createApp({
      data() {
        return {
          selection : ''
        }
      }
    });
    let vm = app.mount('#app');
  </script>
```

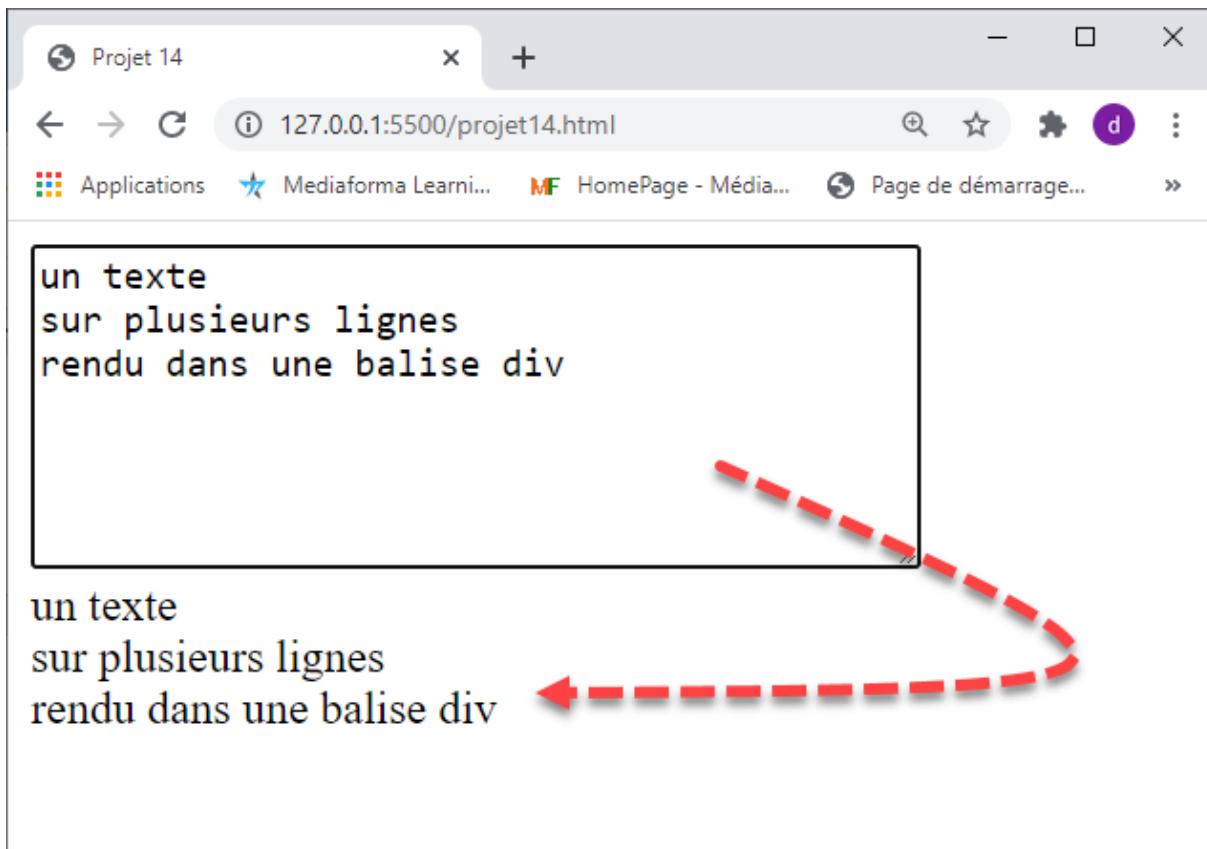
```
</body>
```

Si vous utilisez un attribut **v-model** dans un **<textarea>**, les données saisies seront rendues sous la forme d'une ligne unique, même si elle sont tapées sur plusieurs lignes dans le **<textarea>**.

Pour les afficher sur plusieurs lignes, insérez l'interpolation dans un conteneur dont la propriété CSS **white-space** vaut **pre-line**.

Challenge

Affichez ce qui est saisi dans une balise **<textarea>** dans une balise **<div>**, en conservant le multiligne.



Solution (projet14) :

```
<body>
  <div id="app">
    <textarea v-model="multi" cols="40" rows="7"></textarea>
    <div style="white-space : pre-line;">{{ multi }}</div>
  </div>
```

```

<script>
    const app = Vue.createApp({
        data() {
            return {
                multi : ''
            }
        }
    });
    let vm = app.mount('#app');
</script>
</body>

```

Utilisation conjointe des directives v-model et v-bind

Jusqu'ici, vous avez utilisé :

- Soit la directive **v-bind** dans une balise HTML pour la connecter à une propriété du modèle.
- Soit la directive **v-model** dans une balise HTML de formulaire pour relier son contenu à une propriété du modèle.

Vous pouvez utiliser les directives **v-model** et **v-bind** conjointement dans une même application.

Pour illustrer ce fonctionnement, je vais définir un document HTML qui contient deux **<input>** de type **text**. A chaque frappe d'une touche du clavier, le contenu du premier input sera recopié en majuscules dans le second.

Pour la première fois, il va être nécessaire de réagir à un évènement généré par l'utilisateur. Ici, la frappe sur une touche du clavier. Pour cela, je vais définir la fonction évènementielle **recopie()**. La directive **v-on :keyup** reliera les frappes au clavier à cette fonction :

```
<input v-on :keyup="recopie">
```

La fonction **recopie()** sera définie dans la propriété **methods** du model :

```
methods : {
```

```
    recopie : function() {  
    }  
}
```

Solution (projet15)

```
<div id="app">  
    <input type="text" v-on :keyup="recopie" v-model="texte1">  
    <input type="text" v-bind :value="texte2">  
</div>  
<script>  
    const app = Vue.createApp({  
        data() {  
            return {  
                texte1 : '',  
                texte2 : ''  
            }  
        },  
        methods : {  
            recopie() {  
                this.texte2 = this.texte1.toUpperCase();  
            }  
        }  
    });  
    let vm = app.mount('#app');  
</script>
```

Challenge

Vous allez mettre en pratique l'utilisation conjointe des directives **v-bind** et **v-model** sur un court challenge.

Ecrivez le code nécessaire pour obtenir ce comportement :

Cochez les cases des langages que vous connaissez :

HTML CSS JavaScript

Cases cochées : ["HTML", "CSS", "JavaScript"]

Voici un indice pour vous aider.

En affectant à plusieurs cases à cocher :

- La même directive **v-model**
- des **value** différentes

Et en définissant dans le modèle la propriété correspondante au **v-model** sous la forme d'un tableau, vous pouvez facilement mémoriser l'état des cases à cocher.

Solution (projet16) :

```
<div id="app">
    <p>Cochez les cases des langages que vous connaissez :</p>
    <input type="checkbox" v-model="cases" value="HTML">
<label>HTML</label>
    <input type="checkbox" v-model="cases" value="CSS">
<label>CSS</label>
    <input type="checkbox" v-model="cases" value="JavaScript">
<label>JavaScript</label>
    <p>Cases cochées : {{cases}}</p>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                cases : []
            }
        }
    })
    app.mount('#app')
</script>
```

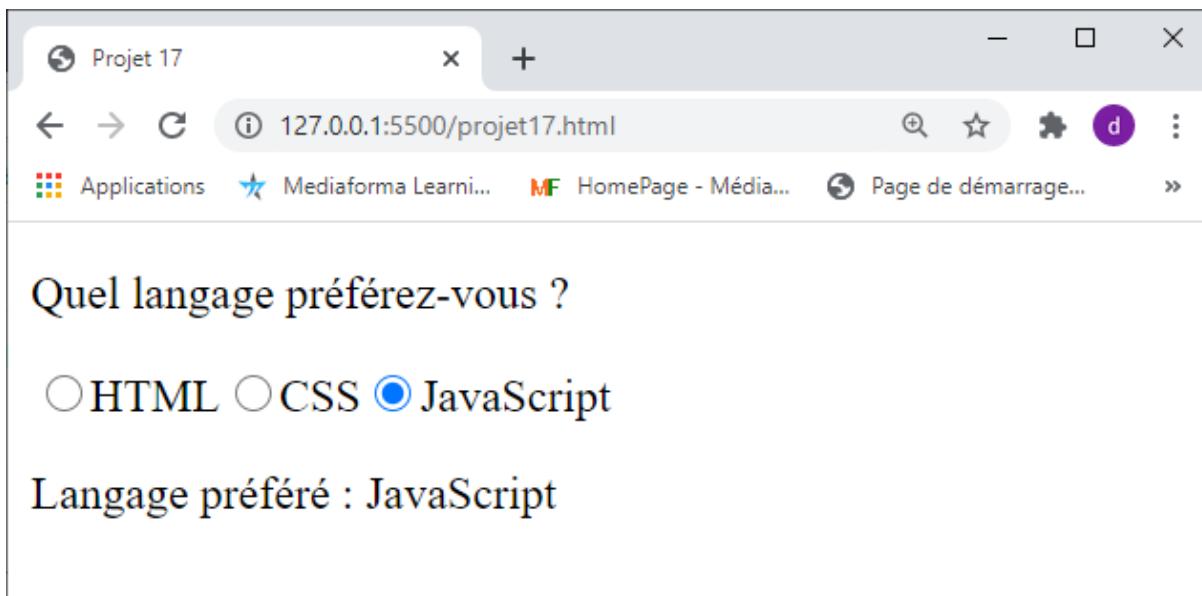
```
        }
    }
}) ;
let vm = app.mount('#app') ;
</script>
```

Challenge

Je vous propose un deuxième challenge pour vous entraîner avec les boutons radio.

Le principe est le même pour les <input> de type **radio**.

Ecrivez le code nécessaire pour obtenir ce résultat :



Solution (projet17) :

```
<div id="app">
    <p>Quel langage préférez-vous ?</p>
    <input type="radio" v-model="lang" value="HTML">
    <label>HTML</label>
    <input type="radio" v-model="lang" value="CSS">
    <label>CSS</label>
    <input type="radio" v-model="lang" value="JavaScript">
    <label>JavaScript</label>
    <p>Langage préféré : {{lang}}</p>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                lang : ''
            }
        }
    })
    app.mount('#app')
</script>
```

```
        }
    });
let vm = app.mount('#app');
</script>
```

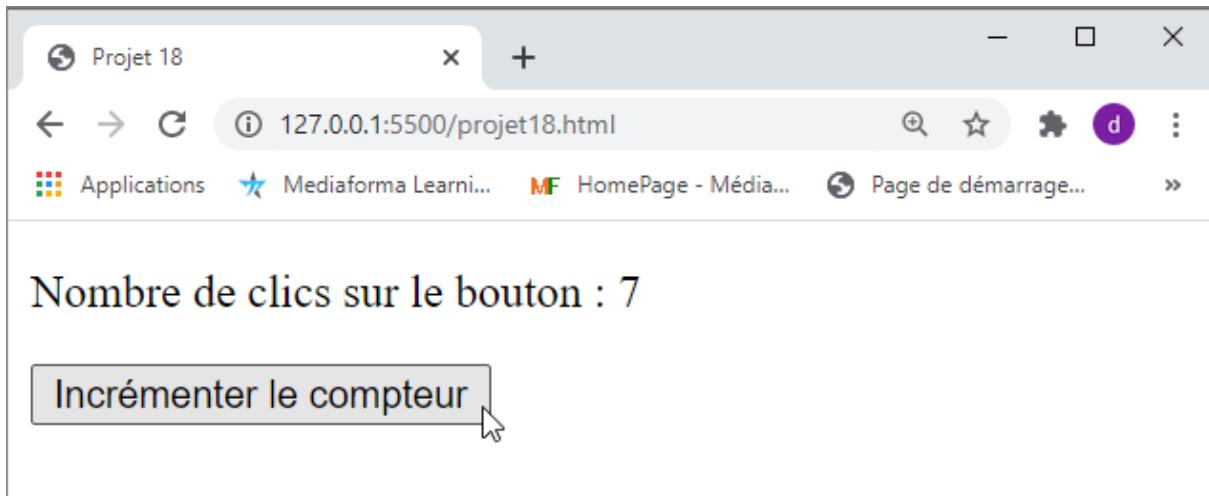
Gestion évènementielle

Vous avez déjà fait connaissance avec la gestion évènementielle dans Vue.js. Rappelez-vous, vous avez utilisé une directive **v-on :keyup** pour réagir aux frappes sur le clavier. Vous pouvez utiliser la directive **v-on :evenement** ou son alias **@evenement** pour gérer tous les évènements JavaScript. Ici, **evenement** représente l'évènement ciblé : **click** ou **mousemove** par exemple.

Vous pouvez affecter à cette directive un code JavaScript ou le nom de la fonction à exécuter sans les parenthèses et sans point-virgule.

Challenge

Définissez le code nécessaire pour incrémenter une propriété du modèle au clic sur un bouton et affichez le nombre de clics. Ici, vous utiliserez une simple instruction JavaScript pour incrémenter le compteur à chaque clic sur le bouton :



Solution (projet18) :

```

<div id="app">
  <p>Nombre de clics sur le bouton : {{compteur}}</p>
  <button @click="compteur++">Incrémenter le compteur</button>
</div>
<script>
  const app = Vue.createApp({
    data() {
      return {
        compteur : 0
      }
    }
  );
  let vm = app.mount('#app');
</script>

```

Challenge

Pour définir la fonction de traitement dans le ViewModel, ajoutez la propriété **methods** dans le ViewModel. Définissez la fonction à exécuter dans cette propriété.

```

const app = Vue.createApp({
  data : {
    ...
  },
  methods : {
    traitement : function() {
      // Une ou plusieurs instructions
    }
  }
});

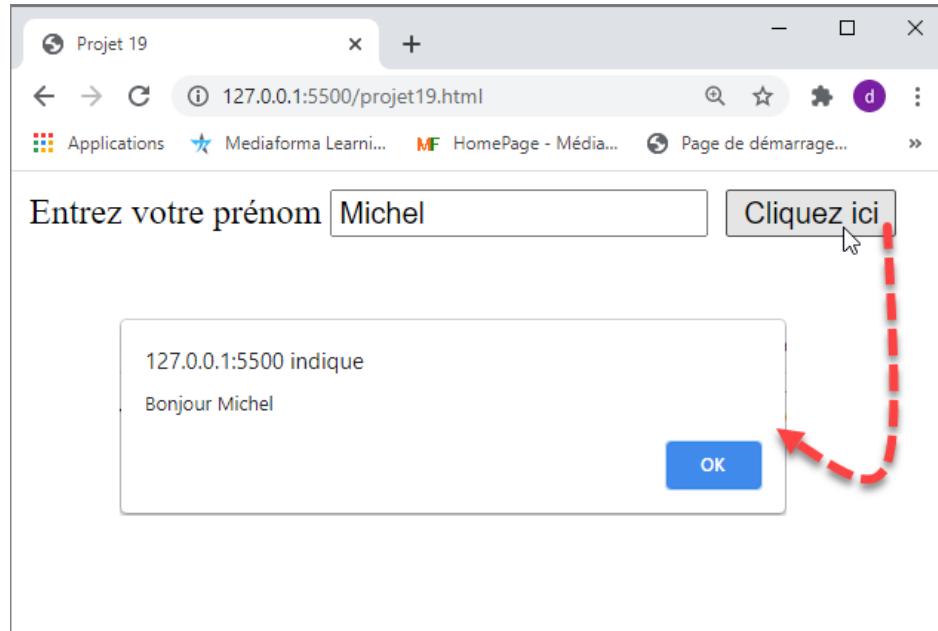
```

Challenge

Je vous propose un petit exercice.

Définissez un formulaire qui contient une zone de texte et un bouton.

Lorsque l'utilisateur clique sur le bouton, affichez une boîte d'alerte qui dit "bonjour" suivi du nom saisi dans la zone de texte.



Solution (projet19) :

```
<div id="app">
    <label>Entrez votre prénom </label><input type="text" v-
model="prenom" />
    <button @click="bonjour">Cliquez ici</button>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                prenom : ''
            }
        },
        methods : {
            bonjour() {
                alert('Bonjour ' + this.prenom);
            }
        }
    });
    let vm = app.mount('#app');
```

```
</script>
```

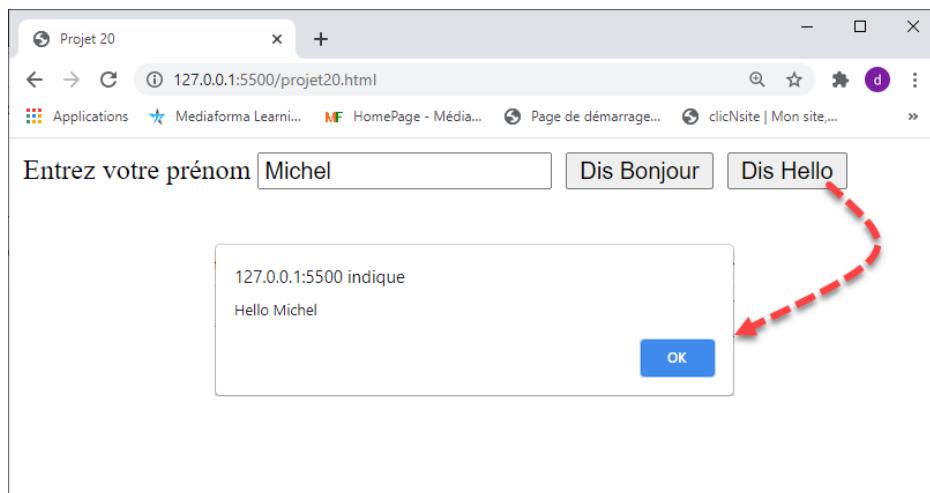
Challenge

Vous pouvez passer un ou plusieurs arguments à une fonction évènementielle. La technique est la même qu'en JavaScript. Pour le vérifier, ajoutez un autre bouton au code précédent.

Le premier affichera "Bonjour nom" dans une boîte de message.

Le deuxième affichera "Hello nom" dans une boîte de message.

Le texte "Bonjour" ou "Hello" sera passé en paramètre de la fonction évènementielle.



Solution :

```
<div id="app">
    <label>Entrez votre prénom </label><input type="text" v-
model="prenom">&nbsp;
    <button @click="bonjour('Bonjour')">Dis
Bonjour</button>&nbsp;
    <button @click="bonjour('Hello')">Dis Hello</button>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                prenom : ''
```

```
        }
    },
    methods : {
        bonjour(texte) {
            alert(texte + ' ' + this.prenom);
        }
    }
));
let vm = app.mount('#app');
</script>
```

L'objet \$event dans la gestion évènementielle

L'objet JavaScript **event** est créé chaque fois qu'un évènement se produit.

Par exemple, dans un évènement clavier, il donne accès (entre autres) à la touche ou à la combinaison de touches pressées.

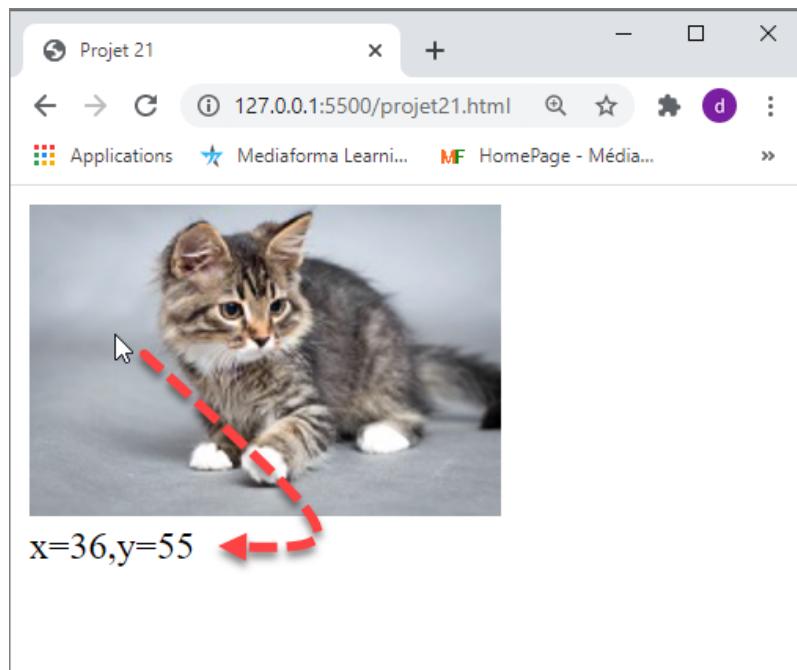
Ou encore dans un évènement souris, il donne accès (entre autres) aux coordonnées de la souris.

Pour accéder aux propriétés de l'objet JavaScript **event** dans le ViewModel, vous passerez l'objet **\$event** (et non l'objet **event**) à la fonction évènementielle.

Challenge

Affichez une image de taille moyenne et sans bords blancs.

Affichez les coordonnées du pointeur de la souris lorsqu'elle est au-dessus de l'image :



Solution (projet21) :

```
<div id="app">
    
        <div>{{xy}}</div>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                xy : ''
            }
        },
        methods : {
            affCoord(ev) {
                this.xy = 'x=' + ev.offsetX + ',y=' + ev.offsetY;
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

Les fonctions stopPropagation() et preventDefault() de l'objet \$event

Les fonctions **stopPropagation()** et **preventDefault()** sont souvent utilisées pour modifier le comportement par défaut du navigateur suite à un évènement. Mais que font au juste ces deux fonctions ?

- La fonction **stopPropagation()** stoppe la propagation de l'évènement aux parents de l'élément sur lequel l'évènement s'est produit.
- La fonction **preventDefault()** annule l'action par défaut du navigateur sur un évènement, mais l'évènement continue à être propagé aux parents de l'élément qui a produit l'évènement.

Pour bien comprendre le fonctionnement des fonctions **preventDefault()** et **stopPropagation()**, nous allons définir une application Vue qui contient une balise `<div></div>` et une balise `<input type="text">` imbriquées :

```
<div>
  <input type="text">
</div>
```

Je vais capturer l'évènement **keydown** dans la balise `<input>` et dans la balise `<div>`.

Dans le traitement évènementiel de la balise `<input>`, exécutez aucune, l'une, l'autre ou les deux fonctions **preventDefault()** et **stopPropagation()** appliquées à l'objet **\$event**.

Affichez des messages dans la console pour bien comprendre ce qu'il se passe.

Voici un petit code pour bien comprendre le fonctionnement des fonctions **preventDefault()** et **stopPropagation()**.

Projet 22 :

```
<div id="app">
  <div v-on :keydown="actionParent">
    <input type="text" v-on :keydown="action($event)" />
  </div>
</div>
```

```
<script>
    const app = Vue.createApp({
        methods : {
            action(ev) {
                console.log("Évènement traité dans le input");
                ev.preventDefault();
                ev.stopPropagation();
            },
            actionParent() {
                console.log("Évènement propagé jusqu'au div");
            }
        });
        let vm = app.mount('#app');
    </script>
```

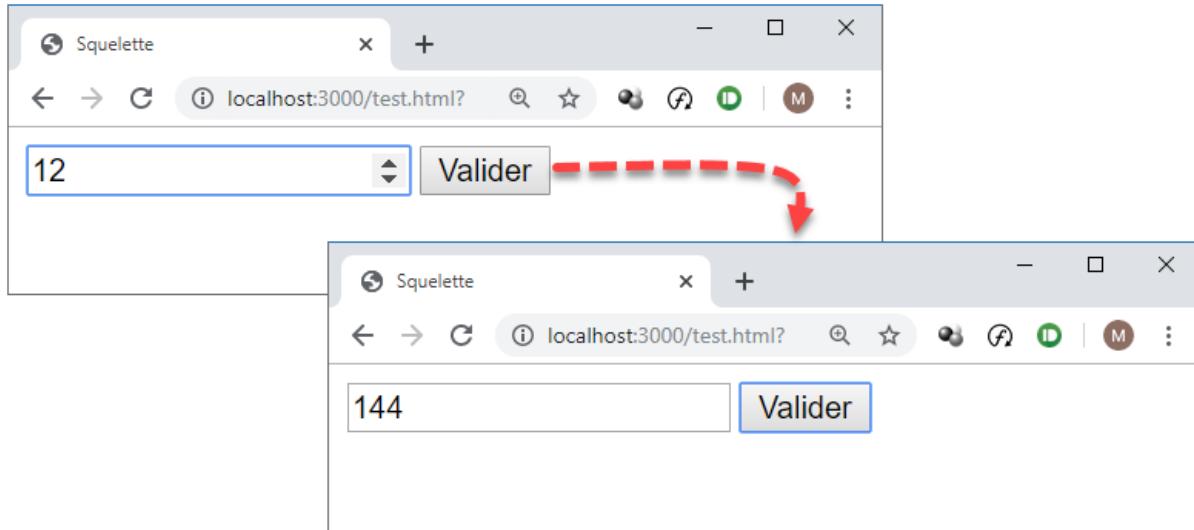
La fonction **preventDefault()** empêche l'affichage des touches tapées. La fonction **stopPropagation()** ne propage pas l'évènement au parent de l'input. Si on met la fonction **preventDefault()** en commentaire, les touches tapées apparaissent. Si on met la fonction **stopPropagation()** en commentaire, les frappes clavier sont propagées jusqu'au parent div.

Challenge

Définissez un formulaire contenant une zone de texte et un bouton de type **submit**.

L'utilisateur saisit un nombre dans la zone de texte puis il appuie sur le bouton.

Le carré du nombre saisi doit alors être affiché dans la zone de texte sans changer de page, même si le bouton est de type **submit**.



Solution :

```
<div id="app">
    <input type="text" v-model="nombre">
    <input type="submit" @click="carre($event)" value="Calculer le
carré">
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                nombre : null
            }
        },
        methods : {
            carre(ev) {
                ev.preventDefault();
                this.nombre = Math.pow(this.nombre, 2);
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

Suffixes stop et prevent de Vue

Vous pouvez simplifier l'utilisation des fonctions **stopPropagation()** et **preventDefault()** de l'objet JavaScript **event**. Pour cela, il suffit d'utiliser les modificateurs **.stop** et/ou **.prevent** en suffixe dans l'attribut HTML **v-on :evenement** :

Par exemple, ce code (projet 24) :

```
<input type="text" v-on :keydown="action($event)" />  
...  
methods : {  
    action(ev) {  
        ev.preventDefault();  
        ev.stopPropagation();  
        // Autres actions  
    }  
}
```

Est équivalent à :

```
<input type="text" v-on :keydown.prevent.stop="action" />  
...  
methods : {  
    action(ev) {  
        // Autres actions  
    }  
}
```

Voyons si ces modificateurs fonctionnent dans le challenge précédent.

Le code sans le modificateur

```
<div id="app">  
    <input type="text" v-model="nombre">  
    <input type="submit" @click="carre($event)" Value="Calculer  
le carré">  
</div>  
<script>  
    const app = Vue.createApp({  
        data() {  
            return {  
                nombre : null  
            }  
        },
```

```

methods : {
    carre(ev) {
        ev.preventDefault();
        this.nombre = Math.pow(this.nombre, 2);
    }
}
});

let vm = app.mount('#app');
</script>

```

Le code avec le modificateur

```

<div id="app">
    <input type="text" v-model="nombre">
    <input type="submit" @click.prevent="carre($event)"
Value="Calculer le carré">
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                nombre : null
            }
        },
        methods : {
            carre(ev) {
                this.nombre = Math.pow(this.nombre, 2);
            }
        }
    });
    let vm = app.mount('#app');
</script>

```

Voyons quelques autres modificateurs intéressants :

Clic unique :

```
<button v-on :click.once="action">Cliquez ici</button>
```

Action sur une touche du clavier :

```
<input type="text" v-on :keyup.code="action">
```

Où **code** est un code de touche (propriété **key** de l'objet **event**).

Voici quelques codes de touches utilisables :

enter, tab, delete, esc, space, up, down, right, left, page-down, page-up, f1 à f12, ctrl, alt, shift, etc.

Vous pouvez également utiliser ces modificateurs pour réagir (respectivement) au clic sur le bouton gauche, le bouton droit et le bouton central de la souris :

.left, .right et .middle.

Challenge

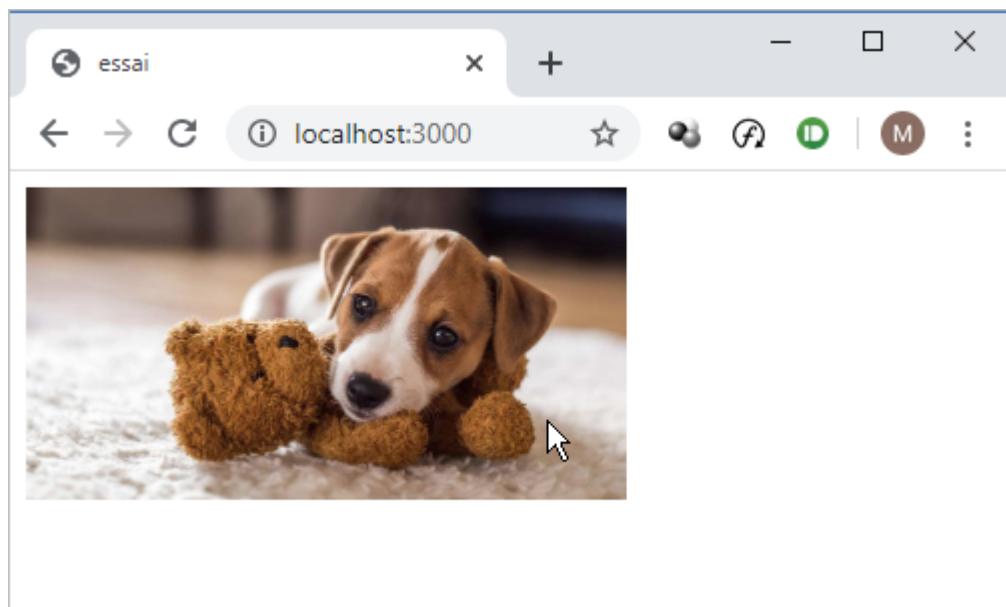
Affichez une image de taille moyenne.

Donnez-lui une largeur de 300px par défaut.

Au clic gauche sur l'image, affectez-lui une largeur de 200px.

Au clic droit sur l'image, affectez-lui une largeur de 100px.

Au clic central sur l'image, affectez-lui une largeur de 400px.

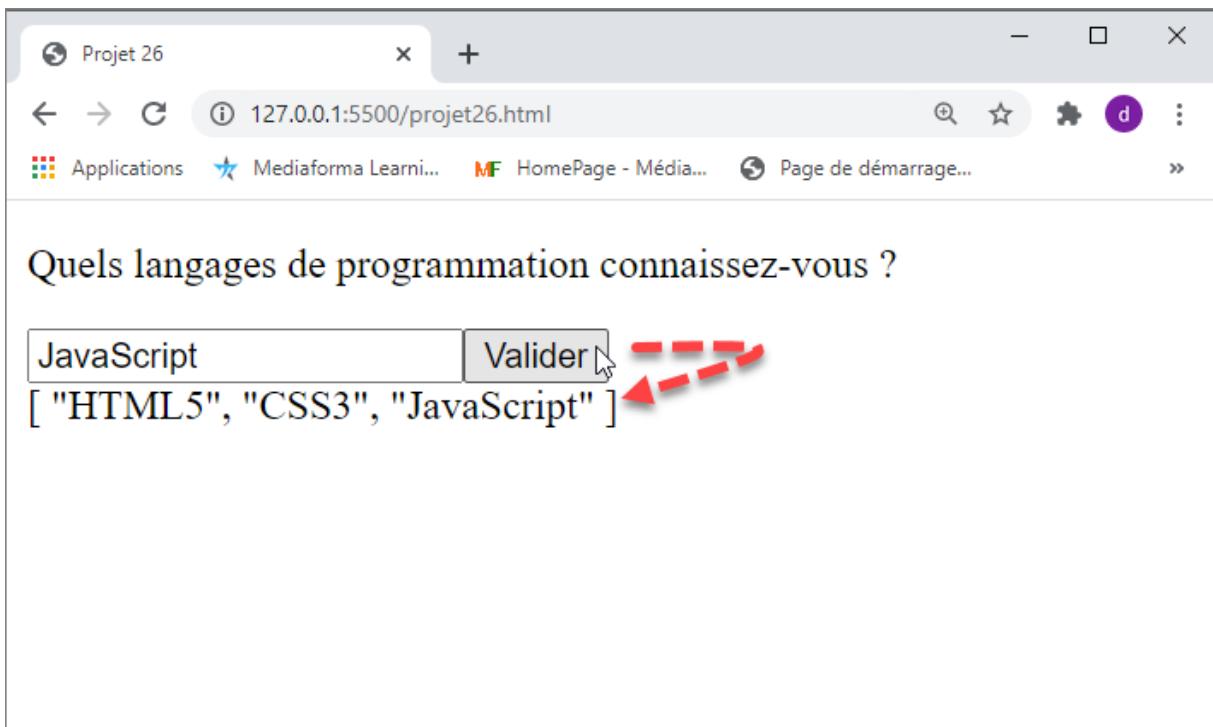


Solution (projet25) :

```
<div id="app">
  
</div>
<script>
const app = Vue.createApp({
  data() {
    return {
      valeur : 300
    }
  },
  methods : {
    largeur(lar) {
      this.valeur = lar;
    }
  }
});
let vm = app.mount('#app');
</script>
```

Challenge

Définissez un formulaire qui contient une zone de texte et un bouton. Demandez à l'utilisateur quels langages de programmation il connaît. Au clic sur le bouton, le langage saisi est ajouté à un tableau défini dans le modèle. Le contenu du tableau s'affiche sous la zone de texte et le bouton :



Si vous ne savez pas comment ajouter une donnée dans un tableau JavaScript, consultez la rubrique **Array methods** du site **w3schools**.

Solution :

```
<div id="app">
  <p>Quels langages de programmation connaissez-vous ?</p>
  <input type="text" v-model="unLang">
  <input type="button" value="Valider" @click="ajouter">
  <div>{{langages}}</div>
</div>
<script>
  const app = Vue.createApp({
    data() {
      return {
        unLang : '',
        langages : []
      }
    },
    methods : {
      ajouter() {
        this.langages.push(this.unLang);
      }
    }
  })
</script>
```

```

        }
    }
}) ;
let vm = app.mount('#app') ;
</script>

```

Affichage des langages dans une liste à puces

Plutôt que d'afficher le tableau **liste[]** avec une interpolation, nous allons maintenant afficher les différents langages dans une liste à puces.

Pour cela, nous allons utiliser la directive **v-for** dans une balise **** :

```

<ul>
<li v-for="lan in liste">{{lan}}</li>
<ul>

```

Voici ce que nous devrions obtenir :



Et voici le code HTML utilisé (projet27). Le code JavaScript est le même que celui du projet précédent (projet26) :

```

<div id="app">
<p>Quels langages de programmation connaissez-vous ?</p>
<input type="text" v-model="unLang">

```

```
<input type="button" value="Valider" @click="ajouter">
<ul>
  <li v-for="lan in langages">{{lan}}</li>
</ul>
</div>
```

Deux paramètres dans la directive v-for

Jusqu'ici, vous avez vu qu'il était possible d'écrire quelque chose comme ceci :

```
<ul>
  <li v-for="item in items">{{item}}</li>
</ul>
```

Pour connaître l'index de chaque élément, vous devez préciser un deuxième argument dans la directive **v-for** :

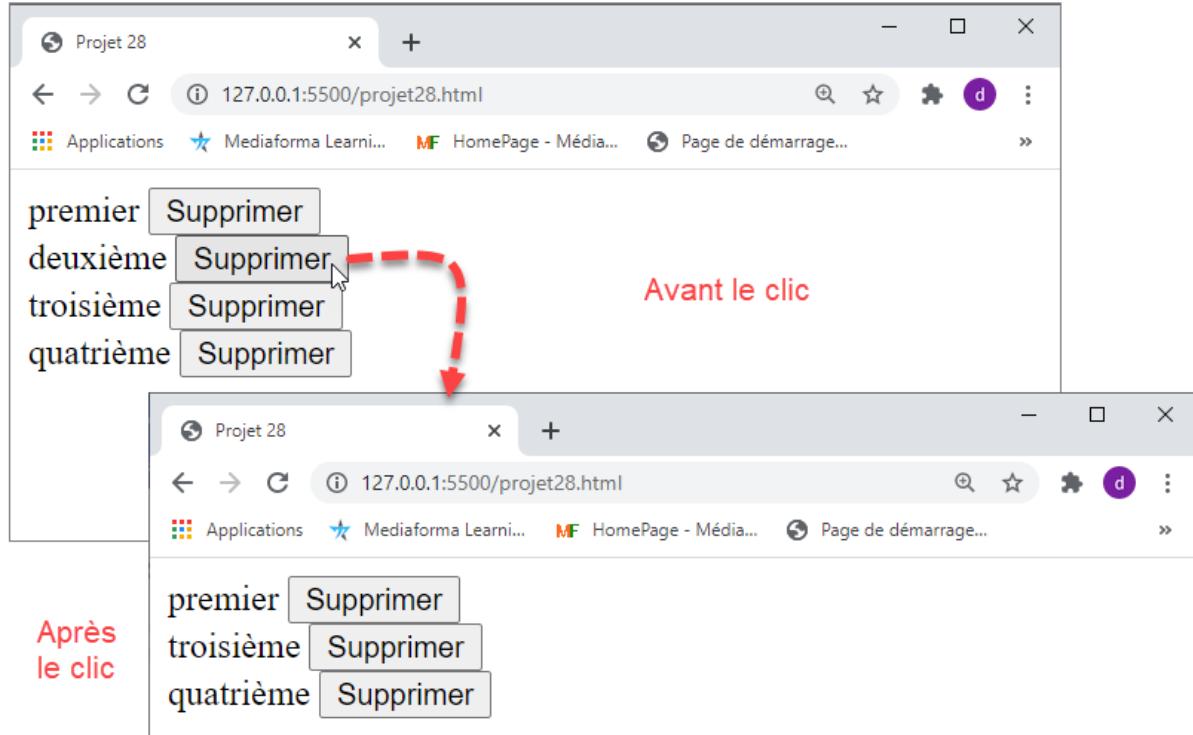
```
<ul>
  <li v-for="(item,index) in items">{{index}} - {{item}}</li>
</ul>
```

Challenge

Créez une nouvelle application Vue. Dans le modèle, définissez le tableau **elems** qui contient les chaînes "premier", "deuxième", "troisième" et "quatrième".

Affichez ces chaînes dans un **<div>** à l'aide d'une directive **v-for**.

Ajoutez un bouton après chaque chaîne et supprimez la chaîne correspondante lorsque l'utilisateur clique dessus.



Si vous ne savez pas comment supprimer une donnée dans un tableau JavaScript, consultez la rubrique **Array methods** du site **w3schools**

Solution (projet28) :

```
<div id="app">
    <div v-for="(elem, id) in elems">
        {{elem}}
        <button @click="suppr(id)">Supprimer</button>
    </div>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                elems :
                ['premier', 'deuxième', 'troisième', 'quatrième']
            }
        },
        methods : {
            suppr(i) {
                this.elems.splice(i,1);
            }
        }
    })
    app.mount('#app')
</script>
```

```
        }
    }
}) ;
let vm = app.mount('#app') ;
</script>
```

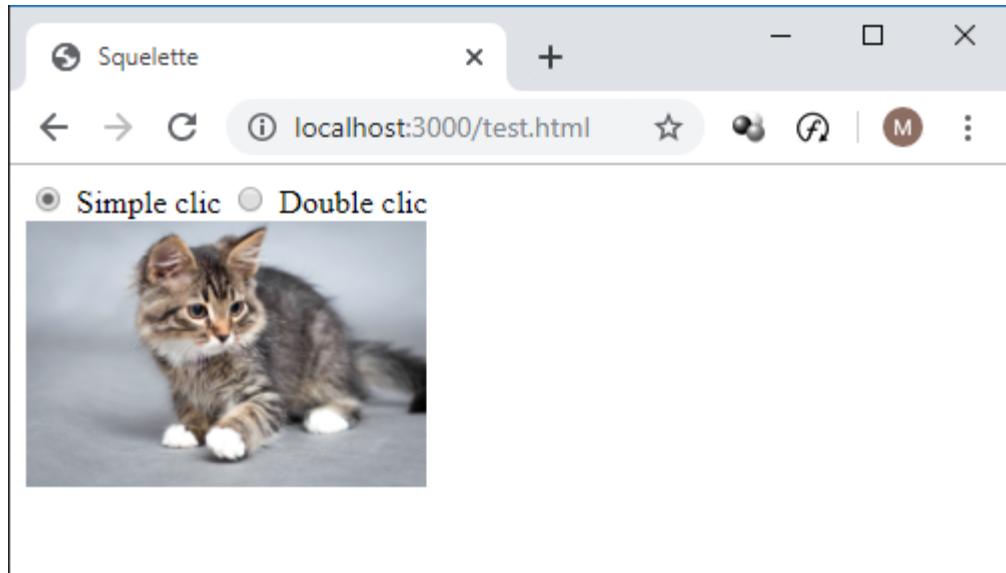
Arguments dynamiques

L'évènement ou l'attribut spécifié après (respectivement) un **v-on** ou un **v-bind** peut être dynamique, c'est-à-dire issu d'une propriété définie dans le modèle. Pour cela, vous encadrerez l'évènement ou l'attribut dynamique par des crochets :

```
<element v-on :[evenement]="traitement"></element>
<element v-bind :[attribut]="valeur"></element>
```

Challenge

Modifiez le code précédent pour changer d'image au clic ou au double-clic en fonction du bouton radio sélectionné :



Solution :

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

```

<input type="radio" v-model="choix" value="click" checked>
<label>Clic</label>
    <input type="radio" v-model="choix" value="dblclick">
<label>Double clic</label><br>
    
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                choix : 'click',
                taille : 200
            }
        },
        methods : {
            modifierTaille() {
                this.taille = this.taille*1.1;
            }
        }
    });
    let vm = app.mount('#app');
</script>

```

Directive v-for pour parcourir des objets JSON

La directive **v-for** peut être utilisée pour parcourir un tableau d'objets JSON.

```

<balise v-for="element in elements">
    {{element.prop1}} ... {{element.propN}}
</balise>

```

Où :

- **<balise>** est une balise HTML quelconque ;
- **elements** est l'objet JSON à parcourir ;
- **element** est un des éléments de l'objet JSON **elements** ;

- les **propriétés** sont les propriétés des éléments à afficher.

Challenge

Définissez l'objet **personnes** dans le modèle. Affectez-lui plusieurs personnes à l'aide d'objets JSON. Voici le format d'une personne :

```
{
  prenom : 'Pierrick',
  nom : 'Martin',
  age : 24
}
```

En utilisant une directive **v-for**, parcourrez l'objet **personnes** et affichez les personnes sous cette forme :



Solution (projet30) :

```
<div id="app">
  <ul>
    <li v-for="personne in personnes">
      {{personne.prenom}} {{personne.nom}}, {{personne.age}}
    </li>
  </ul>
</div>
<script>
  const app = Vue.createApp({
```

```
data() {
    return {
        personnes : [
            {prenom : 'Eric', nom : 'Martin', age : 30},
            {prenom : 'Ever', nom : 'Harmaan', age : 30},
            {prenom : 'Kévin', nom : 'Martin', age : 27},
            {prenom : 'Ai', nom : 'Li', age : 25}
        ]
    }
}

)) ;
let vm = app.mount('#app') ;
</script>
```

Gestion des évènements clavier

JavaScript est capable d'intercepter trois évènements clavier : **keydown**, **keyup** et **keypress**.

- **keydown** est généré lorsqu'une touche est appuyée
- **keyup** est généré lorsqu'une touche est relâchée
- **keypress** est généré lorsqu'une touche est pressée puis relâchée

Ces trois évènements peuvent être utilisées dans les applications Vue.

Challenge

Définissez un formulaire qui contient un champ texte et un bouton.

Lorsque l'utilisateur clique sur le bouton ou appuie sur la touche *Entrée* lorsque le champ texte a le focus, le contenu du champ texte doit s'afficher en majuscules.

En bonus, donnez le focus au champ texte dès l'affichage de la page.

Indice :

Passez l'objet **event** à la fonction évènementielle pour récupérer le caractère tapé. Dans le jargon Vue.js, **event** s'écrit **\$event**. Testez la touche tapée et agissez s'il s'agit de la touche *Entrée*.

Solution (projet31) :

```
<div id="app">
    <input type="text" v-model="texte" @keyup.enter="maj">&nbsp;
    <input type="button" value="Majuscule" @click="maj">
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                texte : ''
            }
        },
        methods : {
            maj () {
                this.texte = this.texte.toUpperCase();
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

La directive v-cloak

Il arrive parfois que les double-accolades des interpolations apparaissent sur les pages générées avec Vue.js. Ceci parce que les expressions à afficher ne sont pas encore calculées par Vue.js.

Si cela vous arrive, vous pouvez utiliser la directive **v-cloak**. Voici un exemple :

```
<div v-cloak>
    {{ message }}
</div>
```

Pour que les éléments contenant une directive **v-cloak** ne soient pas affichés tant que Vue n'est pas prêt, vous ajouterez ce style CSS dans une feuille de styles interne ou externe :

```
[v-cloak] {  
    display : none;  
}
```

La directive v-once

Il est parfois nécessaire qu'un élément ne soit rendu qu'une fois et que ses mises à jour n'apparaissent pas dans la page. Pour cela, vous utiliserez la directive **v-once**.

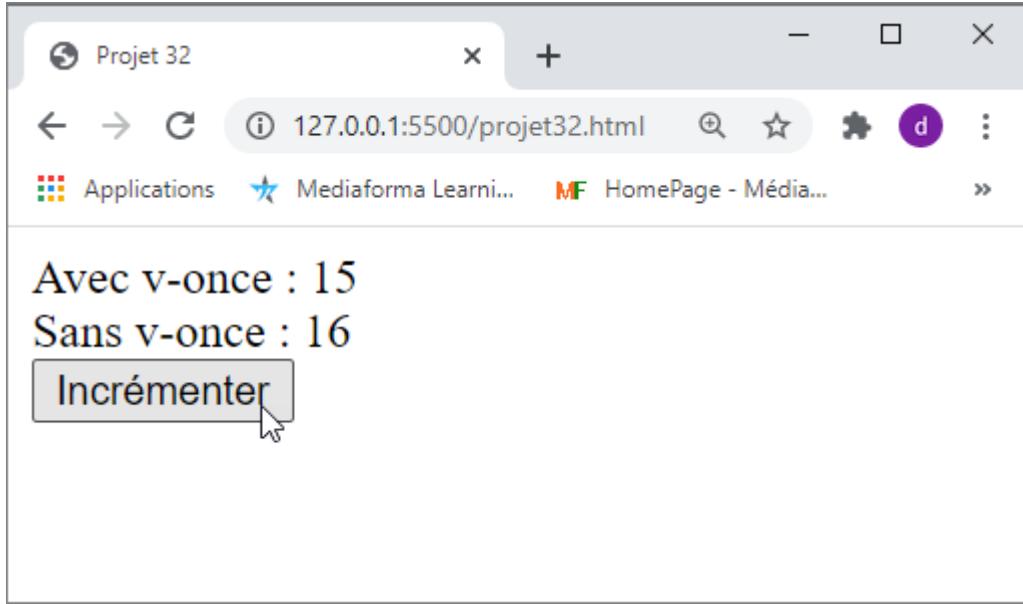
Challenge

Définissez une nouvelle application.

Dans le modèle, définissez la variable **nombre** et affectez-lui la valeur (par exemple) **15**.

Affichez cette variable avec et sans la directive **v-once**.

Ajoutez un bouton qui incrémente cette variable lorsqu'il est cliqué. Que se passe-t-il ?



Solution (projet32) :

Dans la première **div**, la directive **v-once** empêche la mise à jour de l'interpolation. La valeur affichée reste donc figée à **15** :

```
<div id="app">
    <div v-once>Avec v-once : {{nombre}}</div>
    <div>Sans v-once : {{nombre}}</div>
    <button @click="plusUn">Incrémenter</button>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                nombre : 15
            }
        },
        methods : {
            plusUn() {
                this.nombre++;
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

Classes conditionnelles

Il est possible d'appliquer un style en fonction d'une condition mémorisée dans le modèle.

Pour cela, utilisez une directive **v-bind** sur l'attribut HTML **class** :

```
v-bind :class="{ classe : valeur }"
```

Où :

- **classe** est la classe à affecter (ou non)
- **valeur** est une valeur booléenne.

Si **valeur** vaut **true**, la classe est appliquée. Si **valeur** vaut **false**, la classe n'est pas appliquée.

Challenge

Modifiez le code ci-dessous (projet 33) :

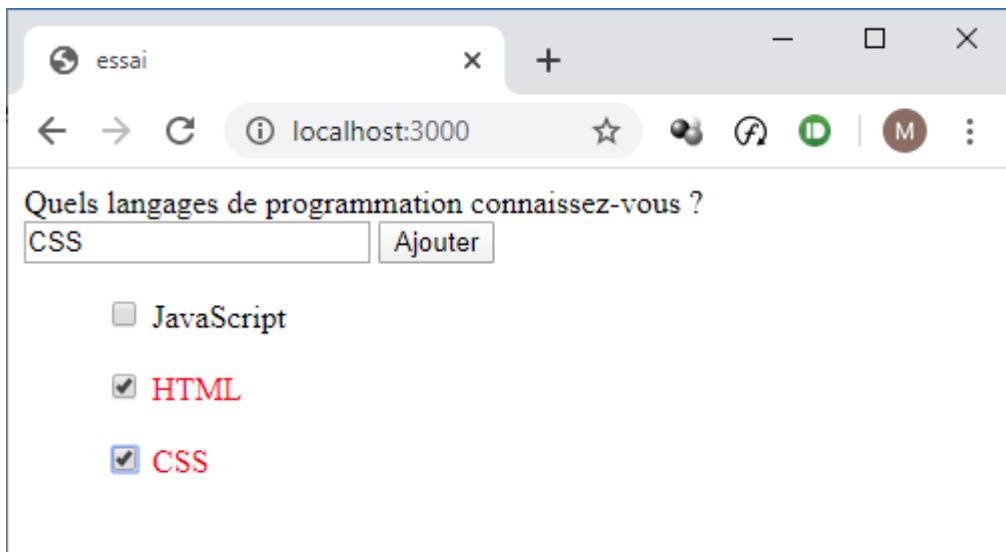
```
<div id="app">
    <p>Quels langages de programmation connaissez-vous ?</p>
    <input type="text" placeholder="Entrez un langage" v-
model="langage" />
    <input type="button" value="Valider" v-
on :click="traitement" /><br />
    <ul>
        <li v-for="el in liste">{{el}}</li>
    </ul>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                langage : "",
                liste : []
            }
        },
        methods : {
```

```

        traitement : function () {
            this.liste.push(this.langage);
        }
    });
let vm = app.mount('#app');
</script>

```

Pour faire précéder chacun des éléments de la liste à puces par des cases à cocher. Lorsqu'une case est cochée, l'élément correspondant se colore en rouge. Lorsqu'elle est décochée, l'élément correspondant retrouve sa couleur par défaut (noir) :



Conseil :

Mémorisez dans le modèle le langage et l'état de la case à cocher (**false** par défaut) lorsque l'utilisateur clique sur le bouton **Ajouter**.

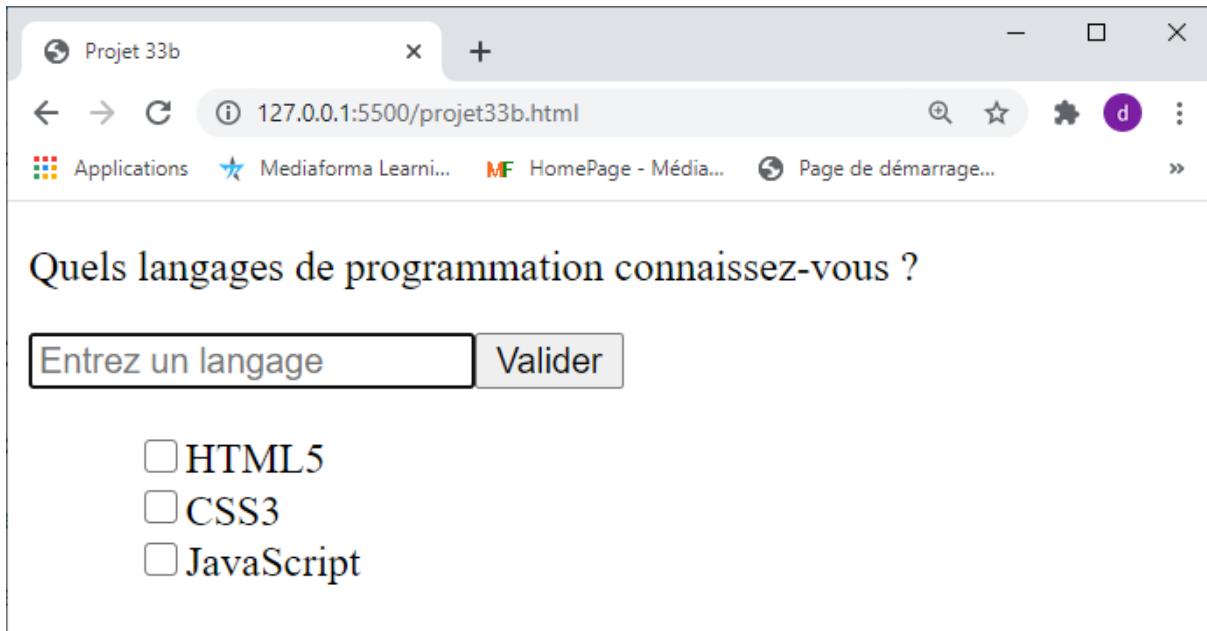
Vous allez modifier le code précédent pour remplacer les puces de la liste par des cases à cocher, c'est-à-dire par des balises :

```
<input type="checkbox">
```

Vous allez mémoriser l'état des checkbox dans une variable booléenne dans le tableau **liste**. Vous devrez donc ne plus stocker simplement les langages entrés par l'utilisateur dans le tableau **liste**, mais également l'état des cases à cocher. Pour cela, le plus simple va consister à mémoriser des objets JSON dans le tableau **liste** :

```
{
    lang : 'un langage',
    etatCase : true ou false pour indiquer l'état de la case à cocher
}
```

Voici ce que vous devez obtenir :



Solution (projet 33b) :

```
<div id="app">
    <p>Quels langages de programmation connaissez-vous ?</p>
    <input type="text" placeholder="Entrez un langage" v-
model="langage" />
    <input type="button" value="Valider" v-
on :click="traitement" /><br />
    <ul>
        <li v-for="el in liste">
            <input type="checkbox" v-model="el.etatCase">
            <span :class="{rouge :el.etatCase}">{{el.lang}}</span>
        </li>
    </ul>
</div>
<script>
    const app = Vue.createApp({
        data() {
```

```

        return {
            langage : '',
            liste : []
        }
    },
methods : {
    traitement : function () {
        this.liste.push(
            {lang : this.langage,
            etatCase : false});
    }
}
)) ;
let vm = app.mount('#app') ;
</script>

```

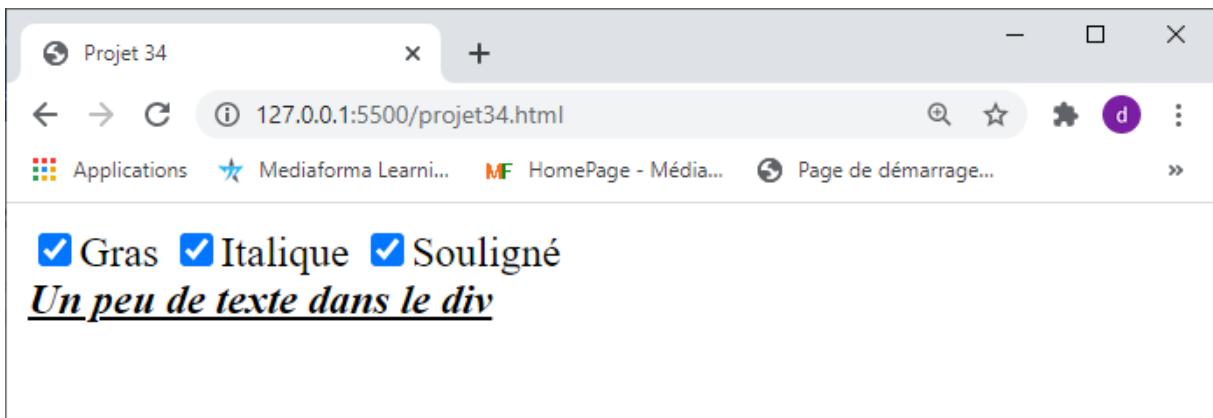
Explications :

- La méthode **traitement()** insère des objets JSON dans le tableau **liste** composés du langage et de l'état de la case à cocher (**false** par défaut).
- Les puces sont supprimées en CSS.
- Les checkbox sont reliés à la propriété booléenne **etatCase** . Lorsqu'ils changent d'état, **etatCase** est mis à jour.
- La directive **:class** applique la classe **rouge** au nom du langage si **el.etatCase** vaut **true**.

Challenge

Plusieurs classes peuvent être ajoutées à un élément avec la directive **v-bind :class**. Il suffit pour cela de préciser plusieurs couples **clé : valeur** dans les accolades qui suivent le **:class** et de les séparer entre eux par des virgules.

Définissez un **<div>**. Insérez-y un peu de texte. Contrôlez les attributs **gras**, **italique** et **souligné** de ce texte à l'aide de cases à cocher :



Solution (projet34) :

```
<style>
    li {
        list-style-type : none;
    }
    .g {
        font-weight : bold;
    }
    .i {
        font-style : italic;
    }
    .s {
        text-decoration : underline;
    }
</style>
</head>

<body>
    <div id="app">
        <input type="checkbox" v-model="gras">Gras
        <input type="checkbox" v-model="ital">Italique
        <input type="checkbox" v-model="soul">Souligné<br>
        <div :class="{g :gras, i :ital, s :soul}">Un peu de texte
dans le div</div>
    </div>
    <script>
        const app = Vue.createApp({
```

```
        data() {
            return {
                gras : false,
                ital : false,
                soul : false
            }
        }
    });
    let vm = app.mount('#app');
</script>
</body>
```

Gestion des styles inline

Vous savez modifier la classe d'éléments HTML à partir de propriétés définies dans le modèle. Pour cela, vous bindez l'attribut class à une propriété du modèle.

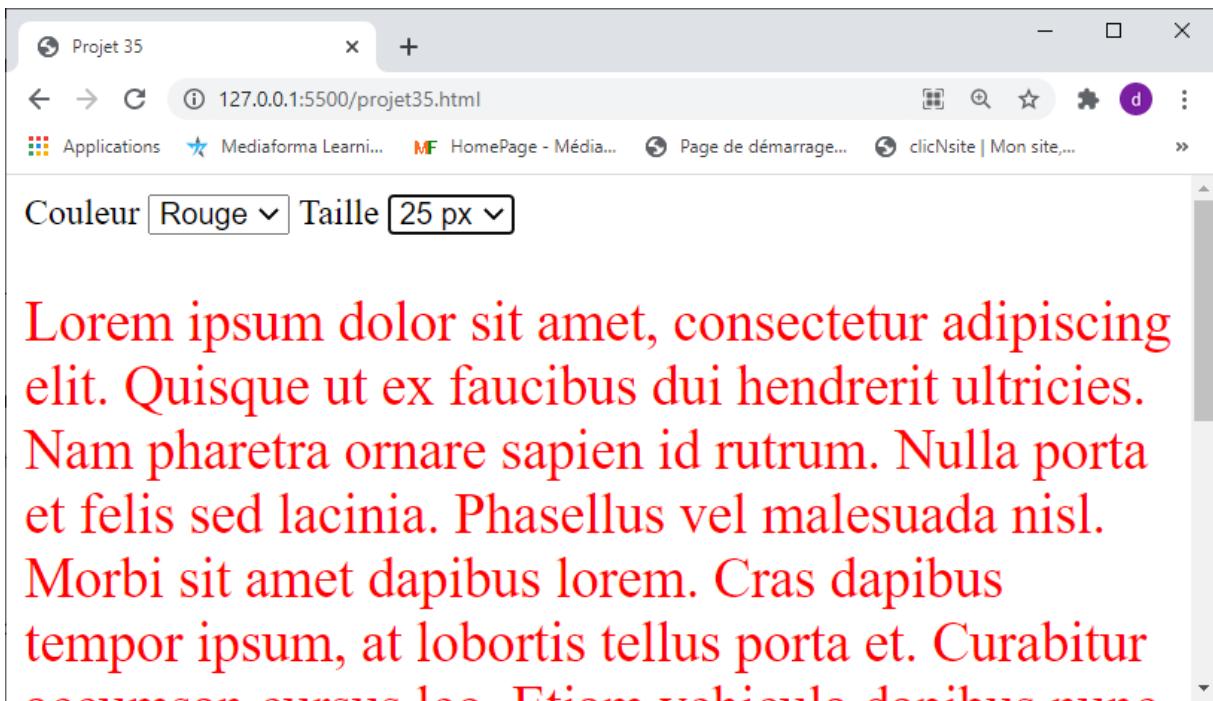
Voyons maintenant comment modifier n'importe quel style inline dans une balise à partir de propriétés définies dans le modèle.

Pour cela, vous utiliserez la directive **v-bind :style** (ou son alias **:style**) et vous lui affecterez un objet JSON :

```
v-bind :style = "{ pcss1 : pvue1, pcss2 : pvue2, pcss3 : pvue3, ... }"
```

Challenge

Définissez un **<div>** qui contient du texte *lorem ipsum*. Ajoutez deux listes déroulantes pour contrôler la couleur (rouge, vert, bleu, noir) et la taille (10px, 15px, 20px, 25px) du texte affiché dans le **<div>**.



Solution (projet35) :

```
<div id="app">
    Couleur
    <select v-model="couleur">
        <option value="black" selected>Noir</option>
        <option value="red">Rouge</option>
        <option value="green">Vert</option>
        <option value="blue">Bleu</option>
    </select>
    Taille
    <select v-model="taille">
        <option value="16px" selected>16 px</option>
        <option value="25px">25 px</option>
        <option value="40px">40 px</option>
        <option value="60px">60 px</option>
    </select>
    <p :style="{color : couleur, fontSize : taille}">Lorem ipsum
dolor sit amet, consectetur adipiscing elit. Quisque ut ex faucibus
dui hendrerit ultricies. Nam pharetra ornare sapien id rutrum.
Nulla porta et felis sed lacinia. Phasellus vel malesuada nisl.
Morbi sit amet dapibus lorem. Cras dapibus tempor ipsum, at
lobortis tellus porta et. Curabitur accumsan cursus leo. Etiam
```

```

vehicula dapibus nunc vitae facilisis. Nam sagittis placerat
sollicitudin.</p>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                couleur : 'black',
                taille : '16px'
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

Remarques :

- Utilisez la propriété **fontSize** (comme en JavaScript) et non **font-size** dans le style.
- Les **<select>** sont liés au modèle avec la directive **v-model**.
- La valeur sélectionnée dans la liste est passée au **<select>** avec l'attribut **value**.

Afficher du code HTML conditionnellement

Les directives **v-if**, **v-else-if** et **v-else** permettent d'afficher ou de cacher la balise dans laquelle elles sont incluses :

```

<div v-if="cond1">
    contenu 1
</div>
<div v-else-if="cond2">
    contenu 2
</div>
<div v-else="cond3">
    contenu 3
</div>
<div v-else>
```

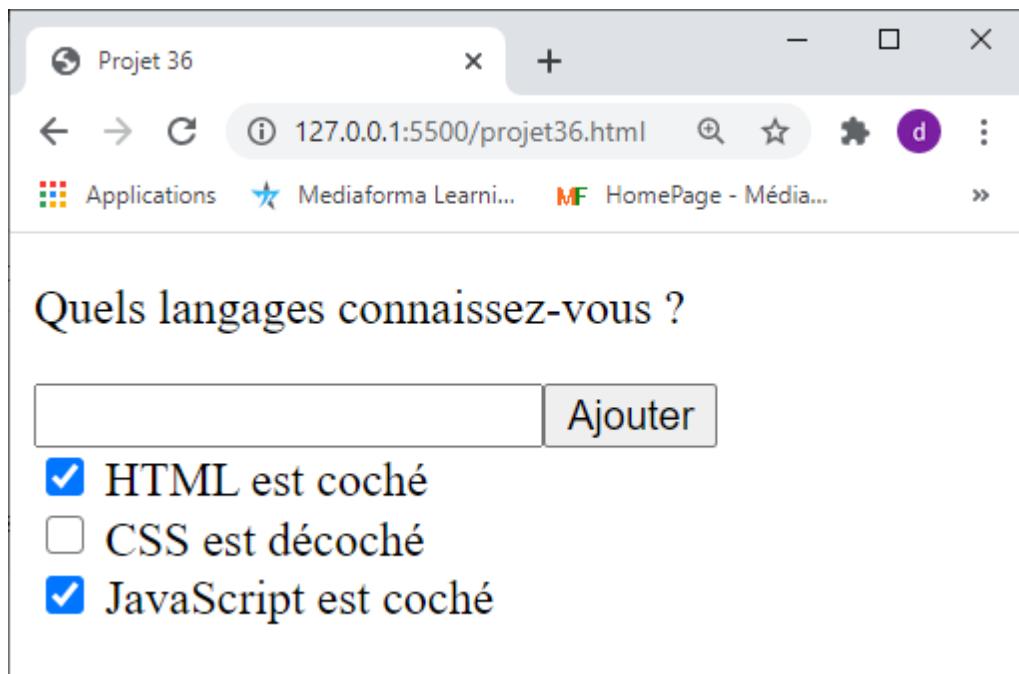
```
contenu 4  
</div>
```

Ici :

- **contenu 1** sera affiché si la **cond1** vaut **true**.
- **contenu 2** sera affiché si la **cond2** vaut **true**.
- **contenu 3** sera affiché si la **cond3** vaut **true**.
- **contenu 4** sera affiché si ni **cond1**, ni **cond2**, ni **cond3** ne vaut **true**.

Challenge

Définissez une application qui permet d'ajouter des éléments dans une liste où chaque élément est précédé d'une case à cocher. Lorsqu'une case est cochée, le texte "**est coché**" suit l'élément. Lorsqu'une case est décochée, le texte "**est décoché**" suit l'élément :



Solution (projet36) :

```
<div id="app">  
  <p>Quels langages connaissez-vous ?</p>  
  <input type="text" v-model="texte">  
  <input type="button" value="Ajouter" @click="ajouter">  
  <div v-for="langage in langages">
```

```

<input type="checkbox" v-model="langage.etat">
{{ langage.lang}}
<span v-if="langage.etat">est coché</span>
<span v-else>est décoché</span>
</div>
</div>
<script>
const app = Vue.createApp({
  data() {
    return {
      langages : [],
      texte : ''
    }
  },
  methods : {
    ajouter() {
      this.langages.push({ lang : this.texte, etat :
false });
    }
  }
});
let vm = app.mount('#app');
</script>

```

Ici, on utilise les directives **v-if** et **v-else** pour afficher un texte ou un autre en suffixe de l'élément.

Afficher du code HTML conditionnellement

Pour afficher conditionnellement un code HTML, vous pouvez également utiliser la directive **v-show** :

```
<div v-show="condition">Je suis affiché si condition vaut
true</div>
```

Contrairement à la directive **v-if**, le code affiché avec la directive **v-show** est toujours dans le DOM.

Si la condition vaut **false**, la propriété CSS **display** de l'élément est mise à **none** par Vue.

Super challenge

Vous avez brillamment terminé cette première partie. Maintenant, vous savez :

- Créer des applications Vue.js 3.
- Définir des propriétés dans le modèle et y accéder dans la vue avec un binding ou une interpolation.
- Effectuer un binding bidirectionnel entre un élément de formulaire et le modèle de l'application.
- Définir des méthodes et des propriétés calculées.
- Utiliser la directive **v-html** pour injecter une propriété du modèle dans le innerHTML d'un élément.
- Gérer les événements sans ou avec des paramètres, y compris le paramètre **\$event**.
- Utiliser les modificateurs **stop** et **prevent** pour gérer la propagation et l'interprétation des événements.
- Créer des boucles dans la vue avec la directive **v-for**.
- Utiliser des arguments dynamiques dans la vue.
- Créer des classes et des styles conditionnels.

Il est temps d'appliquer ce que vous avez appris dans un projet de plus grande envergure. Vous allez utiliser les données qui se trouvent dans le fichier JSON fourni en ressource (**personnes.json**) :

The screenshot shows the Firefox developer tools JSON viewer open in a new tab. The URL is `/C:/vue3/ressources/personnes.json`. The JSON structure represents a single person object with various properties like gender, name, location, coordinates, timezone, and login details.

```
JSON      Données brutes      En-têtes
Enregistrer  Copier  Tout réduire  Tout développer  Filtrer le JSON

0:
  gender: "female"
  name:
    title: "Mrs"
    first: "Elizabeth"
    last: "Ambrose"
  location:
    street:
      number: 4954
      name: "Tecumseh Rd"
      city: "Hudson"
      state: "Alberta"
      country: "Canada"
      postcode: "N7H 5G1"
    coordinates:
      latitude: "-67.5091"
      longitude: "66.5485"
    timezone:
      offset: "+1:00"
      description: "Brussels, Copenhagen, Madrid, Paris"
      email: "elizabeth.ambrose@example.com"
  login:
    uuid: "50777e60-394d-4951-b933-a2f78599065d"
    username: "bluemouse267"
    password: "christop"
    salt: "Ms1KvgCJ"
    md5: "f0fdd5f0811abcbe8d26f871e4dce7dc"
```

Je vous propose de créer une application et d'y inclure ces données JSON. Le premier but est d'afficher certaines des informations issues de ce fichier :

Super Projet 1

127.0.0.1:5500/superprojet1.html

Filtre sur le nom

Options

Homme Femme Téléphone Photo Moyen

	Mrs Elizabeth Ambrose Téléphone 366-033-8063 Pays Canada
	Mr Batista Silveira Téléphone (53) 5553-2892 Pays Brazil
	Mr Blaise Adam Téléphone 079 493 00 77 Pays Switzerland

En utilisant le fieldset **Filtre sur le nom**, vous filtrerez les personnes affichées. Ici par exemple, en saisissant la lettre **A** majuscule, seules les deux personnes dont le nom comporte cette lettre apparaissent :

Super Projet 1

127.0.0.1:5500/superprojet1.html

Filtre sur le nom

Options

Homme Femme Téléphone Photo Moyen Mise en forme

	Mrs Elizabeth Ambrose Téléphone 366-033-8063 Pays Canada
	Mr Blaise Adam Téléphone 079 493 00 77 Pays Switzerland

Définissez un deuxièmefieldset que vous appellerez **Options** pour ajouter d'autres filtres :

- Cochez la case **Homme** pour afficher les hommes, décochez-la pour ne pas les afficher.
- Cochez la case **Femme** pour afficher les femmes, décochez-la pour ne pas les afficher.
- Cochez la case **Téléphone** pour inclure le téléphone dans les informations affichées, décochez-la pour ne pas l'inclure.
- Choisissez la taille des photos entre **Petit**, **Moyen** et **Grand**.
- Cochez la case **Mise en forme** pour modifier l'arrière-plan, la police de caractères (**Georgia**) et la taille des caractères (**1,2rem**) des éléments affichés.

Voici un exemple de filtrage/mise en forme. Ici, le téléphone n'est pas affiché, la photo est petite et une mise en forme est appliquée aux données :

Vous pourrez pour cela utiliser des bindings simples et bidirectionnels, des directive **v-for** et **v-if** et tout autre moyen nécessaire pour obtenir le résultat demandé.

C'est à vous de coder. Prenez votre temps et avancez pas à pas pour créer cette mise en pratique de ce qui a été appris dans le premier chapitre.

Solution (superprojet1.html) :

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="https://unpkg.com/vue@next"></script>
```

```
<title>Super Projet 1</title>

<style>
    input[type='checkbox'], select {
        margin-right : 1rem;
    }
    img {
        float : left;
        margin-right : 1rem;
    }
    .personne {
        padding : 1rem;
        clear : left;
    }
    .misEnForme {
        background : #e0ffff;
        font-family : georgia;
        font-size : 1.2rem;
        margin : 1rem;
    }
</style>

</head>

<body>
    <div id="app">
        <fieldset>
            <legend>Filtre sur le nom</legend>
            <input type="text" v-model="filtreNom">
        </fieldset>
        <fieldset>
            <legend>Options</legend>
```

```

Homme <input type="checkbox" checked v-model="homme">
Femme <input type="checkbox" checked v-model="femme">
Téléphone <input type="checkbox" checked v-model="tel">
Photo

<select v-model="taille">
    <option value="1">Petit</option>
    <option selected value="2">Moyen</option>
    <option value="3">Grand</option>
</select>

Mise en forme <input type="checkbox" v-model="mef">
</fieldset>

<div class="personne" v-bind :class="{misEnForme : mef}" v-for="p in personnes">
    <div v-if="(((homme &&p.name.title=='Mr') || (femme
&&p.name.title=='Mrs')) && (p.name.last.indexOf(filtreNom)>=0 ||

filtreNom==''))">
        
        
        
        <span>
            {{p.name.title}} {{p.name.first}}
            {{p.name.last}}
        </span>
        <div v-if="tel">Téléphone {{p.phone}}</div>
        <div>Pays {{p.location.country}}</div>
    </div>
</div>
<script>
```

```
const app = Vue.createApp({  
  data() {  
    return {  
      homme : true,  
      femme : true,  
      tel : true,  
      taille : 2,  
      filtreNom : '',  
      mef : false,  
      personnes : [  
        {  
          "gender" : "female",  
          "name" : {  
            "title" : "Mrs",  
            "first" : "Elizabeth",  
            "last" : "Ambrose"  
          },  
          "location" : {  
            "street" : {  
              "number" : 4954,  
              "name" : "Tecumseh Rd"  
            },  
            "city" : "Hudson",  
            "state" : "Alberta",  
            "country" : "Canada",  
            "postcode" : "N7H 5G1",  
            "coordinates" : {  
              "latitude" : "-67.5091",  
              "longitude" : "66.5485"  
            },  
          }  
        },  
      ]  
    }  
  }  
})
```

```
        "timezone" : {
            "offset" : "+1 :00",
            "description" : "Brussels,
Copenhagen, Madrid, Paris"
        }
    },
    "email" :
"elizabeth.ambrose@example.com",
    "login" : {
        "uuid" : "50777e60-394d-4951-b933-
a2f78599065d",
        "username" : "bluemouse267",
        "password" : "christop",
        "salt" : "Ms1KvgCJ",
        "md5" :
"f0fdd5f0811abcbe8d26f871e4dce7dc",
        "sha1" :
"85dfd62ef93278b153a6fca54692f72c983fcbb7",
        "sha256" :
"f5f6a95060d7d2bcf4936df32a6d7a1dadeac059bdf10d0b3867db3c32c16a20"
    },
    "dob" : {
        "date" : "1995-11-
20T00 :31 :55.940Z",
        "age" : 26
    },
    "registered" : {
        "date" : "2010-12-
21T12 :23 :45.450Z",
        "age" : 11
    },
    "phone" : "366-033-8063",
    "cell" : "205-901-7452",
```

```
        "id" : {
            "name" : "",
            "value" : null
        },
        "picture" : {
            "large" :
"https ://randomuser.me/api/portraits/women/49.jpg",
            "medium" :
"https ://randomuser.me/api/portraits/med/women/49.jpg",
            "thumbnail" :
"https ://randomuser.me/api/portraits/thumb/women/49.jpg"
        },
        "nat" : "CA"
    },
{
    "gender" : "male",
    "name" : {
        "title" : "Mr",
        "first" : "Batista",
        "last" : "Silveira"
    },
    "location" : {
        "street" : {
            "number" : 1153,
            "name" : "Rua Das Flores "
        },
        "city" : "Eunápolis",
        "state" : "Pará",
        "country" : "Brazil",
        "postcode" : 78657,
        "coordinates" : {

```

```
        "latitude" : "-86.0633",
        "longitude" : "45.8313"
    },
    "timezone" : {
        "offset" : "+9 :30",
        "description" : "Adelaide,
Darwin"
    }
},
"email" :
"batista.silveira@example.com",
"login" : {
    "uuid" : "5308f02b-7838-4b48-8114-
2fe1349c0819",
    "username" : "ticklishmeercat723",
    "password" : "hjkl",
    "salt" : "aGeJRlo8",
    "md5" :
"a7e9382c40f664d1aec39f43a02de4fc",
    "sha1" :
"e6238172483ef79815aa6e2d3a6d55a7e8faccd0",
    "sha256" :
"418c665cd8f8f42e34c7825b0d7e327ab22f56d40c8d7d5dd0c0822863d6c393"
},
"dob" : {
    "date" : "1978-03-
16T14 :00 :38.384Z",
    "age" : 43
},
"registered" : {
    "date" : "2002-11-
14T17 :14 :04.368Z",
    "age" : 19
}
```

```
        } ,  
        "phone" : "(53) 5553-2892",  
        "cell" : "(04) 3893-4079",  
        "id" : {  
            "name" : "",  
            "value" : null  
        } ,  
        "picture" : {  
            "large" :  
                "https ://randomuser.me/api/portraits/men/50.jpg",  
            "medium" :  
                "https ://randomuser.me/api/portraits/med/men/50.jpg",  
            "thumbnail" :  
                "https ://randomuser.me/api/portraits/thumb/men/50.jpg"  
        } ,  
        "nat" : "BR"  
    } ,  
    {  
        "gender" : "male",  
        "name" : {  
            "title" : "Mr",  
            "first" : "Blaise",  
            "last" : "Adam"  
        } ,  
        "location" : {  
            "street" : {  
                "number" : 3056,  
                "name" : "Rue de la Baleine"  
            } ,  
            "city" : "Courrendlin",  
            "state" : "Valais",  
            "lat" : 46.0201,  
            "lon" : 7.3851  
        } ,  
        "coordinates" : {  
            "lat" : 46.0201,  
            "lon" : 7.3851  
        } ,  
        "timezone" : "Europe/Zurich",  
        "atmosphere" : {  
            "humidity" : 87,  
            "pressure" : 1013,  
            "temperature" : 15.2  
        }  
    } ,  
    "dob" : {  
        "age" : 24,  
        "date" : "1994-05-12T00:00:00Z"  
    } ,  
    "registered" : {  
        "date" : "2010-04-19T07:13:47Z",  
        "age" : 9  
    } ,  
    "password" : "1234567890",  
    "hash" : "5f4dbd60dd078d3a0c71331e4a63a7c9",  
    "salt" : "1234567890",  
    "md5" : "5f4dbd60dd078d3a0c71331e4a63a7c9",  
    "sha1" : "5f4dbd60dd078d3a0c71331e4a63a7c9",  
    "sha256" : "5f4dbd60dd078d3a0c71331e4a63a7c9",  
    "superUser" : false  
}
```

```
        "country" : "Switzerland",
        "postcode" : 9476,
        "coordinates" : {
            "latitude" : "37.5062",
            "longitude" : "-115.1853"
        },
        "timezone" : {
            "offset" : "+9 :00",
            "description" : "Tokyo, Seoul, Osaka, Sapporo, Yakutsk"
        }
    },
    "email" : "blaise.adam@example.com",
    "login" : {
        "uuid" : "3f7d9b90-8858-4986-bdf3-075df560ce31",
        "username" : "silverfish710",
        "password" : "ripken",
        "salt" : "LHwjGA7R",
        "md5" :
        "d3ea2282a04ad2ab7a85e482ffb1a5fb",
        "sha1" :
        "e1c7928d4f738199929011bbb42b925eb5f7ee9c",
        "sha256" :
        "39e15f13ec6c01ac1fc1646e9b020bef04c5ce49e90eac5da71025eae981098a"
    },
    "dob" : {
        "date" : "1954-07-03T23:24:58.883Z",
        "age" : 67
    },
    "registered" : {
```

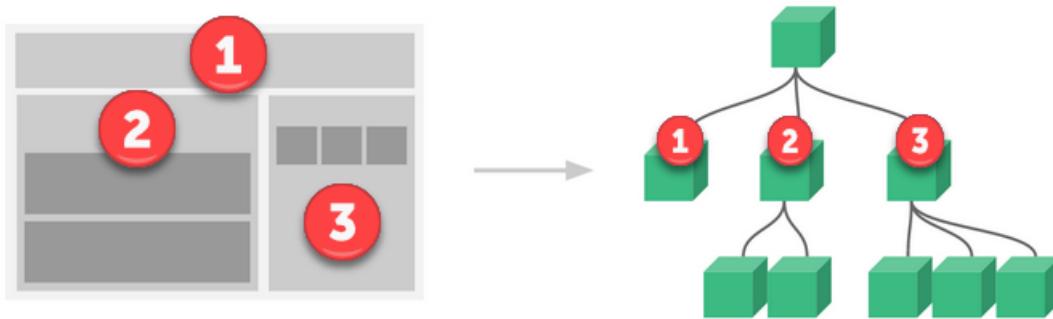
```

        "date" : "2003-01-
18T13 :41 :59.518Z",
        "age" : 18
    },
    "phone" : "079 493 00 77",
    "cell" : "079 283 03 14",
    "id" : {
        "name" : "AVS",
        "value" : "756.3512.8312.75"
    },
    "picture" : {
        "large" :
"https ://randomuser.me/api/portraits/men/29.jpg",
        "medium" :
"https ://randomuser.me/api/portraits/med/men/29.jpg",
        "thumbnail" :
"https ://randomuser.me/api/portraits/thumb/men/29.jpg"
    },
    "nat" : "CH"
}
]
}
})
;
let vm = app.mount('#app');
</script>
</body>
</html>
```

Partie 2 – Les composants Vue.js

Lorsque le code d'une application Vue.js devient trop important, il est fréquent de la diviser en plusieurs composants qui s'appellent de façon hiérarchique.

Vous pourriez par exemple définir des composants pour représenter le header (**header**), la barre latérale (**sidebar**) et la zone de contenu (**main**). Chacun de ces composants pourrait contenir d'autres composants pour représenter, par exemple, une barre de navigation, des articles ou encore des informations de copyright :



En utilisant cette approche, les composants peuvent si nécessaire être réutilisés une ou plusieurs fois dans l'application. Les composants peuvent être définis de façon globale ou locale.

Dans un premier temps, voyons comment créer un composant global :

```
const vm = Vue.createApp({  
  ...  
});  
  
vm.component('nom', {  
  // Options du composant  
})  
  
vm.mount('#app');
```

Où **nom** est le nom du composant.

Une fois le composant nom défini, vous pouvez l'utiliser dans le scope de l'application Vue avec la balise **<nom></nom>**.

Un premier composant global

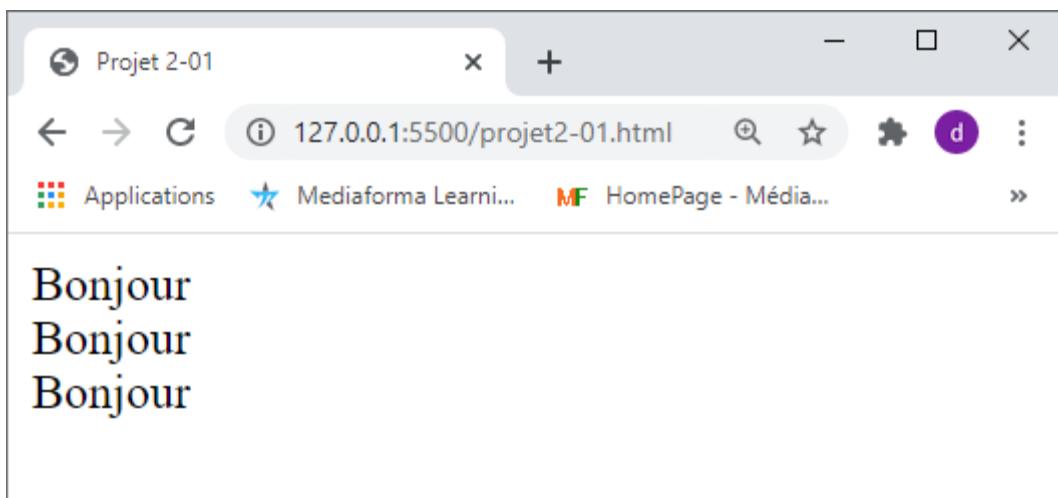
Votre premier composant global affichera simplement le mot "**Bonjour**". Il aura pour nom **bonjour**.

Voici le code à utiliser (projet2-01.html) :

```
<div id="app">
  <bonjour></bonjour>
  <bonjour></bonjour>
  <bonjour></bonjour>
</div>
<script>
  const app = Vue.createApp({
    });
    app.component('bonjour', {
      template : '<div>Bonjour</div>'
    })
    let vm = app.mount('#app');
</script>
```

Le template est une simple chaîne qui contient le code HTML qui doit être affiché par le composant.

Voici ce que vous devriez obtenir :



```
<bonjour></bonjour>
```

ou :

```
<bonjour />
```

Un premier composant local

La définition des composants de façon globale n'est pas toujours la meilleure solution. Par exemple, si vous utilisez le *module bundler* Webpack, les composants définis de façon globale sont toujours inclus dans l'application, même s'ils ne sont pas utilisés. Cela augmente inutilement la quantité de code JavaScript à télécharger. Pour ne plus avoir ce problème, vous pouvez définir les composants localement, dans la section `components` de l'application :

```
const app = Vue.createApp({  
  components : {  
    'component-a' : ComponentA,  
    'component-b' : ComponentB  
  }  
})
```

Les composants seront alors définis comme de simples objets JavaScript :

```
const ComponentA = {  
  /* ... */  
}  
  
const ComponentB = {  
  /* ... */  
}
```

Attention : les composants définis localement ne sont pas disponibles dans les sous-composants. Si, par exemple, vous voulez que le composant A soit disponible dans le composant B, vous devrez écrire quelque chose comme ceci :

```
const ComponentA = {
```

```

/* ... */

}

const ComponentB = {
  components : {
    'component-a' : ComponentA
  }
  // ...
}

```

Voyons un exemple concret. Votre premier composant local affichera "**Le composant local dit bonjour**". Il aura pour nom **BonjourLoc**.

Le composant local **BonjourLocal** est un objet JavaScript qui contient au minimum une propriété **template**. Ce composant local est référencé dans la section **components** de l'application. La propriété est le nom du composant et la valeur est le nom de l'objet associé.

Pour utiliser le composant local, il suffit de spécifier sa balise dans le scope du composant global où il est défini (projet2-02.html) :

```

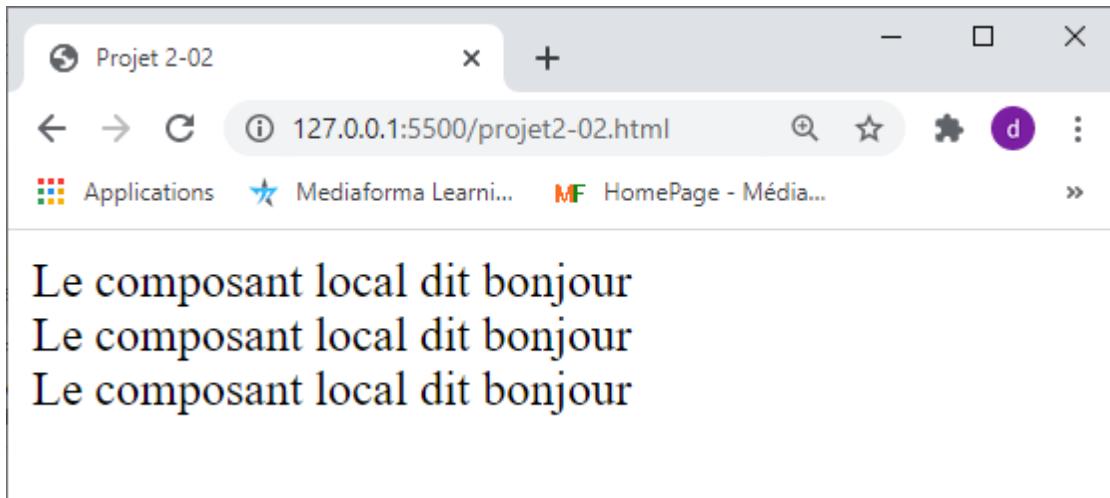
<div id="app">
  <bonjour-loc></bonjour-loc>
  <bonjour-loc></bonjour-loc>
  <bonjour-loc></bonjour-loc>
  <bonjour-loc />
</div>
<script>
  const BonjourLocal = {
    template : '<div>Le composant local dit bonjour</div>'
  }
  const app = Vue.createApp({
    components : {
      'bonjour-loc' : BonjourLocal
    }
  })
  app.mount('#app')
</script>

```

```
        }

    })
let vm = app.mount('#app');
</script>
```

Voici le rendu de ce code :



Le composant local <bonjour> peut s'écrire indifféremment :

<bonjour-loc></bonjour-loc>

ou :

<bonjour-loc />

Ajouter des données dans un composant

Vous pouvez ajouter des données dans un composant global ou local. Pour cela, ajoutez la propriété **data** dans le modèle de ce composant.

Attention : cette propriété est une fonction qui retourne un objet JSON qui contient la ou les données.

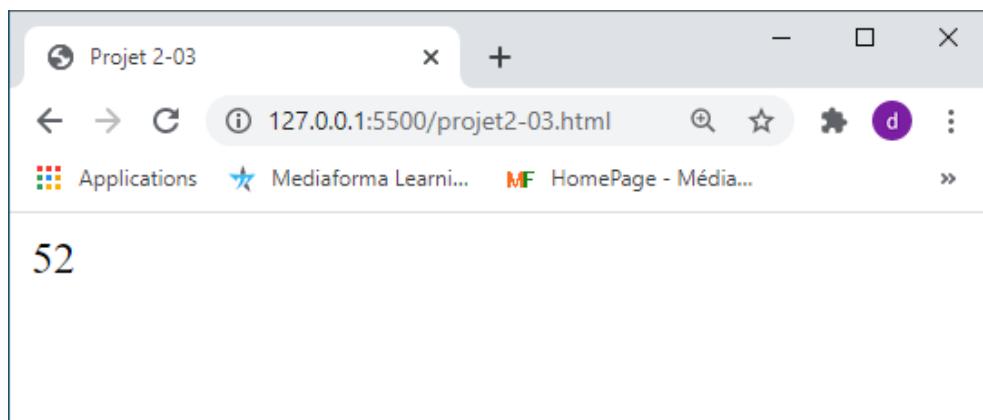
Voici par exemple un composant local <rnd-loc></rnd-loc> qui affiche un nombre aléatoire compris entre **1** et **100** (projet2-03.html) :

```
<div id="app">
    <rnd-loc></rnd-loc>
</div>
```

```

<script>
    const RndLocal = {
        data() {
            return {
                nombre : Math.floor(Math.random() * 100 + 1)
            }
        },
        template : '<div>{{nombre}}</div>'
    }
    const app = Vue.createApp({
        components : {
            'rnd-loc' : RndLocal
        }
    })
    let vm = app.mount('#app');
</script>

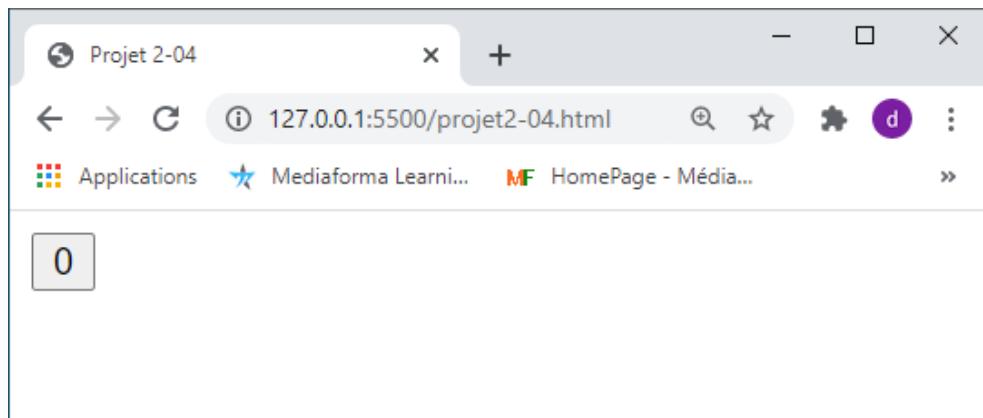
```



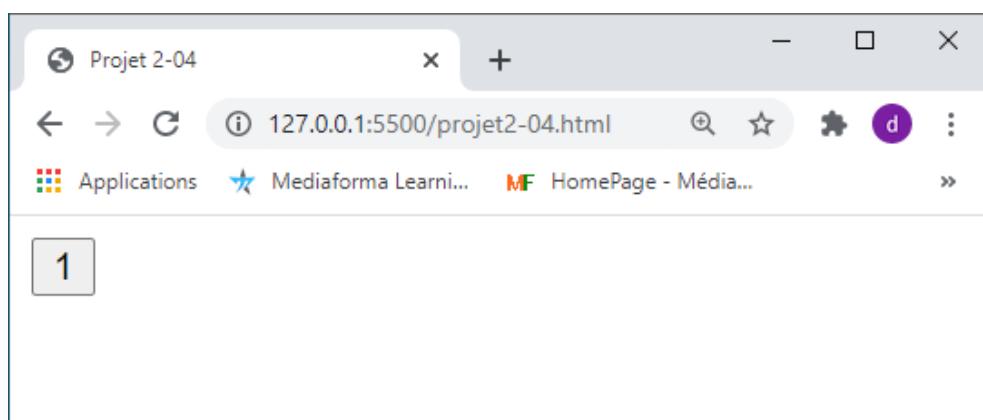
Challenge

Définissez le composant global **compteur** qui affiche un compteur de clic dans un bouton. Chaque fois que l'utilisateur clique sur le bouton, le compteur est incrémenté d'un.

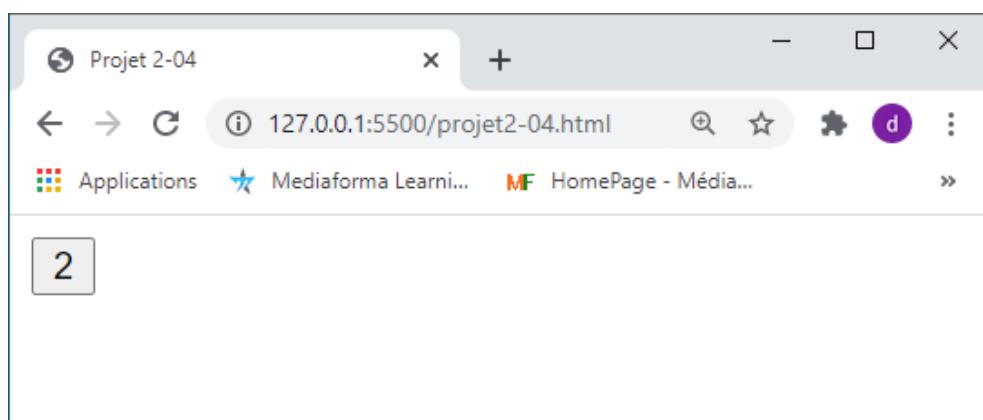
Au lancement :



Après un clic :



Après deux clics :



Solution (projet2-04.html) :

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

```
const app = Vue.createApp({  
});  
app.component('compteur', {  
    data() {  
        return {  
            nbClics : 0  
        }  
    },  
    template : '<button @click="nbClics++">{ nbClics }</button>'  
})  
let vm = app.mount('#app');  
</script>
```

Challenge

Définissez le composant local **cpt-loc** équivalent au composant global **compteur** que vous venez de définir.

Solution (projet2-05.html) :

```
<div id="app">
    <cpt-loc></cpt-loc>
</div>
<script>
    const CptLocal = {
        data() {
            return {
                nbClics : 0
            }
        },
        template : '<button @click="nbClics++">{ nbClics }</button>'
    }
    const app = Vue.createApp({
        components : {
            'cpt-loc' : CptLocal
        }
    })
    let vm = app.mount('#app');
</script>
```

Ajouter des méthodes dans un composant

Vous pouvez ajouter des méthodes dans un composant global ou local.

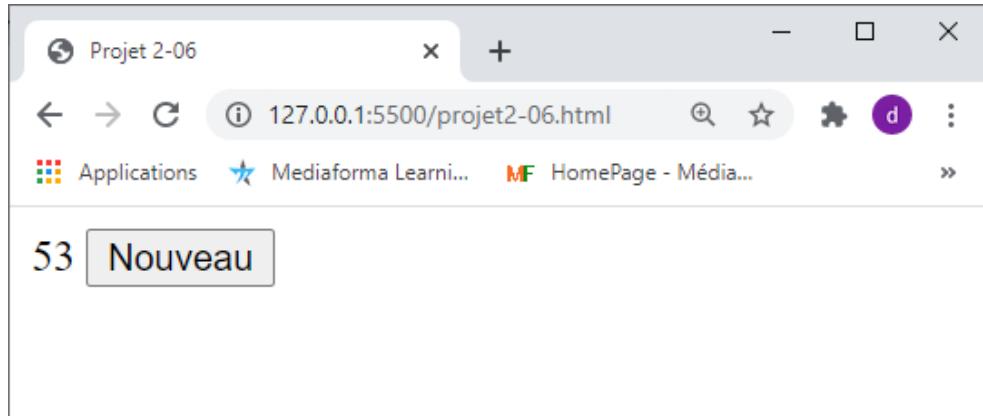
Pour cela, insérez la propriété **methods** dans le composant et définissez les méthodes du composant , comme vous le feriez dans une instance de Vue.

Challenge

Reprenez le code du composant <rnd-loc> (projet2-03.html).

Ajoutez un bouton dans ce composant.

Lorsque l'utilisateur clique dessus, affichez un autre nombre aléatoire compris entre **1** et **100**.



Solution (projet2-06.html) :

```
<div id="app">
    <rnd-loc></rnd-loc>
</div>
<script>
    const RndLocal = {
        data() {
            return {
                nombre : Math.floor(Math.random() * 100 + 1)
            }
        },
        methods : {
            nouveau() {
                this.nombre = Math.floor(Math.random() * 100 + 1)
            }
        },
        template : '<div>{{nombre}} <button
@click="nouveau">Nouveau</button></div>'
    }
    const app = Vue.createApp({
        components : {
            'rnd-loc' : RndLocal
        }
    })
    app.mount('#app')
</script>
```

```
        }
    })
let vm = app.mount('#app');
</script>
```

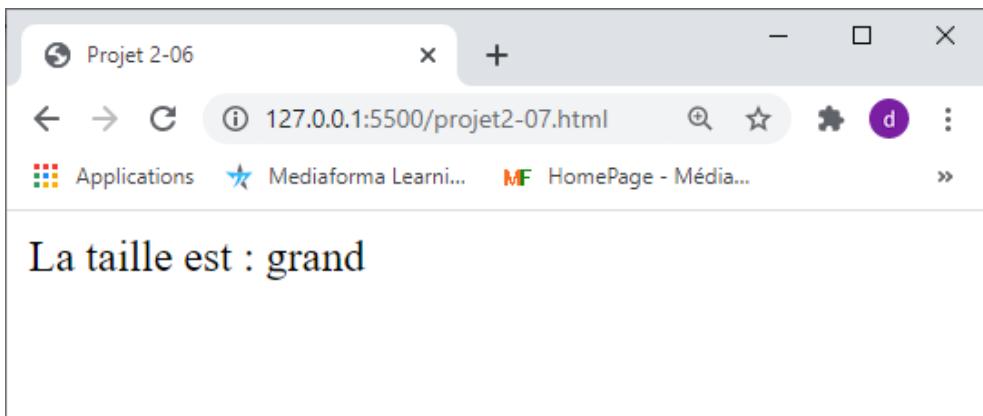
Passer des données aux composants enfants avec des props

Si vous définissez des attributs dans la partie HTML d'un composant (ici **param**), ils peuvent être récupérés dans le composant *via* la propriété **props** et utilisés dans le template avec des interpolations (projet2-07.html) :

```
<div id="app">
    <taille param="grand"></taille>
</div>
<script>
    const app = Vue.createApp({
        });
    app.component('taille', {
        template : '<div>La taille est : {{param}}</div>',
        props : ['param']
    })
    let vm = app.mount('#app');
</script>
```

Dans leur version la plus simple, les **props** sont listées dans un tableau sous la forme de chaînes de caractères.

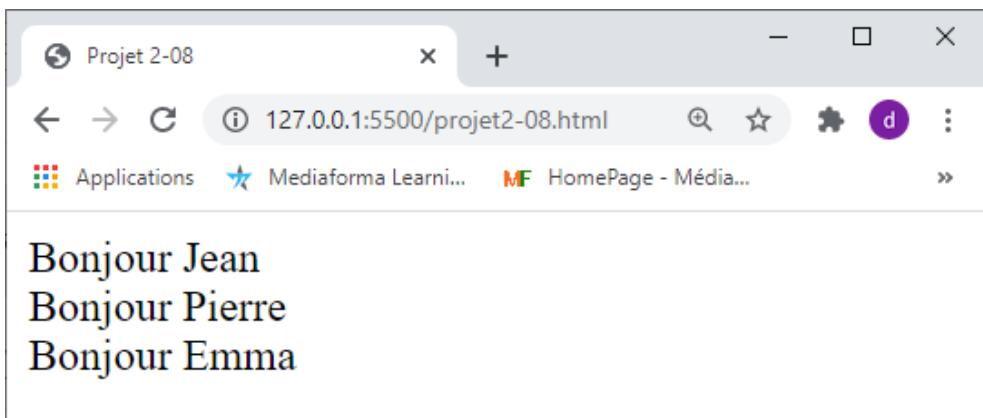
Voici ce que donne ce code :



Challenge

Définissez le composant local **bonjour** et passez-lui l'attribut **prenom**.

Invoquant trois fois ce composant pour obtenir ce résultat :



Solution (projet2-08.html) :

Voici le code du composant local **bonjour** avec la prop **prenom** :

```
<div id="app">
    <bonjour prenom="Jean"></bonjour>
    <bonjour prenom="Pierre"></bonjour>
    <bonjour prenom="Emma"></bonjour>
</div>
<script>
    const Bonjour = {
        template : '<div>Bonjour {{prenom}}</div>',
        props : ['prenom']
```

```
};

const app = Vue.createApp({
    components : {
        'bonjour' : Bonjour
    }
});

let vm = app.mount('#app');
</script>
```

Challenge

Définissez le composant global **bonjour** équivalent.

Solution (projet2-09.html) :

Voici le code du composant global **comp** avec la prop **texte** :

```
<div id="app">
    <bonjour prenom="Jean"></bonjour>
    <bonjour prenom="Pierre"></bonjour>
    <bonjour prenom="Emma"></bonjour>
</div>
<script>
    const app = Vue.createApp({
    });
    app.component('bonjour', {
        template : '<div>Bonjour {{prenom}}</div>',
        props : ['prenom']
    })
    let vm = app.mount('#app');
</script>
```

Accès à une prop dans le modèle du composant

Les props définies dans un composant sont accessibles dans le modèle du composant *via* l'objet **this** (projet2-10.html) :

```
<div id="app">
    <maj texte="Texte à mettre en majuscules"></maj>
</div>
<script>
    const app = Vue.createApp({
        });
        app.component('maj', {
            template : '<div>{{enMaj}}</div>',
            props : ['texte'],
            data() {
                return {
                    enMaj : this.texte.toUpperCase()
                }
            }
        })
        let vm = app.mount('#app');
</script>
```

Voici le rendu :



Challenge

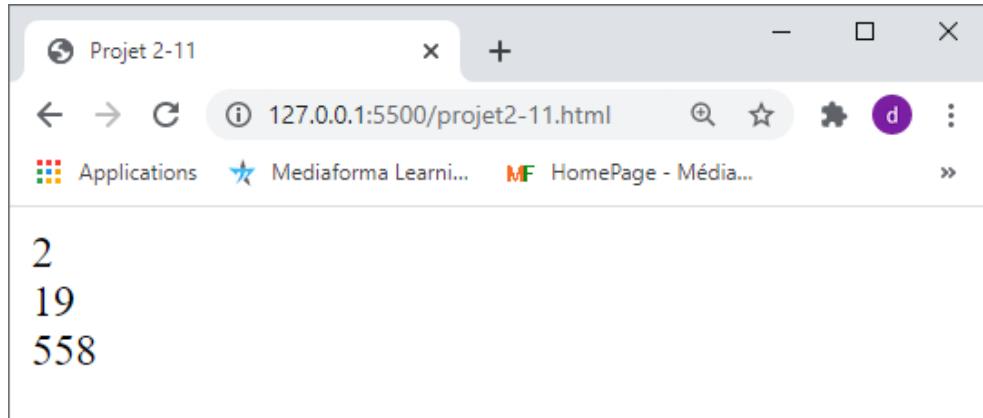
Vous allez travailler sur le composant local **<rnd-loc>** défini précédemment (projet2-03.html). Définissez la prop **max** dans le code du composant **<rnd-loc>**.

Cette prop indiquera la borne supérieure du tirage aléatoire.

Par exemple, ce code affichera un entier aléatoire compris entre **1** et **100** :

```
<rnd-loc max="100"></rnd-loc>
```

Invoquez le composant **<rnd-loc>** à trois reprises en fixant la limite supérieure (respectivement) à **10**, **100**, puis **1000**. Voici un exemple d'exécution :



Solution (projet2-11.html) :

```
<div id="app">
    <rnd-loc max="10"></rnd-loc>
    <rnd-loc max="100"></rnd-loc>
    <rnd-loc max="1000"></rnd-loc>
</div>
<script>
    const RndLocal = {
        props : ['max'],
        data() {
            return {
                nombre : Math.floor(Math.random() * this.max + 1)
            }
        },
        template : '<div>{{nombre}}</div>'
    }
    const app = Vue.createApp({
        components : {
            'rnd-loc' : RndLocal
        }
    })
    app.mount('#app')
</script>
```

```
})
let vm = app.mount('#app');
</script>
```

Remarquez comment la prop **max** est utilisée dans le calcul du nombre aléatoire

Chacune des instances du composant enfant est indépendante des autres car la donnée transmise à la prop n'est pas bindée au composant.

Passer des données de l'application aux composants enfants

Si on définit des données dans une instance de Vue, les composants enfants ne peuvent pas accéder à ces données. Ici, `message` est inaccessible au composant enfant global **taille** (`projet2-12.html`) :

```
<div id="app">
    <taille></taille>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                message : 'Message du parent au composant taille'
            }
        }
    });
    app.component('taille', {
        template : '<div>{{message}}</div>'
    })
    let vm = app.mount('#app');
</script>
```

Ce code produit un warning à l'exécution : "la propriété `message` n'est pas définie dans l'instance du composant".

Pour que le composant taille puisse accéder à la propriété **message** définie dans l'application, vous devez définir :

- **Un attribut** dans la partie HTML du composant pour se binder à message
- **Une prop** dans le composant pour récupérer la valeur de l'attribut

Voici ce que vous devez faire :

```
<div id="app">
  <taille :mess="message"></taille>
</div>
<script>
  const app = Vue.createApp({
    data() {
      return {
        message: 'Message du parent au composant taille'
      }
    }
  });
  app.component('taille', {
    template: <div>{{mess}}</div>,
    props: ['mess']
  })
  let vm = app.mount('#app');
</script>
```

La directive **v-bind** : (ici son alias :) établit un lien avec la data message de l'application. Cette data est affectée à l'attribut **mess** du composant, récupérée dans la prop **mess** du composant et utilisée dans le modèle du composant avec une interpolation.

Voici le code (projet2-12.html) :

```
<div id="app">
  <taille :mess="message"></taille>
</div>
<script>
  const app = Vue.createApp({
```

```

        data() {
            return {
                message : 'Message du parent au composant taille'
            }
        }
    });

app.component('taille', {
    template : '<div>{{mess}}</div>',
    props : ['mess']
})
let vm = app.mount('#app');
</script>

```

Et voici le rendu :



Custom-events - Envoi de messages au parent par un composant avec \$emit

La fonction **\$emit('ev')** déclenche un évènement **ev** sur l'instance courante d'un composant. Le composant réagit à cet évènement avec un **v-on** et il exécute une méthode du parent.

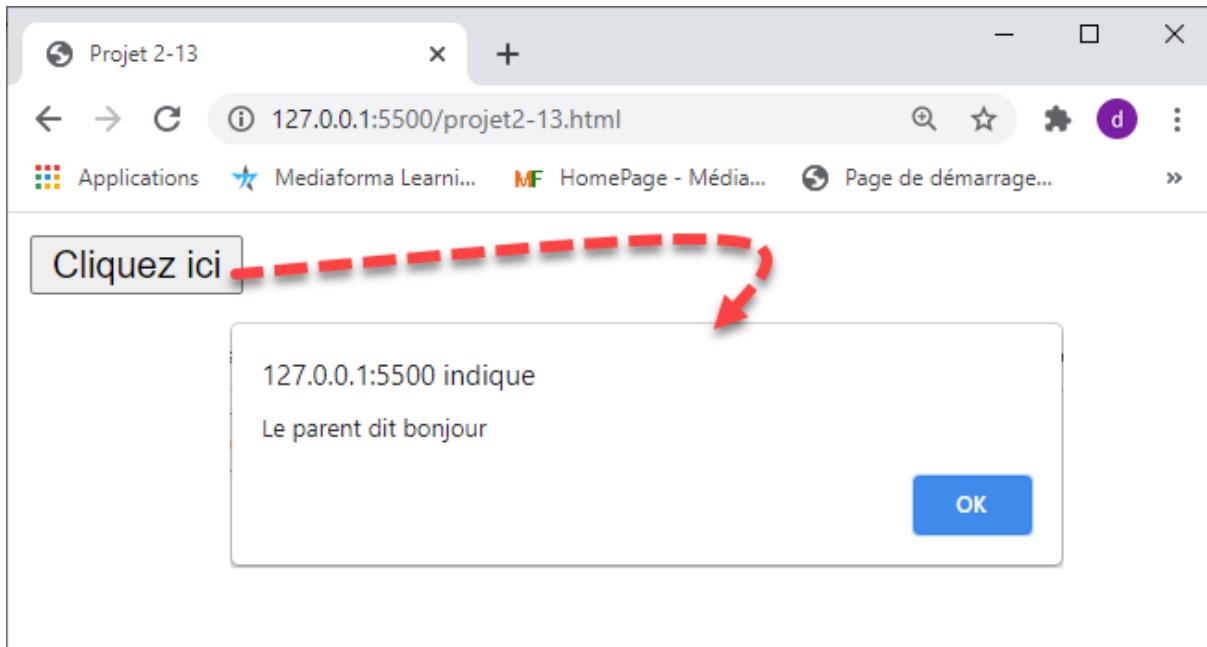
Voici un exemple de code (projet2-13.html) :

```

<div id="app">
    <bienvenue @coucou="disBonjour"></bienvenue>
</div>
<script>
    const app = Vue.createApp({
        methods : {
            disBonjour() {
                alert('Le parent dit bonjour');
            }
        }
    });
    app.component('bienvenue', {
        template : '<button @click="$emit('coucou')">Cliquez ici</button>'
    })
    let vm = app.mount('#app');
</script>

```

Quand le bouton est cliqué dans le composant, l'évènement **coucou** est émis. Il est capturé dans le code HTML du composant qui exécute la fonction **disBonjour()** définie dans le parent, qui affiche une boîte d'alerte.

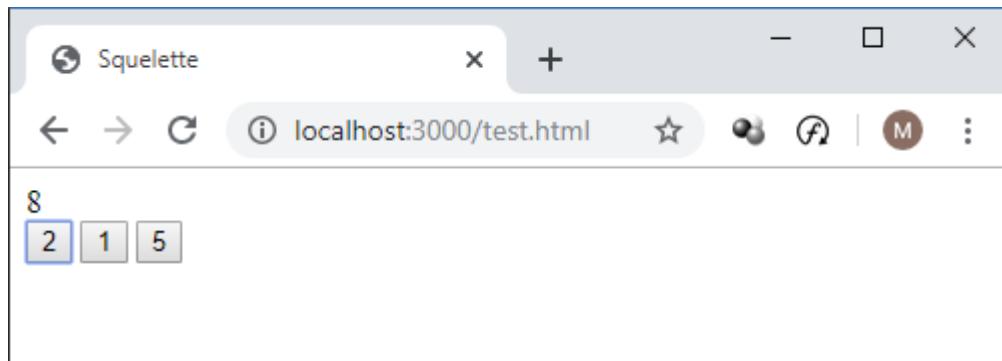


Challenge

Définissez un composant **compteur** qui contient un bouton. Ce bouton affiche **0** au lancement de l'application. Lorsqu'on clique dessus, il s'incrémente d'un et affiche **1**. Lorsqu'on clique à nouveau dessus, il s'incrémente d'un et affiche **2**. Ainsi de suite...

Ajoutez trois composants **compteur** dans la Vue de l'application.

En utilisant la fonction **\$emit()**, faites en sorte que la somme des trois compteurs s'affiche dans la Vue de l'application.



Solution (projet2-14.html) :

```
<div id="app">
    {{total}}
    <compteur @incremente="plusUn"></compteur>
    <compteur @incremente="plusUn"></compteur>
    <compteur @incremente="plusUn"></compteur>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                total : 0
            }
        },
        methods : {
            plusUn() {
                this.total++;
            }
        }
    });

```

```

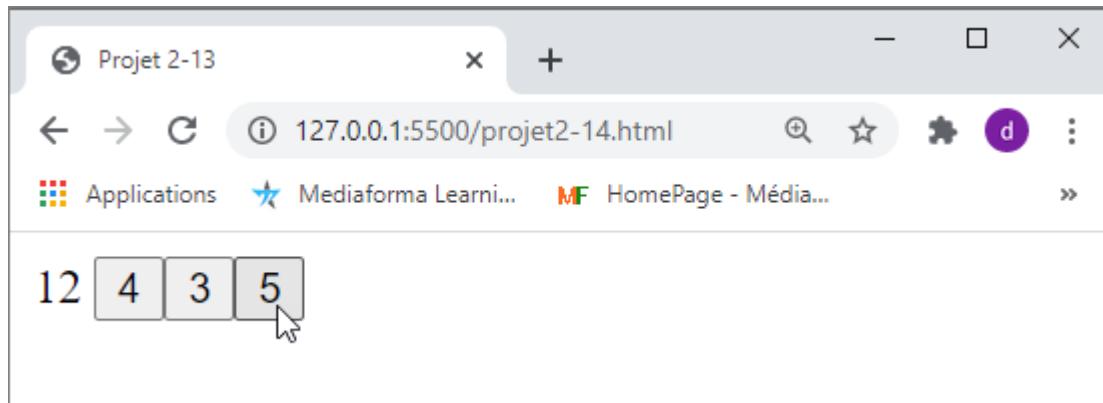
app.component('compteur', {
  data() {
    return {
      valeur : 0
    }
  },
  methods : {
    ajouteUn() {
      this.valeur++;
      this.$emit('incremente');
    }
  },
  template : '<button @click="ajouteUn">{{valeur}}</button>'
})
let vm = app.mount('#app');
</script>

```

Au clic sur un des boutons, la méthode d'instance **ajouteUn()** est exécutée. Elle incrémente la variable d'instance **valeur** (la valeur du compteur) et émet l'évènement **incremente** avec **\$emit()**.

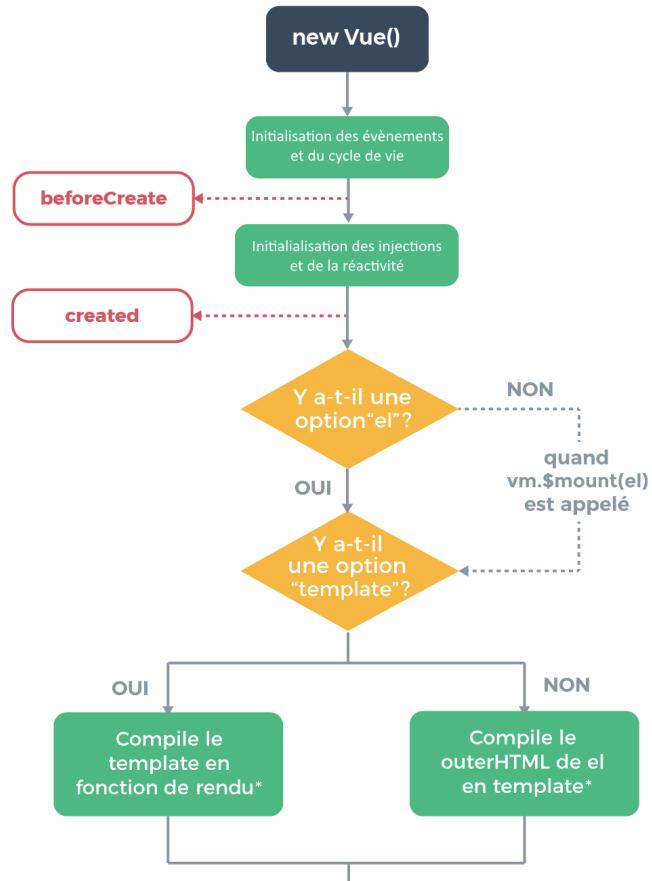
Cet évènement est capturé dans le HTML avec **@incremente** et la fonction **plusUn()** est exécutée dans le parent. Cette fonction incrémente la variable **total** (la valeur totale) qui est affichée dans l'application.

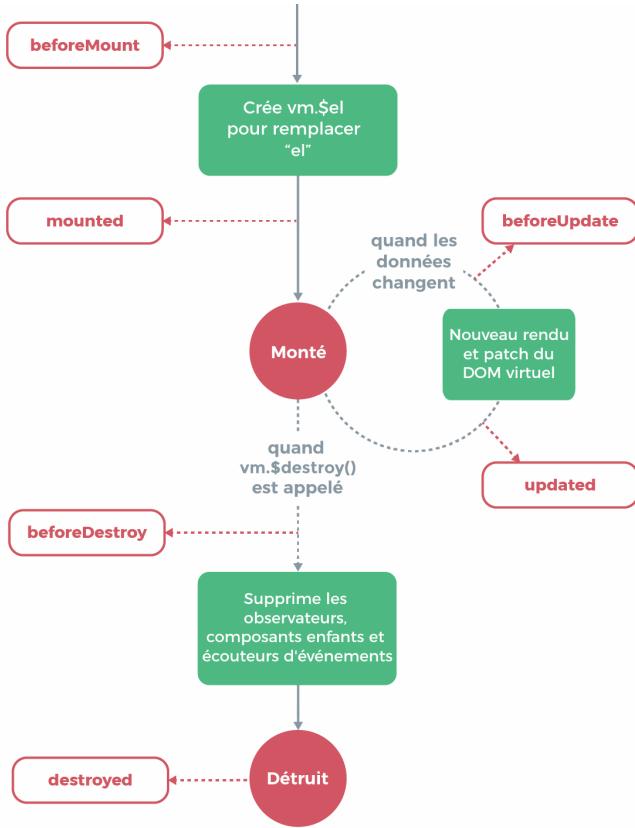
Voici Un exemple de rendu. Ici, on voit bien que les trois compteurs sont indépendants et que le total des trois compteurs est affiché tout à gauche :



Lifecycle Hooks

Les **Lifecycle Hooks** permettent d'exécuter du code à un moment précis du cycle de vie d'un composant : lors de l'initialisation, de la mise à jour ou de la destruction du composant par exemple.





Nous allons nous intéresser au hook **created**, qui est appelé après la mise à disposition des données définies dans la propriété **data** du modèle, mais avant que le composant ne soit attaché au DOM.

Pour utiliser un hook, il suffit de le mettre en place dans la vue avec un code de ce type (ici pour le hook **created**) :

```

const app = Vue.createApp({
  data() {
    return {
    }
  },
  methods : {
  },
  created() {
  }
}) ;
let vm = app.mount('#app') ;
  
```

Nous allons invoquer l'API REST **randomuser.me**.

Essayez cette URL pour récupérer au format JSON trois utilisateurs tirés aléatoirement :

<https://randomuser.me/api/?results=3>

Voici le résultat dans Firefox :

The screenshot shows the Firefox browser window with the URL <https://randomuser.me/api/?results=3> in the address bar. The page content is displayed in the JSON viewer tool, which is part of the developer tools. The JSON structure is as follows:

```
results:
  0:
    gender: "male"
    name:
      title: "Mr"
      first: "Faruk"
      last: "Seidler"
    location:
      street:
        number: 3019
        name: "Erlenweg"
        city: "Greiz"
        state: "Niedersachsen"
        country: "Germany"
        postcode: 41717
      coordinates:
        latitude: "-38.0269"
        longitude: "-63.2370"
      timezone:
        offset: "+1:00"
        description: "Brussels, Copenhagen, Madrid, Paris"
    email: "faruk.seidler@example.com"
    login:
      uuid: "4d744585-a696-4f28-bab3-9e0e43fd36e8"
      username: "silverduck630"
      password: "peterson"
```

Comme vous pouvez le voir, de nombreuses données sont disponibles pour chaque personne.

Pour obtenir les données renvoyées par cette API, vous allez utiliser la bibliothèque **Axios** dont voici le CDN :

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

Axios permet de faire des requêtes XMLHttpRequest. Il est basé sur les promises. Vous utiliserez donc la fonction **then()** pour récupérer la réponse :

```
axios.get(adresse).then(function(reponse) {
```

```
    console.log(reponse);  
});
```

Où **adresse** est l'adresse qui délivre les données.

Ici : <https://randomuser.me/api/?results=3>

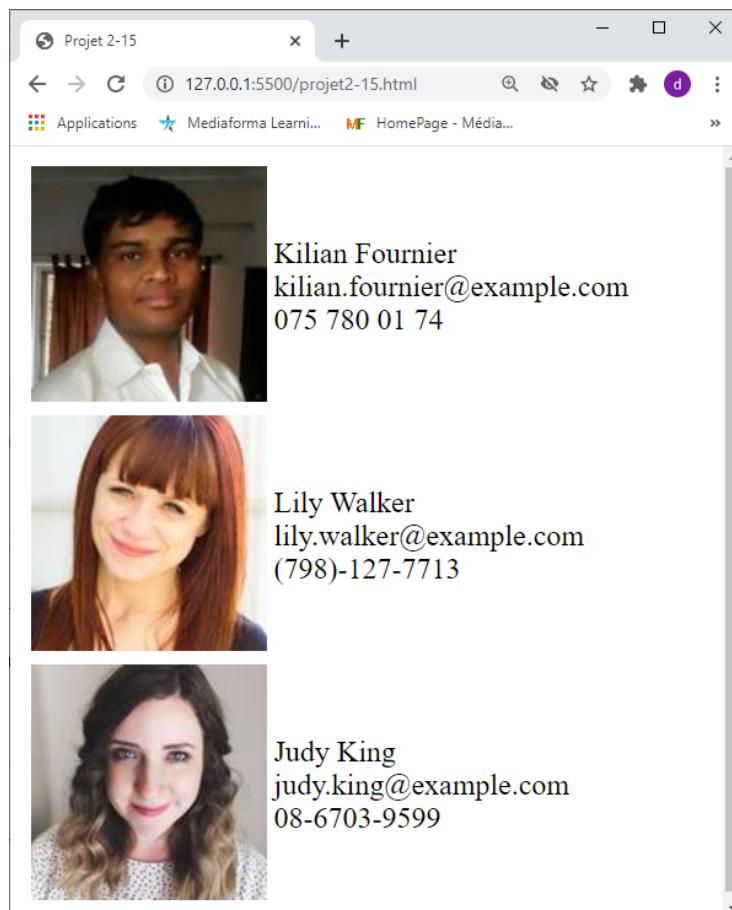
La documentation sur Axios se trouve ici : <https://github.com/axios/axios>

Challenge

Définissez une nouvelle application.

Dans son hook **created**, utilisez **Axios** pour récupérer les données de trois utilisateurs tirés aléatoirement sur le site **randomuser.me**.

Affichez-les pour obtenir quelque chose comme ceci :



Solution (projet2-15.html) :

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

```

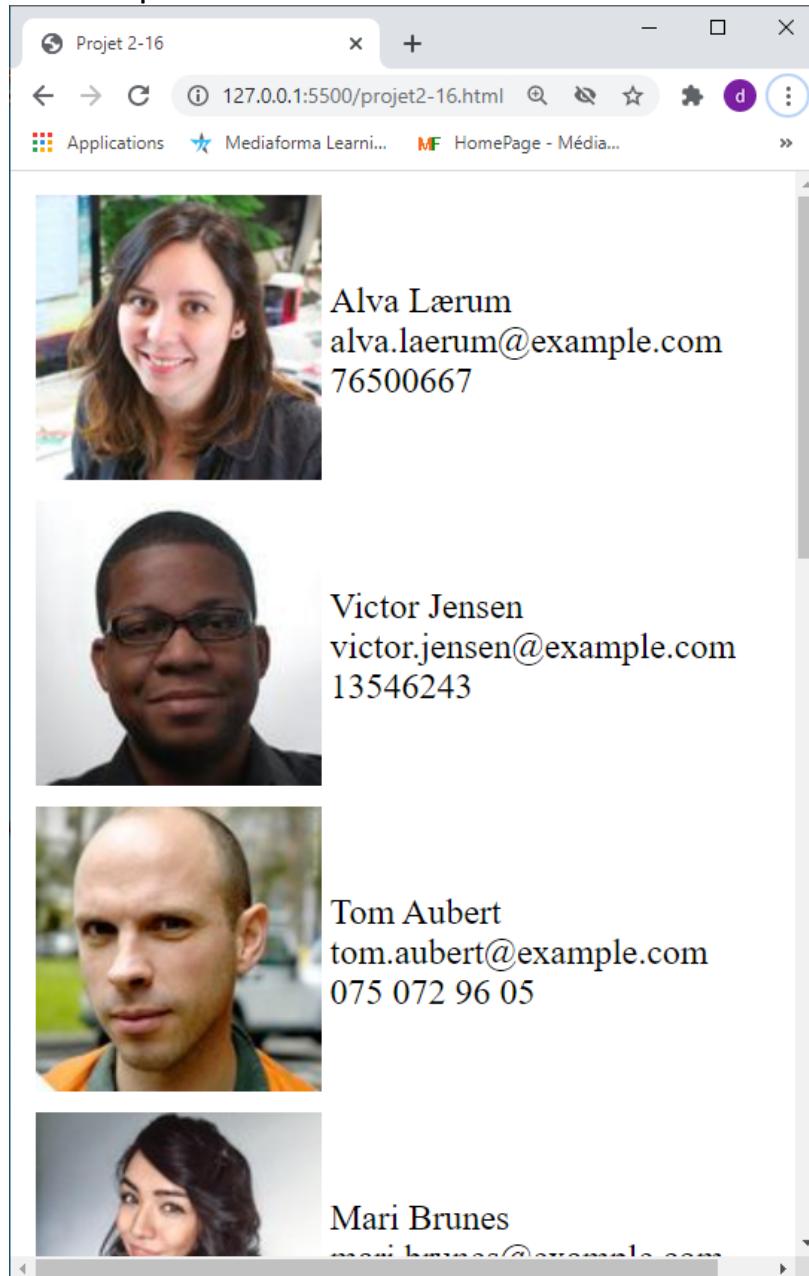
<table>
  <tr v-for="user in users">
    <td> </td>
    <td>
      {{user.name.first}} {{user.name.last}}<br>
      {{user.email}}<br>
      {{user.phone}}
    </td>
  </tr>
</table>
</div>
<script>
  const app = Vue.createApp({
    data() {
      return {
        users : []
      }
    },
    methods : {
    },
    created() {
      axios.get('https ://randomuser.me/api/?results=3').then(function(reponse) {
        vm.users = reponse.data.results;
      });
    }
  );
  let vm = app.mount('#app');
</script>

```

Organiser une application en hiérarchie de composants

En vous basant sur le code précédent, définissez :

- Une application qui récupère **10 utilisateurs** sur **randomuser.me** via **Axios** dans le hook **created**.
- Un composant local appelé **un** qui affiche la photo, le nom, le mail et le téléphone d'un utilisateur.
- Un composant local appelé **tous** qui affiche les dix utilisateurs via le composant **un** et une boucle **v-for**.



Solution avec deux composants locaux **un** et **tous** (projet2-16.html) :

```
<div id="app">
  <tous v-bind :personnes="users"></tous>
```

```

</div>
<script>
    const Un = {
        template : '<tr> \
            <td> \
                 \
            </td> \
            <td> \
                {{unePersonne.name.first}} \
                {{unePersonne.name.last}}<br> \
                {{unePersonne.email}}<br> \
                {{unePersonne.phone}} \
            </td> \
        </tr>',
        props : ['unePersonne']
    }
    const Tous = {
        template : '<table v-for="personne in personnes"> \
            <un v-bind :unePersonne="personne"></un> \
        </table>',
        components : {
            'un' : Un
        },
        props : ['personnes']
    }
    const app = Vue.createApp({
        data() {
            return {
                users : []
            }
        },
        created() {
            axios.get('https ://randomuser.me/api/?results=10').then(function (reponse) {
                vm.users = reponse.data.results;
            });
        },
    },

```

```

components : {
  'tous' : Tous
}
});

let vm = app.mount('#app');
</script>

```

Rien de bien compliqué :

- L'application récupère les 10 utilisateurs et les stocke dans le tableau **users**. Elle utilise le composant **Tous** sous la forme de la balise **<tous></tous>**. La directive **v-bind:personnes="users"** de la balise **<tous>** donne accès au tableau **users** sous le nom **personnes** dans le composant **Tous**.
- Le composant local **Tous** récupère le tableau personnes avec une **props**. Il utilise le composant **Un**. Son template effectue une boucle **v-for** sur les différentes personnes contenues dans le tableau **personnes**. Chaque personne est accessible sous le nom **unePersonne** dans le composant **Un** grâce à la directive **v-bind:unePersonne="personne"** dans la balise **<un>**.
- Le composant **Un** récupère chaque personne avec la props **unePersonne**. La photo est extraite de la personne et affectée à la propriété **src** de la balise ****. Le nom, le mail et le téléphone sont extraits de la personne et affichés avec des interpolations.

Challenge

Modifiez le code de l'exercice précédent pour transformer les composants **locaux un et tous** en composants **globaux**.

Solution avec deux composants globaux **un** et **tous** (projet2-17.html) :

```

<div id="app">
  <tous v-bind :personnes="users"></tous>
</div>
<script>
  const app = Vue.createApp({

```

```

data() {
    return {
        users : []
    }
},
created() {
    axios.get('https ://randomuser.me/api/?results=10').then(function (reponse) {
        vm.users = reponse.data.results;
    });
}
);
app.component('un', {
    template : '<tr> \
        <td> \
             \
        </td> \
        <td> \
            {{unePersonne.name.first}} \
            {{unePersonne.name.last}}<br> \
            {{unePersonne.email}}<br> \
            {{unePersonne.phone}} \
        </td> \
    </tr>',
    props : ['unePersonne']
});
app.component('tous', {
    template : '<table v-for="personne in personnes"> \
        <un v-bind :unePersonne="personne"></un> \
    </table>',
    props : ['personnes']
});
let vm = app.mount('#app');
</script>

```

Passer des données à un composant avec des slots

Vous savez passer des données à un composant avec des **attributs** (côté View) et des **props** (côté ViewModel).

Vous allez apprendre une autre technique pour passer des données à un composant. Cette fois-ci en utilisant la partie **innerHTML** du composant, c'est-à-dire ce qui se trouve entre la balise ouvrante et la balise fermante du composant.

Dans le jargon Vue.js, vous utiliserez des **slots**.

Il existe trois types de slots :

- **Simples**
- **Nommés**
- **Avec une portée**

Nous allons examiner ces trois types de slots dans les prochaines pages.

Slots simples

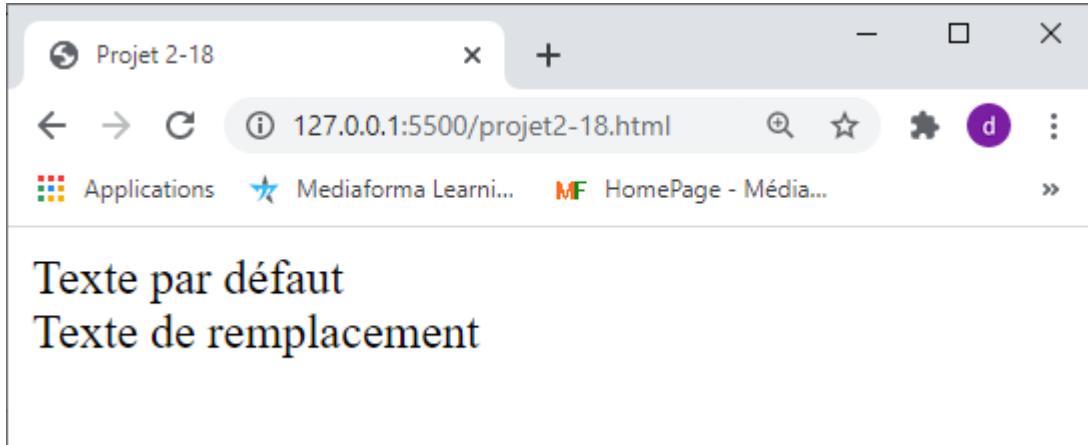
Ce qui est placé entre la balise ouvrante et la balise fermante d'un composant est ignoré lors du rendu, ... à moins d'utiliser un slot.

Essayez ce code (pojet2-18.html) :

```
<div id="app">
  <test></test>
  <test>Texte de remplacement</test>
</div>
<script>
  const app = Vue.createApp({
    });
    app.component('test', {
      template : '<div><slot>Texte par défaut</slot></div>'
    });
    let vm = app.mount('#app');
```

```
</script>
```

Voici le résultat :



Si aucun texte n'est spécifié, c'est le texte par défaut qui s'affiche, sinon, c'est le texte spécifié qui s'affiche.

Maintenant que vous savez ce qu'est un slot, allons un peu plus loin en mélangeant **props** et **slots** (projet2-19.html).

```
<div id="app">
    <art titre="Titre de l'article"><p>Texte de l'article</p></art>
</div>
<script>
    const app = Vue.createApp({
        });
        app.component('art', {
            template : '<div><h2>{{titre}}</h2><p><slot></slot></p>
</div>',
            props : ['titre']
        });
        let vm = app.mount('#app');
</script>
```

Ici, l'attribut **titre** est passé au composant **art** via une **prop** et le **innerHTML** est passé au composant via un **slot**.

Voici le rendu :



Slots nommés

Les slots peuvent être nommés grâce à la directive **v-slot** de la balise **<template>**. Reprenons l'exemple précédent, en donnant le nom **texte** au slot (projet2-20.html) :

```
<div id="app">
    <art titre="Titre de l'article">
        <template v-slot :texte>
            <p>Texte de l'article</p>
        </template>
    </art>
</div>
<script>
    const app = Vue.createApp({
    });
    app.component('art', {
        template : '<div><h2>{{titre}}</h2><p><slot name="texte">
</slot></p></div>',
        props : ['titre']
    });
    let vm = app.mount('#app');
</script>
```

La balise **<template>** a été ajoutée dans la partie HTML du composant. Le slot a été nommé avec la directive **v-slot** et récupéré via l'attribut **name**

dans le modèle.

Le rendu est le même :



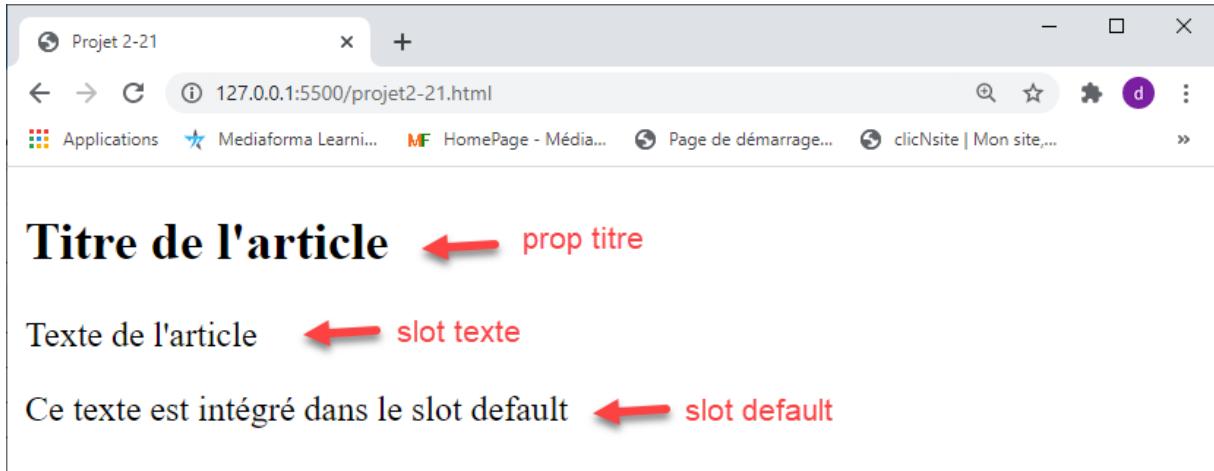
Si vous ajoutez un contenu dans un composant à l'extérieur d'un **<template>**, il est accessible dans le slot **default** du modèle (projet2-21.html) :

```
<div id="app">
    <art titre="Titre de l'article">
        <template v-slot :texte>
            <p>Texte de l'article</p>
        </template>
        <p>Ce texte est intégré dans le slot default</p>
    </art>
</div>
<script>
    const app = Vue.createApp({
    });
    app.component('art', {
        template : '<div>\n            <h2>{{titre}}</h2>\n            <p><slot name="texte"></slot></p>\n            <p><slot name="default"></slot></p>\n        </div>',
        props : ['titre']
    });

```

```
let vm = app.mount('#app');  
</script>
```

Voici le résultat :



Slots avec portée (scoped slots)

Les slots avec portée permettent de récupérer dans un slot le contenu uniquement accessible à un composant enfant.

Voyons un exemple.

Ici, les données définies dans le composant global **employes** sont uniquement accessibles dans le template de ce composant. Elles sont affichées à l'aide d'interpolations dans une boucle **v-for** (projet2-23.html) :

```
<div id="app">  
  <employes></employes>  
</div>  
<script>  
  const app = Vue.createApp({  
    } );  
  app.component('employes', {  
    data() {  
      return {  
        personnes : [  
          { nom: 'John', prenom: 'Doe', age: 40 },  
          { nom: 'Jane', prenom: 'Doe', age: 36 },  
          { nom: 'Mike', prenom: 'Smith', age: 52 },  
          { nom: 'Sarah', prenom: 'Williams', age: 29 }  
        ]  
      };  
    },  
    template: '  
      <div>  
        <h2>{{ personnes.length }} personnes </h2>  
        <table border="1">  
          <thead>  
            <tr>  
              <th>Nom</th>  
              <th>Prénom</th>  
              <th>Age</th>  
            </tr>  
          </thead>  
          <tbody>  
            <tr v-for="personne in personnes" :key="personne.nom">  
              <td>{{ personne.nom }}</td>  
              <td>{{ personne.prenom }}</td>  
              <td>{{ personne.age }}</td>  
            </tr>  
          </tbody>  
        </table>  
      </div>  
    '</template>  
  });  
  app.mount('#app');
```

```

        {
            prenom : 'Jean',
            nom : 'Segur'
        },
        {
            prenom : 'Pierre',
            nom : 'Dumanier'
        }
    ]
}

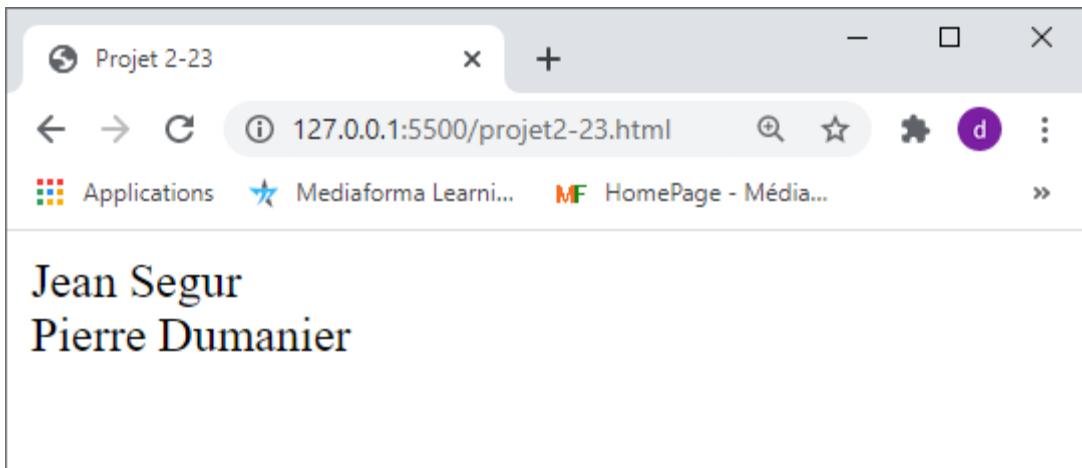
},
template : `<div v-for="personne in personnes">
    {{personne.prenom}} {{personne.nom}}
</div>`
}) ;

let vm = app.mount('#app');

</script>

```

Voici le rendu :



Supposons que vous vouliez modifier la mise en forme des données affichées dans le parent. Le tableau **personnes** n'est accessible que dans le composant. Pour le rendre accessible dans le parent, vous allez définir un

slot dans l'enfant et binder sa propriété personne (par exemple) à la personne en cours :

```
template : `<div v-for="personne in personnes">
    <slot :personne="personne"></slot>
</div>`
```

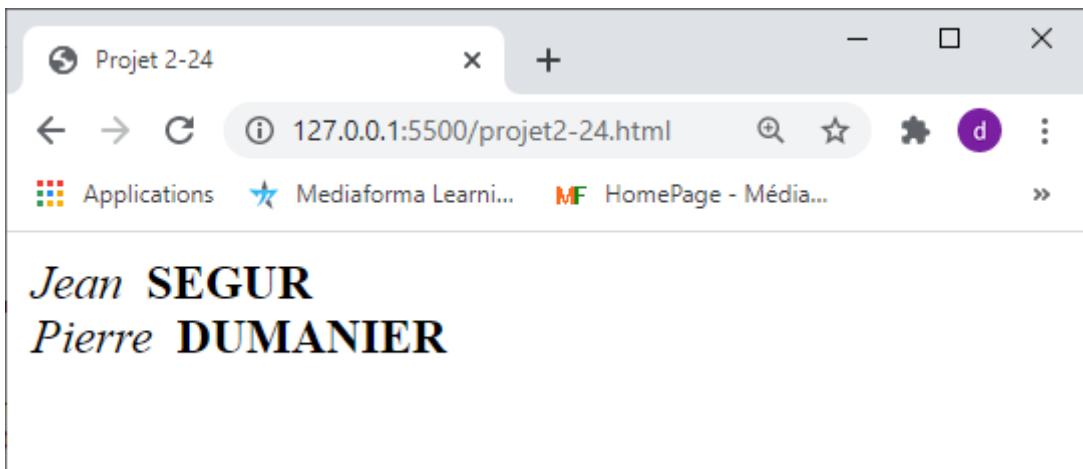
Pour récupérer les données transmises par l'enfant dans le parent, vous utiliserez un slot nommé. Si vous appelez ce slot slotProps, les propriétés prenom et nom du slot personne sont accessibles comme ceci :

```
<employees>
    <template v-slot="slotProps">
        {{slotProps.personne.prenom}}
        {{slotProps.personne.nom}}
    </template>
</employees>
```

Vous pouvez maintenant modifier l'affichage des propriétés. Ici par exemple, le prénom est affiché en italique et le nom en majuscules et en gras :

```
<employees>
    <template v-slot="slotProps">
        <span><em>{{slotProps.personne.prenom}}</em></span>&nbsp;
        <span><strong>{{slotProps.personne.nom.toUpperCase()}}</strong></span>
    </template>
</employees>
```

Voici le rendu :



Et voici le code complet (projet2-24.html) :

```
<div id="app">
  <employees>
    <template v-slot="slotProps">
      <span><em>{{ slotProps.personne.prenom }}</em>
      <span><strong>{{ slotProps.personne.nom.toUpperCase() }}</strong></span>
    </template>
  </employees>
</div>
<script>
  const app = Vue.createApp({
  });
  app.component('employees', {
    data() {
      return {
        personnes : [
          {
            prenom : 'Jean',
            nom : 'Segur'
          },
        ]
      }
    }
  })
</script>
```

```

        {
            prenom : 'Pierre',
            nom : 'Dumanier'
        }
    ]
}

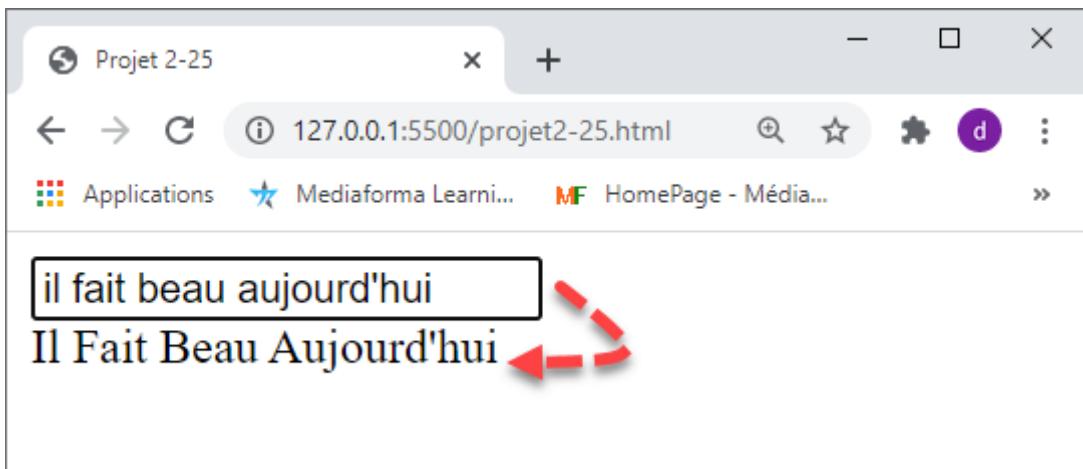
},
template : `<div v-for="personne in personnes">
    <slot :personne="personne"></slot>
</div>`
}) ;
let vm = app.mount('#app') ;
</script>

```

Filtres

Dans Vue.js 2, les filtres étaient utilisés dans les interpolations et les directives **v-bind**. Ils permettaient d'appliquer un traitement sur ces éléments. Dans Vue.js 3, les filtres ont été dépréciés. Si vous aviez l'habitude d'utiliser des filtres, remplacez-les par des appels à des méthodes ou par des propriétés calculées.

À titre d'exemple, voyons comment créer une propriété calculée qui retourne la chaîne qui lui est passée en mettant une majuscule sur la première lettre de chaque mot. Voici le résultat recherché :



Voici le code utilisé (projet2-25.html) :

```
<div id="app">
    <input type="text" v-model="texte"><br>
    {{capitalize}}
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                texte : ''
            }
        },
        computed : {
            capitalize() {
                const t = this.texte.split(' ');
                for (let i = 0; i < t.length; i++) {
                    t[i] = t[i].slice(0, 1).toUpperCase() +
t[i].slice(1);
                }
                return t.join(" ");
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

Les limites des expressions calculées

Vous pouvez ajouter des expressions calculées dans un template, à condition qu'elles soient simples. Il est en effet déconseillé d'inclure trop de logique dans vos templates pour faciliter leur maintenance. Même s'il fonctionne, ce code par exemple est trop complexe :

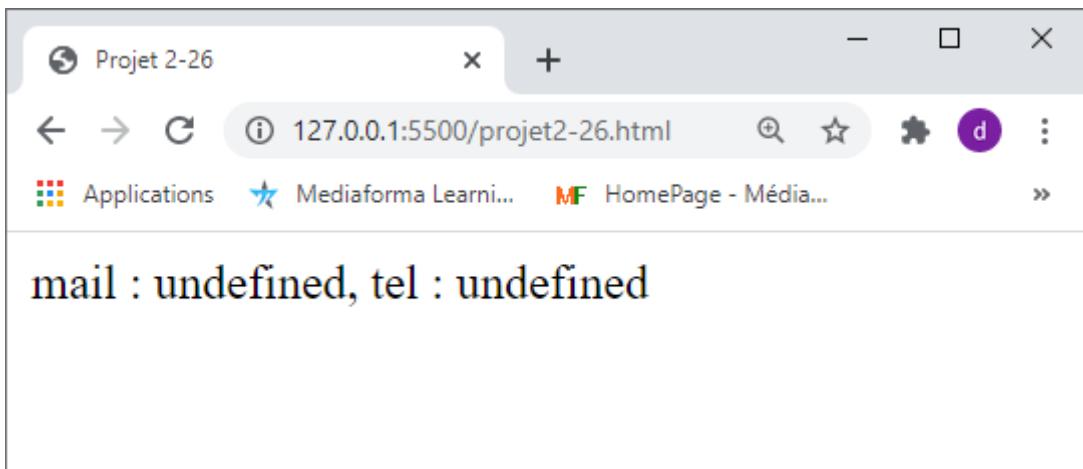
```
<div>
  {{ chaine.split('').reverse().join('') }}
</div>
```

D'autre part, vous ne pouvez pas créer une propriété calculée dans le modèle car **this** n'est pas encore disponible.

Ce code par exemple ne fonctionne pas (projet2-26.html) :

```
<div id="app">
  {{contact}}
</div>
<script>
  const app = Vue.createApp({
    data() {
      return {
        email : "test@test.com",
        phone : "0123456789",
        contact : "mail : " + this.email + ", tel : " +
this.phone
      }
    }
  });
  let vm = app.mount('#app');
</script>
```

Voici ce qui se produit à l'exécution :

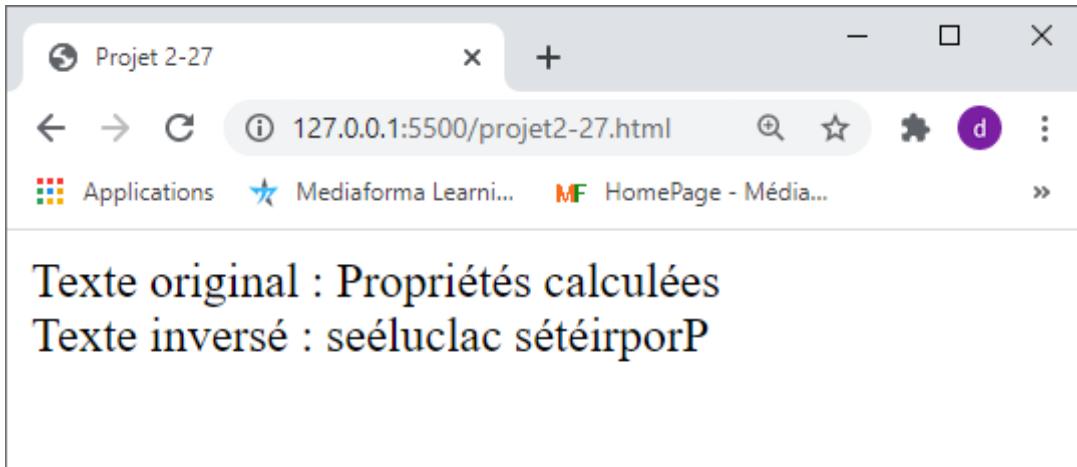


Plutôt que d'inclure du code dans un template ou dans le modèle d'une application ou d'un composant, vous définirez des propriétés calculées, dans la section **computed** du modèle de l'application ou du composant (projet2-27.html) :

```
<div id="app">  
    <div>Texte original : {{texte}}</div>  
    <div>Texte inversé : {{texteInverse}}</div>  
</div>  
<script>  
    const app = Vue.createApp({  
        data() {  
            return {  
                texte : 'Propriétés calculées'  
            }  
        },  
        computed : {  
            texteInverse() {  
                return this.texte.split('').reverse().join('');  
            }  
        }  
    });  
    let vm = app.mount('#app');
```

```
</script>
```

Voici le résultat :



Observateurs (watchers)

Dans la suite logique des propriétés calculées viennent les **observateurs**.

Ils permettent de détecter tout changement dans une propriété définie dans le modèle, et de réagir en effectuant une action systématique ou dans certains cas particuliers.

Pour créer un observateur, ajoutez la propriété **watch** dans le modèle de l'application ou du composant et définissez une fonction anonyme qui a le même nom que la propriété à observer. Ici, nous allons observer la propriété **prop**. La fonction créée dans la section **watch** est donc appelée **prop()**.

```
data() {
  return {
    prop : valeur
  }
},
watch : {
  prop(nouvValeur) {
    // action
  }
}
```

}

Où **prop** est la propriété à surveiller et **nouvValeur** représente la nouvelle valeur de cette propriété.

Vous pouvez également préciser deux paramètres pour avoir la nouvelle valeur et l'ancienne valeur :

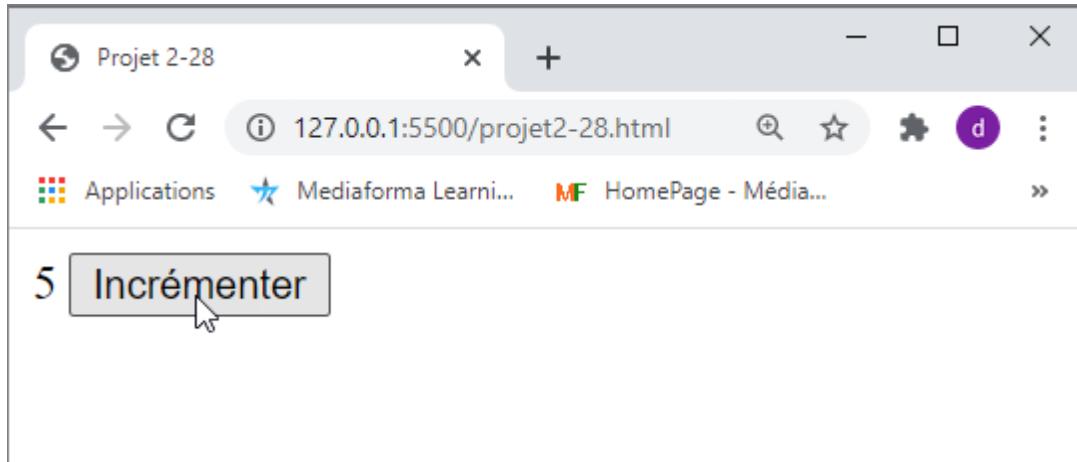
```
prop(nouvValeur, ancValeur) {  
    //action  
}
```

Challenge

Définissez :

- Une propriété qui représente un compteur dans le modèle de l'application.
- Un bouton pour incrémenter de **1** cette propriété.

Définissez un observateur sur cette propriété. Cet observateur mettra la propriété à **0** lorsqu'elle atteint la valeur **10**.



Solution (projet2-28.html) :

```
<div id="app">  
    {{compteur}}  
    <button @click="compteur++;">Incrémenter</button>  
</div>  
<script>  
    const app = Vue.createApp({
```

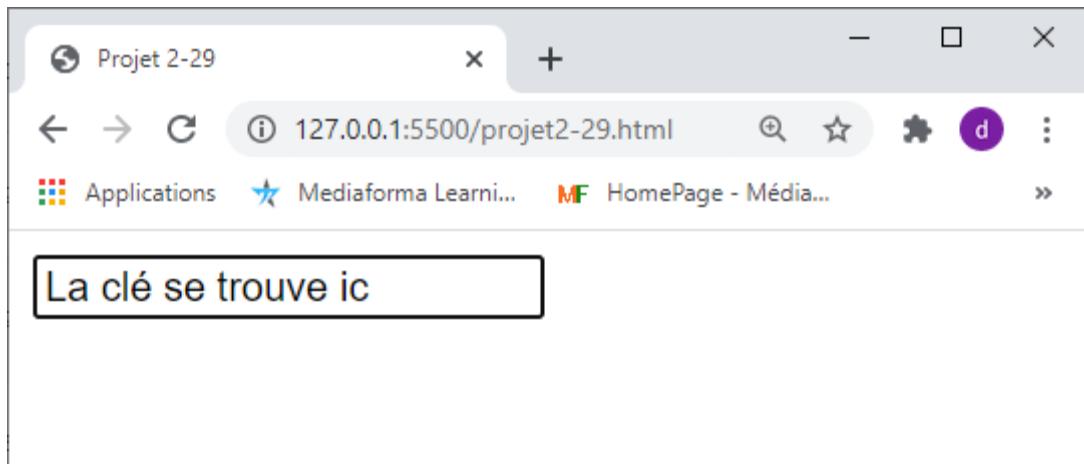
```
data() {
    return {
        compteur : 0
    }
},
watch : {
    compteur(valeur) {
        if (valeur==10) {
            this.compteur = 0;
        }
    }
})
;
let vm = app.mount('#app');
</script>
```

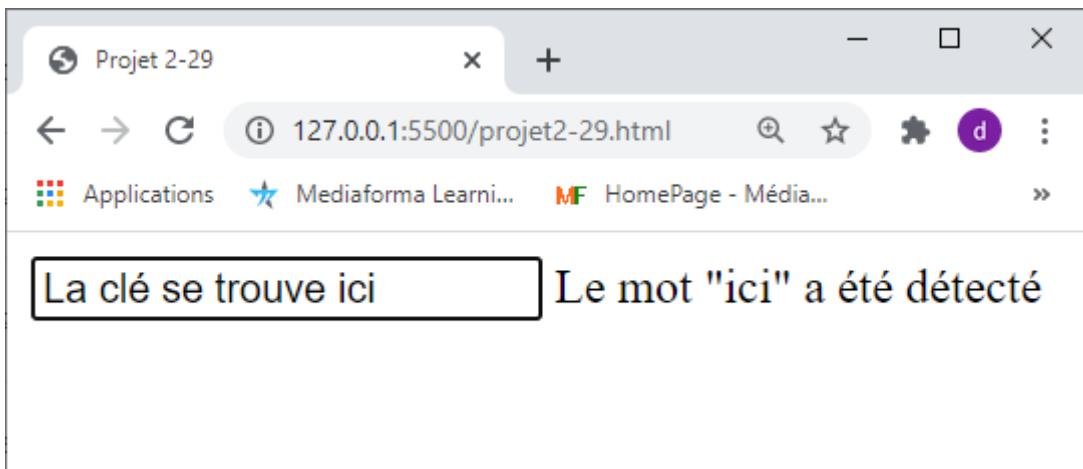
Challenge

Créez un formulaire contenant une zone de texte.

Définissez un observateur sur cette zone de texte.

A l'aide d'un observateur, affichez un message lorsque le mot "**ici**" est détecté dans la zone de texte.





Solution (projet2-29.html) :

```
<div id="app">
    <input type="text" v-model="texte">
    {{message}}
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                texte : '',
                message : ''
            }
        },
        watch : {
            texte(valeur) {
                if (valeur.indexOf('ici')!=-1) {
                    this.message = 'Le mot "ici" a été détecté';
                }
                else {
                    this.message=' ';
                }
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

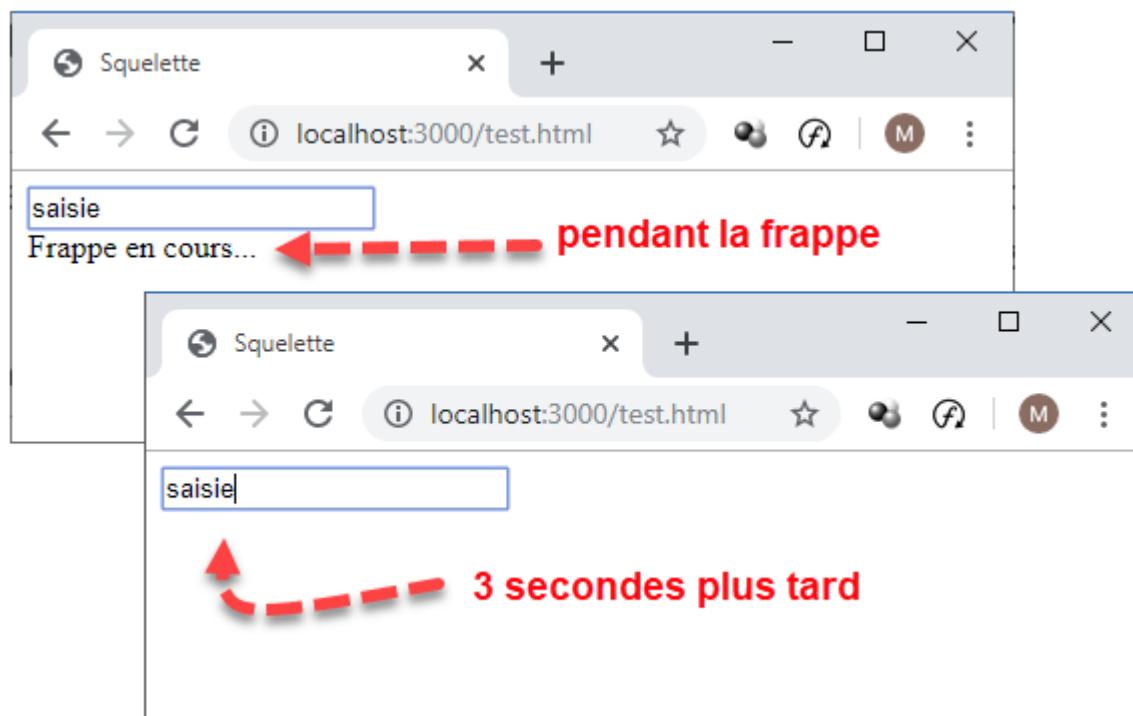
Requêtes asynchrones dans un observateur

Vous connaissez certainement la fonction JavaScript `setTimeout()` qui permet d'exécuter un code au bout d'un certain temps exprimé en millisecondes. Si vous ne maîtrisez pas cette fonction, allez jeter un œil ici : <https://bit.ly/2OZABY7>

Challenge

Je vous propose un petit challenge.

Affichez une zone de texte. Mettez en place un observateur sur cette zone de texte. Dès que l'utilisateur tape quelque chose dans la zone de texte, affichez le message "**Frappe en cours**". Faites disparaître ce message trois secondes après l'arrêt de la frappe.



Remarque

Je vous suggère d'utiliser une arrow function en paramètre de `setTimeout()`. Ainsi, `this` pointera sur l'application et non sur l'objet

window comme ce serait le cas si vous passiez une fonction anonyme à setTimeout().

Solution (projet2-30.html) :

```
<div id="app">
    <input type="text" v-model="texte">
    <span v-show="affiche">Frappe en cours ...</span>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                texte : '',
                affiche : false,
                t : null
            }
        },
        watch : {
            texte(valeur) {
                if (this.affiche) {
                    clearTimeout(t);
                }
                this.affiche = true;
                t = setTimeout(() => {
                    this.affiche = false;
                }, 3000);
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

Remarque

Etant donné que chaque touche frappée lance un nouveau timer, vous devez supprimer le timer en cours si un timer est déjà en cours. Dans le cas contraire, des effets de bord se produiront lorsque plusieurs touches seront tapées.

Animations et transitions

Vue peut appliquer des effets de transition lorsque des éléments sont insérés, mis à jour ou supprimés du DOM. Vous pouvez :

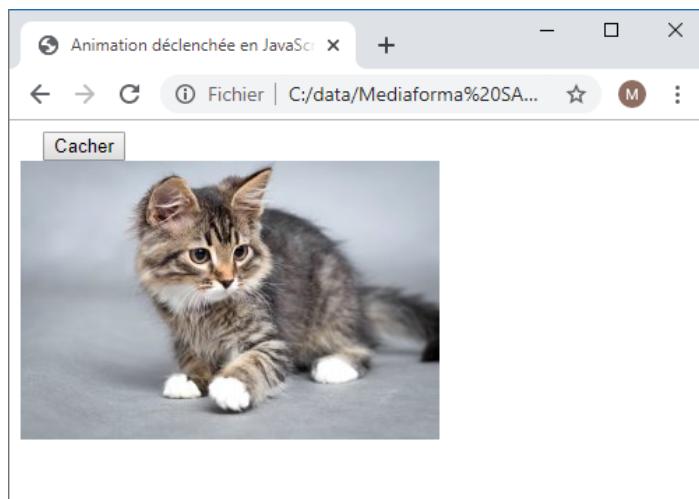
- Appliquer automatiquement des classes CSS pour les transitions et les animations
- Intégrer des bibliothèques d'animation CSS tierces, comme **Animate.css**
- Utiliser JavaScript pour manipuler directement le DOM durant les hooks de transition
- Intégrer des bibliothèques d'animation JavaScript tierces, comme par exemple **Velocity.js**

Nous allons examiner tour à tour ces possibilités dans les pages suivantes.

Transitions d'entrée et de sortie

Les transitions sont des effets qui peuvent être appliqués lorsque des éléments sont insérés, mis à jour et supprimés du DOM.

Examinons une transition traditionnelle en JavaScript et CSS. Le but du jeu est de faire disparaître l'image en 5 secondes au clic sur le bouton, puis de la faire réapparaître au deuxième clic sur le bouton :



En JavaScript (projet2-31.html) :

```
<!DOCTYPE html>
```

```

<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Animation déclenchée en JavaScript</title>
    <style>
      img {
        transition : opacity 5s ease-in;
      }
    </style>
    <script>
      function cacherAfficher() {
        var ima = document.getElementById('chat');
        var bout = document.getElementById('bouton');
        if (bout.innerHTML == "Cacher") {
          bout.innerHTML = 'Afficher';
          ima.setAttribute("style","opacity : 0;");
        }
        else {
          bout.innerHTML = 'Cacher';
          ima.setAttribute("style","opacity : 1;");
        }
      }
    </script>
  </head>
  <body>
    <button id="bouton" onclick="cacherAfficher ()">Cacher</button>
    <br>
    
  </body>
</html>

```

Transitions d'entrée et de sortie avec Vue.js

La technique à utiliser est bien différente dans Vue.js.

- 1)** Vous définirez un conteneur **<transition></transition>** et vous lui affecterez un attribut **name**.
- 2)** Dans cette balise, vous insérerez l'élément que vous voulez

- cacher/afficher en lui affectant une directive **v-if** ou **v-show**.
- 3) Vous définirez enfin les classes CSS **nom-enter-from**, **nom-enter-active**, **nom-leave-to** et **nom-leave-active** (si **nom** est le nom donné à l'attribut **name** de la balise **<transition>** **</transition>**).

.**nom-enter-from** : cette classe représente l'état de départ de la transition lors de l'apparition.

.**nom-leave-to** : cette classe représente l'état final de la transition lors de la disparition.

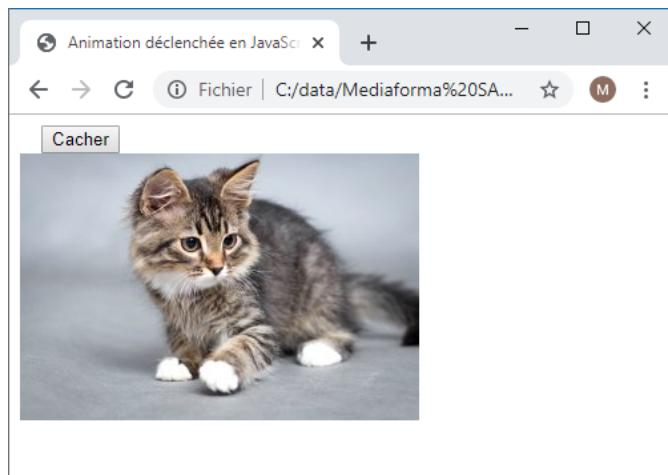
.**nom-enter-active** : cette classe définit les paramètres de la transition pour passer de l'état invisible à l'état visible.

.**nom-leave-active** : cette classe définit les paramètres de la transition pour passer de l'état visible à l'état invisible.

Si l'attribut **name** n'est pas défini dans la balise **<transition>** **</transition>**, les classes à utiliser sont **.v-enter**, **.v-enter-active**, **.v-leave-to** et **.v-leave-active**.

Challenge

Définissez une application Vue.js 3 pour obtenir le même résultat que dans le code précédent :



Solution (projet2-32.html) :

```
<style>
```

```

.t-enter-from, .t-leave-to {
    opacity : 0;
}
.t-enter-active, .t-leave-active {
    transition : opacity 2s;
}
</style>
...
<div id="app">
    <button v-on :click="visible=!visible;">
        <span v-if="visible">Cacher</span>
        <span v-else>Afficher</span>
    </button>
    <transition name="t">
        <div v-if="visible"></div>
    </transition>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                visible : true
            }
        }
    });
    let vm = app.mount('#app');
</script>

```

Transitions d'entrée et de sortie non simultanées

Les transitions simultanées d'entrée et de sortie ne sont pas toujours désirées. C'est pour cela que Vue propose des modes de transition alternatifs :

- **in-out** : La transition entrante du nouvel élément s'effectue en premier. Une fois terminée, la transition sortante de l'élément

courant est déclenchée.

- **out-in** : La transition sortante de l'élément courant s'effectue en premier. Une fois terminée, la transition entrante du nouvel élément est déclenchée.

Pour définir le type de la transition, utilisez l'attribut **mode** dans le composant **<transition></transition>** :

```
<transition name="tr" mode="in-out">  
</transition>
```

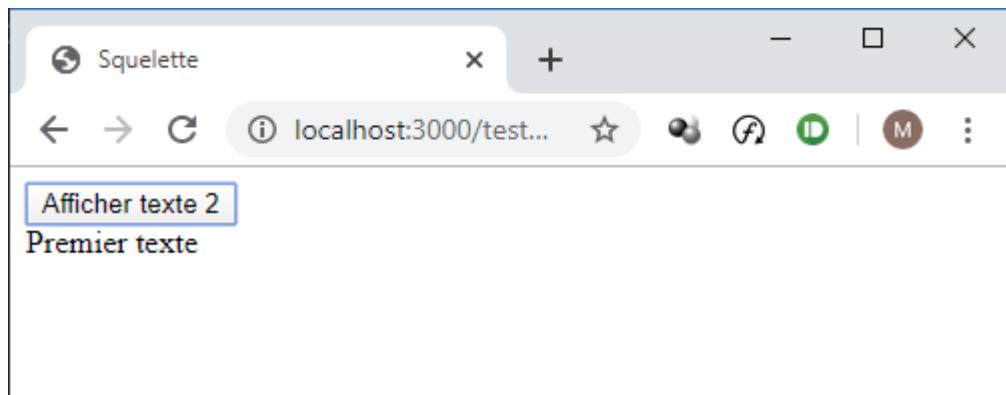
Ou :

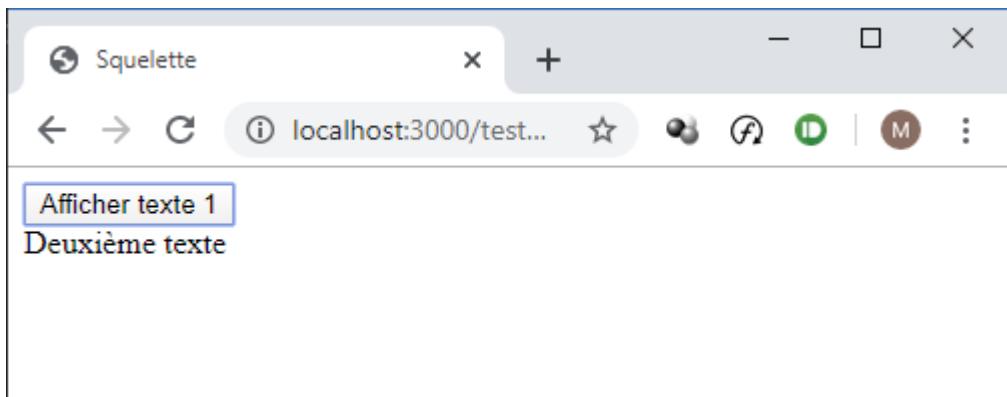
```
<transition name="tr" mode="out-in">  
</transition>
```

Challenge

Entraînez-vous à utiliser les deux valeurs de l'attribut **mode** pour remplacer un texte par un autre lorsqu'un bouton est cliqué. Au premier clic, "**Premier texte**" est remplacé par "**Deuxième texte**". Au deuxième clic, "**Deuxième texte**" est remplacé par "**Premier texte**". Ainsi de suite...

Les transitions se feront en une seconde, en jouant sur l'opacité.





Attention : vous devez définir un attribut **key** différent dans chacun des éléments à animer pour que Vue puisse les différentier.

Solution (projet2-33.html) :

```
<style>
  .t-enter-active,
  .t-leave-active {
    transition : opacity 4s ease;
  }
  .t-enter-from,
  .t-leave-to {
    opacity : 0;
  }
</style>
...
<div id="app">
  <button @click="label=!label">
    <span v-if="label">Afficher texte 2</span>
    <span v-else>Afficher texte 1</span>
  </button>
  <transition name="t" mode="out-in">
    <div v-if="label" key="1">Premier texte</div>
    <div v-else key="2">Deuxième texte</div>
  </transition>
</div>
<script>
  const app = Vue.createApp({
    data() {
```

```
        return {
          label : false
        }
      });
let vm = app.mount('#app');
</script>
```

Transitions entre éléments

En insérant plusieurs éléments affichés conditionnellement avec des directives **v-if**, **v-else-if** et **v-else** entre les balises **<transition>** et **</transition>** et en leur donnant un attribut **key** différent, il est possible d'enchaîner plusieurs transitions à chaque action de l'utilisateur. N'essayez pas d'insérer plusieurs éléments non liés par des directives **v-if**, **v-else-if** et **v-else**, sans quoi une erreur empêchera l'exécution de votre code.

Challenge

Définissez une nouvelle application qui comporte un bouton.

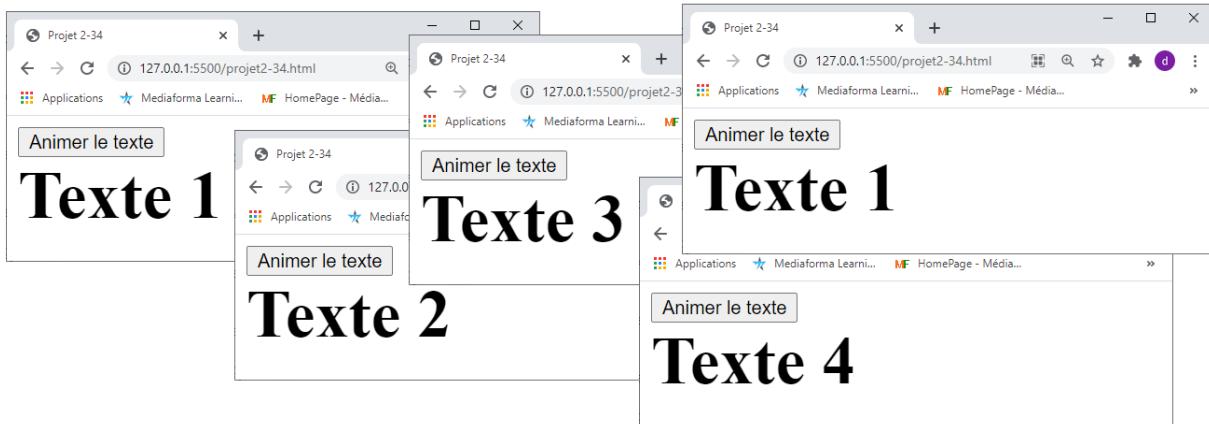
Le titre H1 "**Texte 1**" est affiché par défaut.

Lorsque l'utilisateur clique sur le bouton, affichez le titre H1 "**Texte 2**".

Lorsque l'utilisateur clique à nouveau sur le bouton, affichez le titre H1 "**Texte 3**".

Lorsque l'utilisateur clique à nouveau sur le bouton, affichez le titre H1 "**Texte 4**".

Lorsque l'utilisateur clique à nouveau sur le bouton, affichez le titre H1 "**Texte 1**" Ainsi de suite...



Solution (projet2-34.html) :

```

<style>
    .fade-enter-active,
    .fade-leave-active {
        transition : opacity 2s;
    }
    .fade-enter-from,
    .fade-leave-to {
        opacity : 0;
    }
    h1 {
        margin : 0;
        position : absolute;
        font-size : 3em;
    }
</style>
...
<div id="app">
    <button @click="anime<4?anime++ :anime=1">Animer le
    texte</button>
    <transition name="fade">
        <h1 v-if="anime==1" key="t1">Texte 1</h1>
        <h1 v-else-if="anime==2" key="t2">Texte 2</h1>
        <h1 v-else-if="anime==3" key="t3">Texte 3</h1>
        <h1 v-else="anime>3" key="t4">Texte 4</h1>
    </transition>
</div>

```

```
<script>
  const app = Vue.createApp({
    data() {
      return {
        anime : 1
      }
    }
  );
  let vm = app.mount('#app');
</script>
```

Transitions de listes

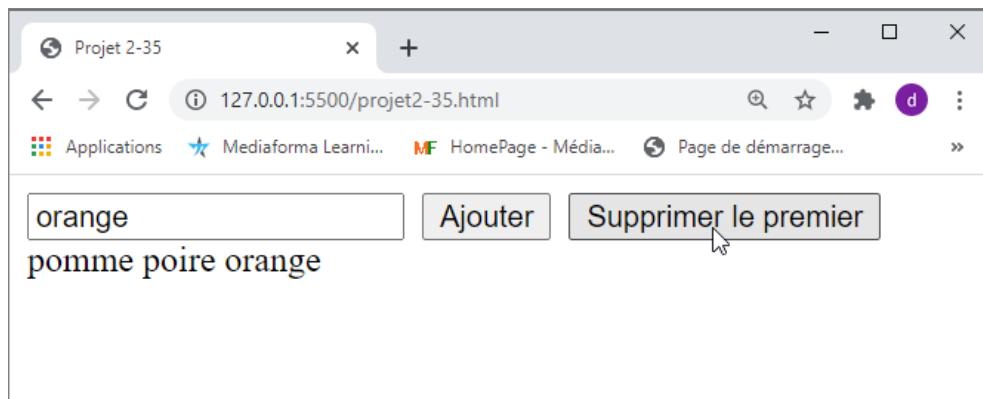
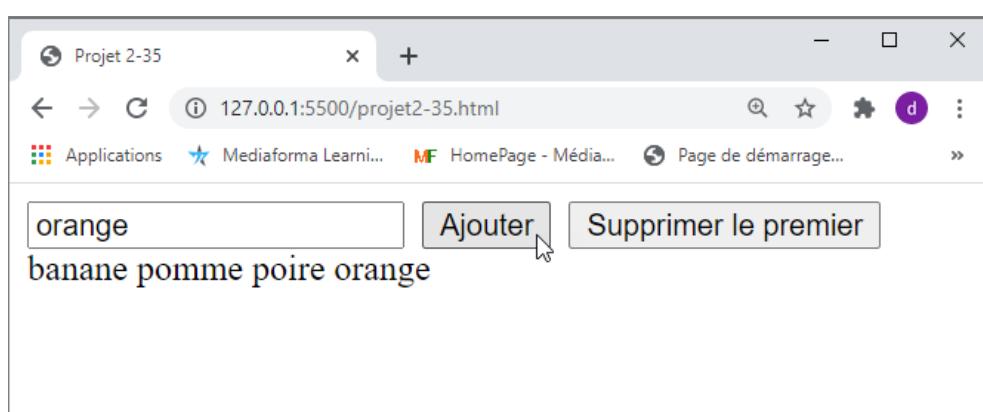
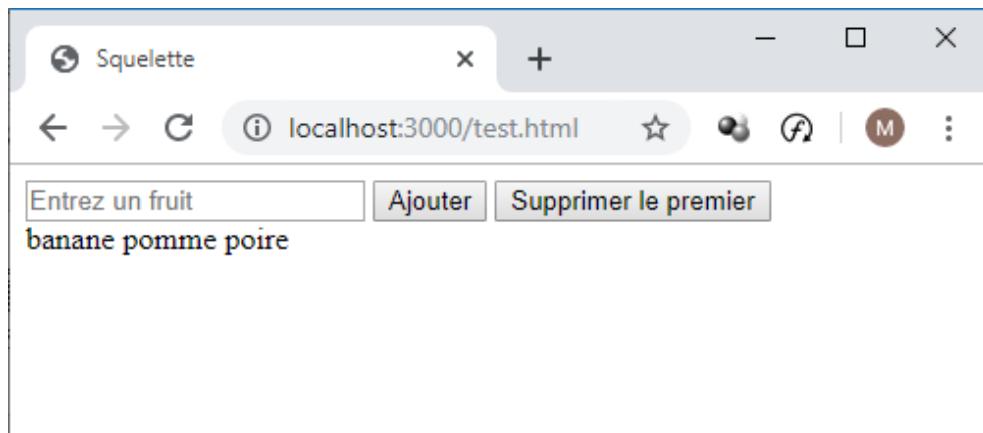
En utilisant un composant **<transition-group></transition-group>**, vous pouvez appliquer une transition lorsque vous ajoutez ou lorsque vous supprimez des éléments dans une liste.

Quelques restrictions. Vous devez :

- 1)** Donner un nom à la balise **<transition-group>** et faire porter l'animation sur ce nom.
- 2)** Ajouter un attribut **v-bind :key** unique dans chaque élément de la liste.

Challenge

Définissez une nouvelle application qui contient un tableau de fruits initialisé avec ces éléments : **banane**, **pomme** et **poire**. Créez une interface qui permet d'ajouter un nouveau fruit à la fin de la liste et de supprimer le premier fruit de la liste. Les ajouts et suppressions se feront avec une transition sur l'opacité d'une durée de deux secondes.



Solution (projet2-35.html) :

```
<style>
  .fade-enter-active,
  .fade-leave-active {
    transition : opacity 2s;
  }
  .fade-enter-from,
  .fade-leave-to {
    opacity : 0;
  }
</style>
```

```
        }

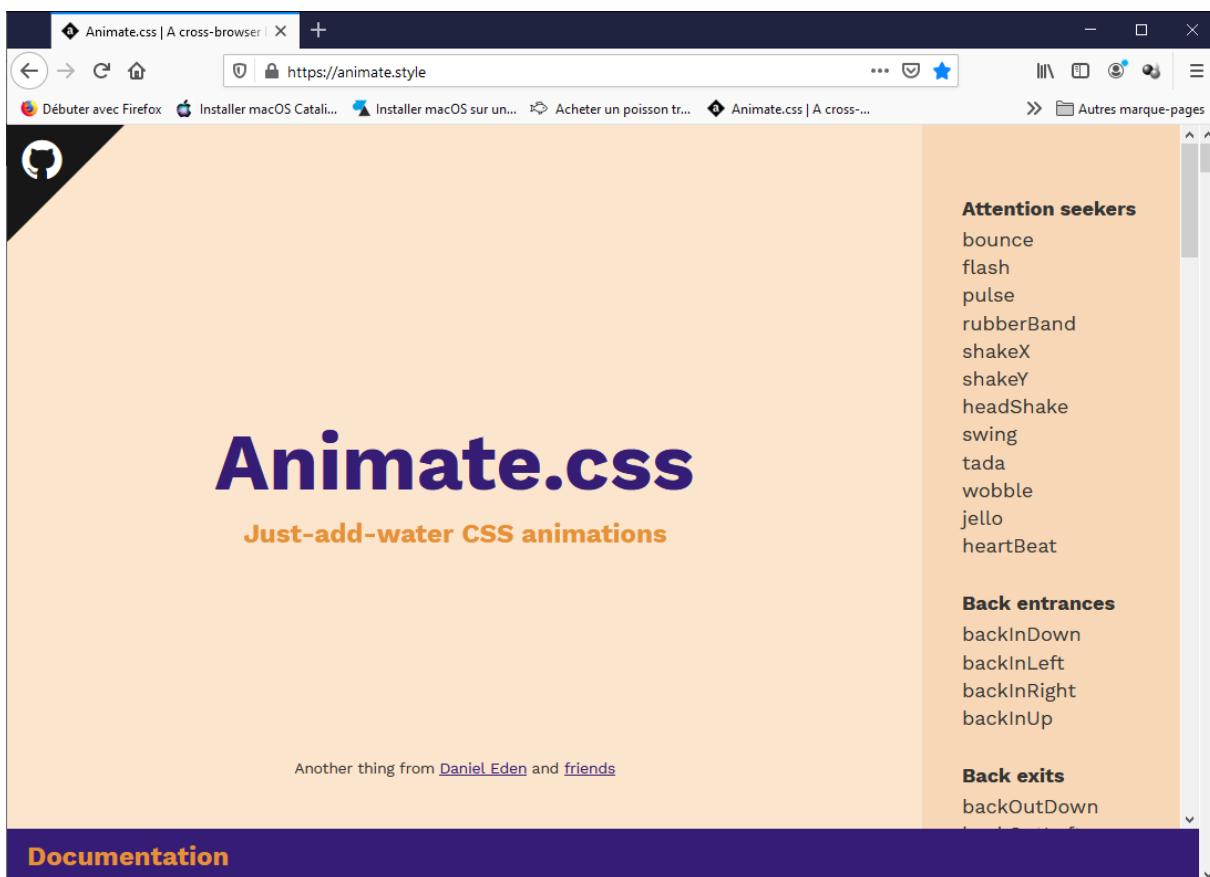
    input {
        margin-right : .5rem;
    }
</style>

...
<div id="app">
    <input type="text" v-model="fruit">
    <input type="button" value="Ajouter" @click="ajouter">
    <input type="button" value="Supprimer le premier"
@click="supprimer"><br>
    <transition-group name="fade">
        <span v-for="element in liste" :key="element">
{{element}}&nbsp;</span>
    </transition-group>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                liste : ['banane', 'pomme', 'poire'],
                fruit : ''
            }
        },
        methods : {
            ajouter() {
                this.liste.push(this.fruit);
            },
            supprimer() {
                this.liste.shift();
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

Transitions d'entrée et de sortie avec animate.css

Vous voulez aller plus loin avec les transitions d'entrée et de sortie dans vos applications Vue.js ? Une piste consiste à utiliser la feuille de styles **animate.css**.

Pour avoir une idée des animations possibles, allez sur la page <https://animate.style/>:



Les informations pour utiliser la feuille de styles Animate se trouvent sous le titre **Documentation**. Le plus simple consiste à ajouter une référence vers la feuille de styles **animate.css** dans le `<head></head>` de votre application. Vous ferez par exemple appel au CDN **cloudflare** :

```
https://cdnjs.cloudflare.com/ajax/libs/animate.css/4.1.1/animate.min.css
```

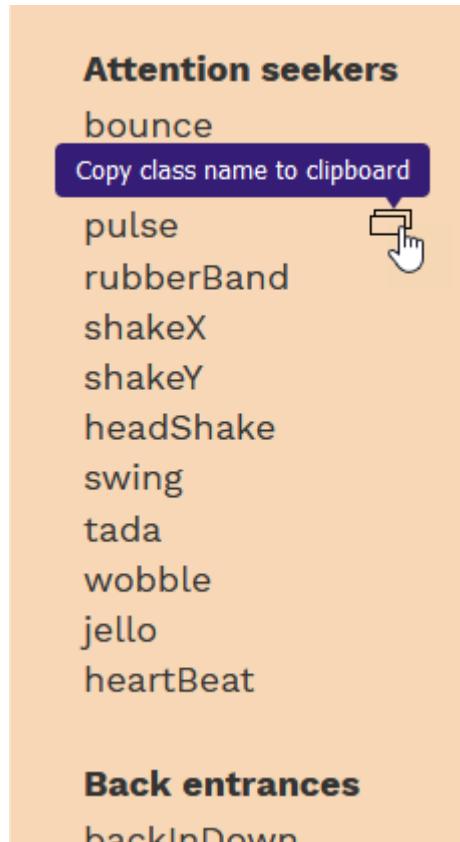
Ajoutez un composant `<transition></transition>` et insérez l'élément à animer à l'intérieur de ce composant (ici par exemple, un `<div></div>`) :

```
<transition>
  <div>Texte, image ou autre contenu</div>
</transition>
```

Définissez les animations d'entrée et de sortie avec les attributs `enter-active-class` et `leave-active-class` dans le composant `<transition></transition>` :

```
<transition
  enter-active-class="animate__animated anim1"
  leave-active-class="animate__animated anim2">
```

Où `anim1` et `anim2` sont les animations à utiliser. Utilisez le site <https://animate.style/> pour tester les différentes animations possibles et pour obtenir leur classe en cliquant sur l'icône à droite de l'animation choisie :



Définissez une directive `v-if` dans l'élément à animer pour gérer l'entrée et la sortie, et un bouton qui modifie la valeur de la variable utilisée dans le `v-`

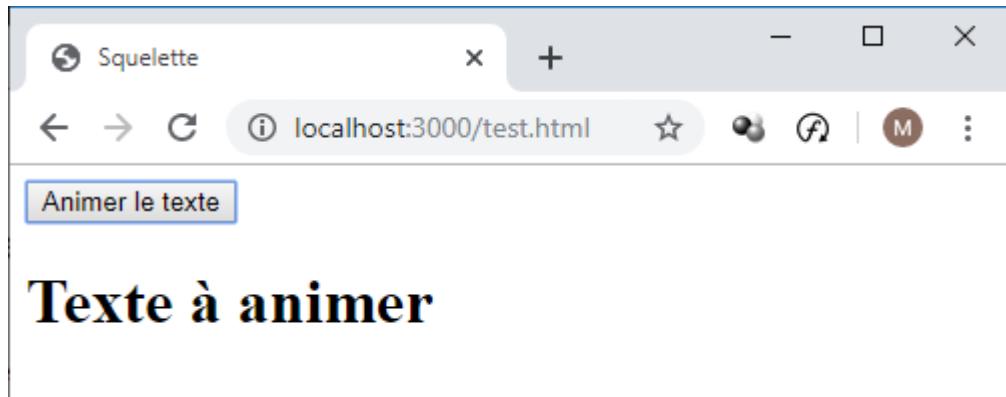
if.

Challenge

Animez un titre H1 en utilisant ces animations :

- Entrée : **animate_hinge**
- Sortie : **animate_bounceOutRight**

Les animations seront déclenchées par l'appui sur un bouton :



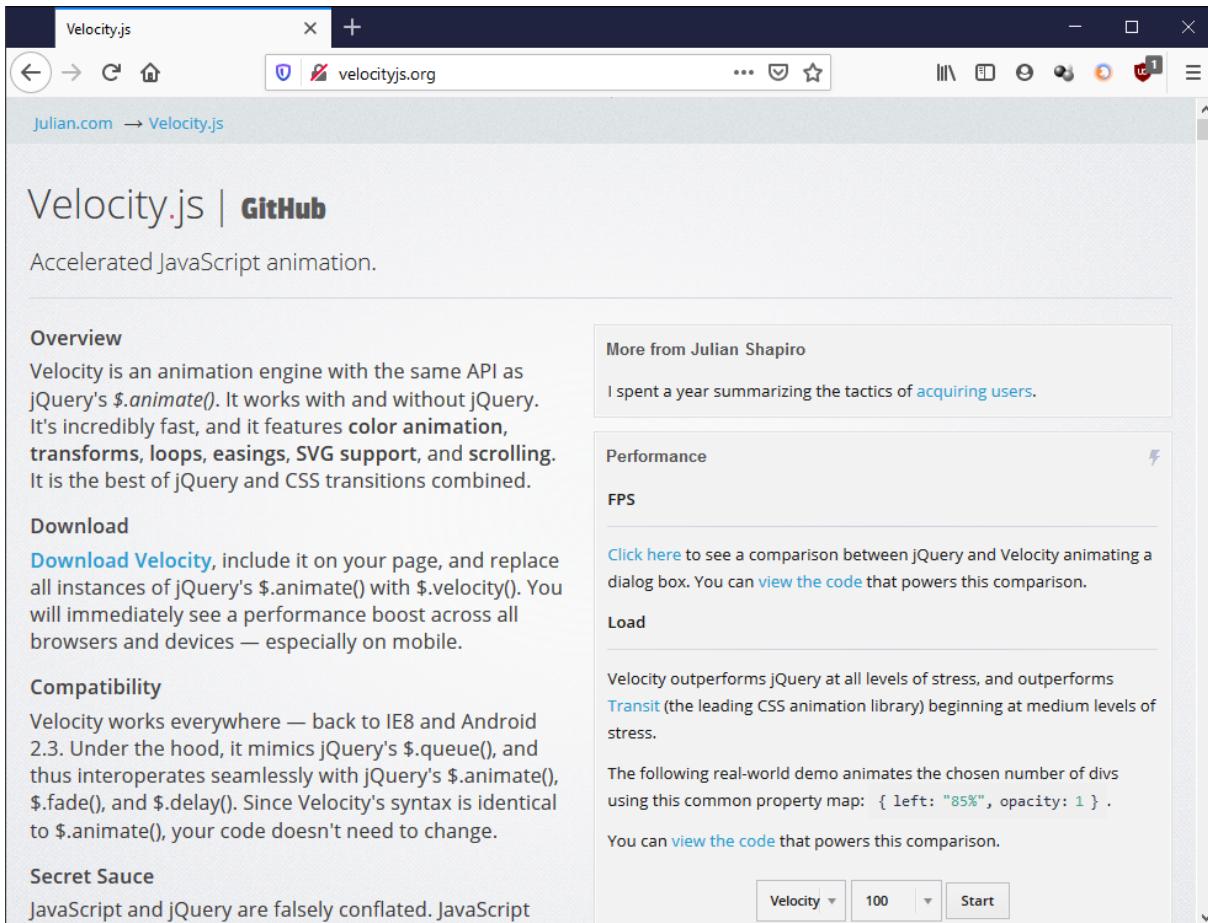
Solution (projet2-36.html) :

```
<div id="app">
    <button @click="anime=!anime">Animer le texte</button>
    <transition enter-active-class="animate_animated
animate_hinge"
                leave-active-class="animate_animated
animate_bounceOutLeft">
        <h1 v-if="anime">Texte à animer</h1>
    </transition>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                anime : false
            }
        }
    });
    let vm = app.mount('#app');
```

```
</script>
```

Animations en JavaScript

Les animations peuvent également se faire en JavaScript. A titre d'exemple, nous allons utiliser la bibliothèque **Velocity.js**, disponible sur <http://velocityjs.org/>.

A screenshot of a web browser displaying the Velocity.js website. The title bar says "Velocity.js". The address bar shows "velocityjs.org". The page content includes an "Overview" section with text about Velocity.js being an animation engine with the same API as jQuery's \$.animate(). It also features color animation, transforms, loops, easings, SVG support, and scrolling. Below this is a "Download" section, a "Compatibility" section (mentioning it works everywhere from IE8 to modern browsers), and a "Secret Sauce" section (noting it's JavaScript and jQuery are falsely conflated). To the right, there's a sidebar titled "More from Julian Shapiro" with a link to a post about user acquisition. Below that are sections for "Performance" (with an FPS counter) and "Load" (with a note about Velocity outperforming jQuery and Transit). At the bottom right is a control panel with dropdowns for "Velocity", "100", and a "Start" button.

Ajoutez une référence vers la dernière version de la bibliothèque **Velocity** dans le `<head></head>` de votre application. Vous ferez par exemple appel au CDN **cloudflare** :

<https://cdnjs.cloudflare.com/ajax/libs/velocity/2.0.6/velocity.min.js>

Tout comme pour une animation CSS, vous insérerez un composant `<transition></transition>` autour de l'élément à animer. Désactivez les éventuelles règles CSS sur cet élément avec la directive `v-bind :css="false"` et ajoutez un des gestionnaires événementiels suivants :

Évènement	Signification
before-enter	Avant l'exécution de la fonction enter
	Pendant l'insertion de l'élément
after-enter	Après l'insertion de l'élément
	Si l'animation d'entrée est annulée
before-leave	Avant l'exécution de la fonction leave
	Pendant la suppression de l'élément
after-leave	Après la suppression de l'élément
	Si l'animation de sortie est annulée

Par exemple, pour animer une image lorsqu'elle est affichée, vous utiliserez ce code :

```
<transition v-on :enter="traitement" v-bind :css="false">
  
</transition>
```

Les méthodes évènementielles sont définies dans le ViewModel de l'application ou du composant. L'élément sur lequel porte l'animation est automatiquement passé à la méthode évènementielle :

```
traitement : function(el, done) {
```

}

Pour appliquer une animation Velocity, écrivez quelque chose comme ceci :

```
Velocity(  
  el,  
  {  
    prop1 : ["valFin1", "valDeb1"],  
    ...  
    propN : ["valFinN", "valDebN"]  
  },  
  { duration : duree, easing : progression, complete : done }  
) ;
```

Où :

- **prop1 à propN** sont les propriétés CSS à animer,
- **valDeb1 à valDebN** sont les valeurs initiales des propriétés **prop1 à propN**
- **valFin1 à valFinN** sont les valeurs finales des propriétés **prop1 à propN**
- **durée** est la durée de l'animation en millisecondes
- **progression** est le type de progression de l'animation (**linear, ease, ease-in, ease-out, ease-in-out**)

Challenge

Définissez un bouton qui applique trois animations simultanées Velocity sur un titre H1 :

- Translation de **0 à 20 rem** vers la droite.
- Taille des caractères de **1 rem à 4 rem**.
- Opacité de **0 à 1**.

L'animation doit durer **2 secondes** et être de type **ease-out**.

Solution (projet2-37.html) :

```
<div id="app">  
  <button @click="anime=!anime">Animer le texte</button>  
  <transition @enter="traitement" :css="false">  
    <h1 v-if="anime">Texte à animer</h1>  
  </transition>
```

```
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                anime : false
            }
        },
        methods : {
            traitement(el, done) {
                Velocity(
                    el,
                    {
                        transform : ["translateX(20rem)",
                        "translateX(0)"],
                        fontSize : ["4rem", "1rem"],
                        opacity : [1, 0]
                    },
                    { duration : 2000, easing : "ease-out",
                    complete : done }
                );
            }
        });
        let vm = app.mount('#app');
    </script>
```

Challenge

Ajoutez le code nécessaire pour définir l'animation inverse, activée au deuxième clic sur le bouton.

Solution (projet2-38.html) :

```
<div id="app">
    <button @click="anime=!anime">Animer le texte</button>
    <transition @enter="traitement"
@leave="traitement2" :css="false">
        <h1 v-if="anime">Texte à animer</h1>
    </transition>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                anime : false
            }
        },
        methods : {
            traitement(el, done) {
                Velocity(
                    el,
                    {
                        transform : ["translateX(20rem)",
"translateX(0)"],
                        fontSize : ["4rem", "1rem"],
                        opacity : [1, 0]
                    },
                    { duration : 2000, easing : "ease-out",
complete : done }
                );
            },
            traitement2(el, done) {
                Velocity(
                    el, {

```

```

        transform : ['translateX(0)',  

'translateX(20rem)'],
        fontSize : ['1rem', '4rem'],
        opacity : [0, 1]
    },
{
    duration : 2000, easing : 'ease-out',
complete : done
}
);
}
);
}
);
let vm = app.mount('#app');
</script>

```

Partie 3 - Les applications SPA (*Single Page Applications*)

Création d'une application SPA avec vue-router

De nombreuses applications modernes sont basées sur le modèle SPA (*Single Page Application*).

Du point de vue des utilisateurs, cela signifie que l'ensemble du site Web ressemble à une application sur une seule page. Le temps pour passer d'une page à l'autre est réduit ... car il n'y a pas de nouvelles pages à charger : tout le site Web tient sur une seule page. C'est ainsi que Facebook, Google et de nombreux autres sites Web fonctionnent.

Les URL ne pointent plus vers des pages HTML, mais vers des états particuliers de l'application (qui ressemblent le plus souvent à des pages différentes).

Supposons que votre application se trouve dans le fichier **index.html**. Lorsque l'utilisateur veut consulter la page **B**, cette information est passée à **index.html** qui l'interprète comme un itinéraire vers la page **B**.

La suite suppose que vous utilisez Visual Studio Code et que l'extension Liver Server y a été installée.

Passons à la pratique.

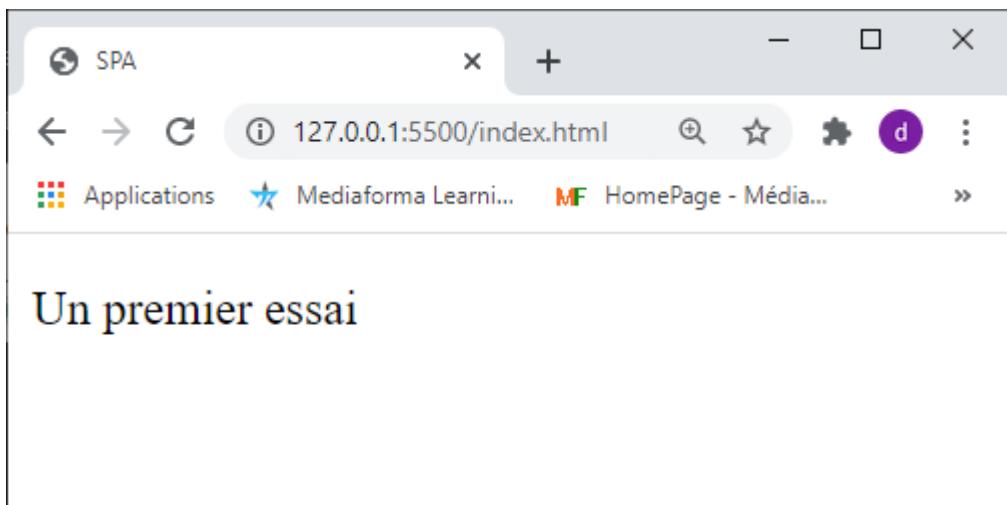
Créez un nouveau dossier (**spa** par exemple) dans votre dossier de développement.

Créez le fichier **index.html** dans ce dossier :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=<p>Un premier essai</p>, initial-scale=1.0">
    <title>SPA</title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
    <p>Un premier essai</p>
</body>
</html>
```

Pour ouvrir ce code dans Live Server, cliquez du bouton droit dans le code et sélectionnez Open with Live Server dans le menu contextuel. Le fichier **index.html** est lancé dans le navigateur par défaut.

Voici ce que vous devez obtenir :



Vue.js 3 implémente le modèle SPA *via* le plugin **vue-router**. Pour **vue-router**, chaque route correspond à un composant. En d'autres termes, chaque composant est responsable de l'affichage d'une page de l'application.

Etape 1 - Pour pouvoir utiliser le plugin vue-router, il suffit d'inclure le CDN correspondant dans l'en-tête de l'application, juste en-dessous du CDN de Vue.js :

```
<script src="https://unpkg.com/vue-router@next"></script>
```

Vous allez maintenant définir votre première application SPA qui comporte trois pages.

Le passage d'une page à l'autre se fera avec un menu constitué d'une liste à puces.



Etape 2 - Commencez par définir le menu de l'application avec une simple liste à puces, et ajoutez un composant **<router-view></router-view>** dans lequel seront rendues les pages de l'application :

```
<div id="app">
  <h1>Première application SPA</h1>
  <ul>
    <li>Page 1</li>
    <li>Page 2</li>
    <li>Page 3</li>
  </ul>
  <router-view></router-view>
</div>
```

Etape 3 - Vous allez maintenant modifier les éléments de la liste pour pointer sur les différentes pages de l'application :

```
<li><router-link to="/">Page 1</router-link></li>
<li><router-link to="/page2">Page 2</router-link></li>
<li><router-link to="/page3">Page 3</router-link></li>
```

Le composant `<router-link></router-link>` est utilisé pour naviguer dans l'application.

Ici, le chemin "/" pointe vers la première page, le chemin "/**page2**" pointe vers la deuxième page et le chemin "/**page3**" pointe vers la troisième page. Le composant `<router-link></router-link>` ajoute des liens (`to` est l'équivalent de `href`) sur les éléments de la liste et pointe vers les composants correspondants.

Etape 4 – Définition des composants (*ie* des pages de l'application)

Définissez les trois composants locaux qui correspondent aux trois pages de l'application :

```
const Page1 = { template : "<div>Bienvenue sur la page 1</div>" };  
const Page2 = { template : "<div>Bienvenue sur la page 2</div>" };  
const Page3 = { template : "<div>Bienvenue sur la page 3</div>" };
```

Etape 5 – Définition du routeur

Définissez le routeur avec la méthode `createRouter()` de la classe **VueRouter**.

Initialisée à `VueRouter.createWebHistory()`, la propriété **history** permet d'utiliser l'API **history** du langage HTML5. Ainsi, les URL des différentes pages de l'application seront bien plus SEO friendly : les hashtags seront remplacés par les chemins définis dans le tableau **routes**.

Les routes sont reliées aux composants dans le tableau **routes**.

clarifie les URL pour les moteurs de recherche) :

```
const router = VueRouter.createRouter({  
    history : VueRouter.createWebHistory(),  
    routes : [  
        { path : "/", component : Page1 },  
        { path : "/page2", component : Page2 },  
        { path : "/page3", component : Page3 }  
    ]  
});
```

Etape 6 – Définition d'une instance de Vue

Pour que l'application soit opérationnelle, définissez une instance de Vue en n'oubliant pas de faire référence au routeur avec l'instruction **app.use(router)** :

```
const app = Vue.createApp({  
});  
  
app.use(router);  
  
let vm = app.mount('#app');
```

L'application SPA est terminée. Vérifiez que vous pouvez vous déplacer dans ses trois pages en cliquant sur les liens.

Voici le code complet de l'application (c :\vue3\spa\index.html) :

```
<!DOCTYPE html>  
<html lang="en">  
  
<head>  
    <meta charset="UTF-8">  
    <meta http-equiv="X-UA-Compatible" content="IE=edge">  
    <meta name="viewport" content="width=<p>Un premier essai</p>, initial-scale=1.0">  
    <title>SPA</title>  
    <script src="https://unpkg.com/vue@next"></script>  
    <script src="https://unpkg.com/vue-router@next"></script>  
</head>  
  
<body>  
    <div id="app">  
        <h1>Première application SPA</h1>  
        <ul>  
            <li>  
                <router-link to="/">Page 1</router-link>  
            </li>  
            <li>  
                <router-link to="/page2">Page 2</router-link>  
            </li>
```

```

<li>
    <router-link to="/page3">Page 3</router-link>
</li>
</ul>
<router-view></router-view>
</div>
<script>
    const Page1 = { template : "<div>Bienvenue sur la page 1</div>" };
    const Page2 = { template : "<div>Bienvenue sur la page 2</div>" };
    const Page3 = { template : "<div>Bienvenue sur la page 3</div>" };
    const router = VueRouter.createRouter({
        history : VueRouter.createWebHistory(),
        routes : [
            { path : "/", component : Page1 },
            { path : "/page2", component : Page2 },
            { path : "/page3", component : Page3 }
        ]
    });
    const app = Vue.createApp({
    });
    app.use(router);
    let vm = app.mount('#app');
</script>
</body>

</html>

```

Il est possible de passer un objet à l'attribut to du composant **<router-link> **<router-link>**.**

Ce code :

```
<li><router-link to="/">Page 1</router-link></li>
```

Est équivalent à :

```
<li><router-link v-bind :to="{path : '/' }">Page 1</router-link>
</li>
```

Ou, en utilisant l'alias : à la place de la directive **v-bind** :

```
<li><router-link :to="{path : '/'}">Page 1</router-link></li>
```

Lorsque vous utilisez la directive **v-bind :to** (ou **:to**), vous pouvez ajouter un argument avec la propriété **query** :

```
<li>
  <router-link :to="{path : '/', query : {nom : 'Jean'}}">Page
1</router-link>
</li>
```



Allure du lien actif

Lorsqu'un lien de routage est cliqué, la classe **router-link-active** lui est affectée. Il est donc très simple de modifier l'allure du lien actif avec quelques lignes de CSS :

```
<style>
  .router-link-active {
    background : lightgreen;
  }
</style>
```

Si nécessaire, il est possible de changer le nom de la classe **router-link-active** en ajoutant l'attribut **active-class** au composant **<router-link>**. Ici par exemple, la classe affectée au lien de routage sélectionné sera **_active** :

```
<router-link to="/" active-class="_active">Page 1</router-link>
```

Indicateurs de navigation - Vue Router Navigation Guards

Les **Navigation Guards** permettent d'exécuter du code à certains moments précis dans une application Vue.js. Vous les mettrez en place pour (par

exemple) réservé certains contenus privés aux seuls utilisateurs enregistrés.

Vous savez utiliser les Lifecycle Hooks du Core Vue.js. Il existe également les Navigation Guards en rapport avec les applications SPA et les changements de routes.

Il existe trois types de Navigation Guards :

- **Global** : exécuté à chaque changement de route
- **Par route** : exécuté sur une route particulière
- **Par composant** : exécuté sur certains composants spécifiques

Nous allons les examiner dans les pages qui suivent.

Indicateurs de navigation globaux

La navigation se met "en attente" jusqu'à ce que tous les Navigation Guards soient résolus.

Vous pouvez mettre en place ces Navigation Guards (il y en a d'autres) :

```
router.beforeEach((to, from, next) => {}); // Lorsqu'une navigation est déclenchée
```

ou :

```
router.beforeEach(function(from, to, next) {});
```

```
router.afterEach((to, from) => {}); // Après chaque navigation
```

ou :

```
router.afterEach(function(to, from) {});
```

to est la route vers laquelle on navigue, **from** est la route depuis laquelle se fait la navigation et **next()** est appelée pour résoudre le hook.

Lorsque la fonction **next()** est appelée, on peut lui transmettre aucun paramètre ou le booléen **false** :

- **next()** se déplace jusqu'au prochain hook de navigation ou finalise la navigation si aucun autre hook n'est présent.
- **next(false)** annule la navigation

Pour mettre en place un Navigation Guard global, insérez la fonction **beforeEach()** ou **afterEach()** après la définition du routeur :

```
const router = VueRouter.createRouter({
  history : VueRouter.createWebHistory(),
  routes : [
    { path : "/", component : Page1 },
    { path : "/page2", component : Page2 },
    { path : "/page3", component : Page3 }
  ]
}) ;
router.beforeEach((to, from, next) => {
});
```

Challenge

En partant de ce code (c :\vue3\spa\index.html), ajoutez un Navigation Guard **beforeEach()**. Affichez **to** et **from** dans la console et testez les différentes façons d'appeler **next()**.

```
<div id="app">
  <h1>Première application SPA</h1>
  <ul>
    <li>
      <router-link to="/">Page 1</router-link>
    </li>
    <li>
      <router-link to="/page2">Page 2</router-link>
    </li>
    <li>
      <router-link to="/page3">Page 3</router-link>
    </li>
  </ul>
  <router-view></router-view>
</div>
<script>
  const Page1 = { template : "<div>Bienvenue sur la page 1</div>" };
  const Page2 = { template : "<div>Bienvenue sur la page 2</div>" };
  const Page3 = { template : "<div>Bienvenue sur la page 3</div>" };
</script>
```

```

    const Page2 = { template : "<div>Bienvenue sur la page 2</div>" };
    const Page3 = { template : "<div>Bienvenue sur la page 3</div>" };
    const router = VueRouter.createRouter({
        history : VueRouter.createWebHistory(),
        routes : [
            { path : "/", component : Page1 },
            { path : "/page2", component : Page2 },
            { path : "/page3", component : Page3 }
        ]
    });
    const app = Vue.createApp({
    });
    app.use(router);
    let vm = app.mount('#app');
</script>

```

Solution :

En ajoutant ce code, on vérifie que la navigation n'est jamais résolue car **next()** n'est jamais appelée :

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

En ajoutant ce code, on vérifie les routes **from** et **to** dans la console. La navigation a bien lieu :

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

En ajoutant ce code, on vérifie les routes **from** et **to** dans la console. La navigation n'a pas lieu :

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

Indicateurs de navigation par route

L'intercepteur de navigation **beforeEnter** peut être défini directement sur une route :

```
const router = VueRouter.createRouter({
  history : VueRouter.createWebHistory(),
  routes : [
    { path : "/", component : Page1 },
    { path : "/page2", component : Page2 },
    {
      path : "/page3",
      component : Page3,
      beforeEnter :(to, from, next) => {
        console.log(to, from);
        next();
      }
    }
  ]
});
```

Indicateurs de navigation par composant

Il est possible de définir une interception de navigation dans un composant en utilisant les fonctions **beforeRouteEnter()**, **beforeRouteUpdate()** et **beforeRouteLeave()** :

```
const Page2 = {
  template : "<div>Bienvenue sur la page 2</div>",
  beforeRouteEnter : (to, from, next) => {
    // Avant la confirmation de la route
  },
  beforeRouteUpdate : (to, from, next) => {
    // Quand la route change
  },
  beforeRouteLeave : (to, from, next) => {
```

```

    // Juste avant le changement de route
}
};


```

Routage – Transition entre les pages

Pour appliquer une transition entre chacune des pages d'une application SPA, encapsulez le composant `<router-view></router-view>` dans un composant `<transition></transition>` :

```

<transition>
  <router-view></router-view>
</transition>


```

Si vous voulez appliquer des transitions différentes en fonction des pages affichées, vous pouvez utiliser des transitions par routes :

```

const Page1 = {
  template : "<transition name='tr1'><div></div></transition>"
};

const PageN = {
  template : "<transition name='trN'><div></div></transition>"
};


```

Le composant `<transition></transition>` s'utilise comme pour l'animation des éléments sur une page.

- 1)** Définissez un attribut **name** dans le composant `<transition>`.
- 2)** Définissez un attribut **mode** dans le composant `<transition>` et affectez-lui la valeur **in-out** ou **out-in**.
- 3)** Définissez les classes CSS **nom-enter**, **nom-enter-active**, **nom-leave-to** et **nom-leave-active** (si **nom** est le nom donné à l'attribut **name** du composant `<transition></transition>`).

.nom-enter-from : cette classe représente l'état de départ de la transition lors de l'apparition.

.nom-enter-active : cette classe définit les paramètres de la transition pour passer de l'état invisible à l'état visible.

.nom-leave-to : cette classe représente l'état final de la transition lors de la disparition.

.nom-leave-active : cette classe définit les paramètres de la transition pour passer de l'état visible à l'état invisible.

Si l'attribut **name** n'est pas défini dans la balise `<transition></transition>`, les classes à utiliser sont **.v-enter-from**, **.v-enter-active**, **.v-leave-to** et **.v-leave-active**.

Challenge

En partant de ce code (c :\vue3\spa\index.html), définissez une transition de 2 secondes sur l'opacité à chaque changement de page. Essayez successivement les modes **in-out** et **out-in** sur les transitions.

```
<body>
  <div id="app">
    <h1>Première application SPA</h1>
    <ul>
      <li><router-link to="/page1">Page 1</router-link></li>
      <li><router-link to="/page2">Page 2</router-link></li>
      <li><router-link to="/page3">Page 3</router-link></li>
    </ul>
    <router-view></router-view>
  </div>
  <script>
    const Page1 = { template : "<div>Bienvenue sur la page 1</div>" };
    const Page2 = { template : "<div>Bienvenue sur la page 2</div>" };
    const Page3 = { template : "<div>Bienvenue sur la page 3</div>" };
    const router = new VueRouter({
      routes : [
        { path : "/page1", component : Page1 },
        { path : "/page2", name : "page2", component : Page2 },
        { path : "/page3", component : Page3 }
      ]
    })
  </script>
</body>
```

```

    });
    var vm = new Vue({
        router,
        el : "#app"
    });
</script>
</body>

```

Solution (c :\vue3\spa2\index.html) :

```

<style>
    .fade-enter-active,
    .fade-leave-active {
        transition : opacity 2s ease;
    }
    .fade-enter-from,
    .fade-leave-to {
        opacity : 0;
    }
</style>
...
<div id="app">
    <h1>Première application SPA</h1>
    <ul>
        <li>
            <router-link to="/">Page 1</router-link>
        </li>
        <li>
            <router-link to="/page2">Page 2</router-link>
        </li>
        <li>
            <router-link to="/page3">Page 3</router-link>
        </li>
    </ul>
    <router-view v-slot="{Component}">
        <transition name="fade" mode="out-in">
            <component :is="Component"></component>
        </transition>
    </router-view>
</div>

```

```

        </router-view>
</div>
<script>
    const Page1 = { template : "<div>Bienvenue sur la page 1</div>" };
    const Page2 = { template : "<div>Bienvenue sur la page 2</div>" };
    const Page3 = { template : "<div>Bienvenue sur la page 3</div>" };
    const router = VueRouter.createRouter({
        history : VueRouter.createWebHistory(),
        routes : [
            { path : "/", component : Page1 },
            { path : "/page2", component : Page2 },
            { path : "/page3", component : Page3 }
        ]
    });
    const app = Vue.createApp({
    });
    app.use(router);
    let vm = app.mount('#app');
</script>

```

La syntaxe de la transition est particulière. Dans le composant **<router-view></router-view>**, on définit un **slot par défaut** qui contient le composant courant (**v-slot="{Component}"**). La transition est définie comme enfant du router-view et le composant courant (**Page1**, **Page2** ou **Page3**) est injecté dans le composant **<transition></transition>** :

```

<router-view v-slot="{Component}">
    <transition name="fade" mode="out-in">
        <component :is="Component"></component>
    </transition>
</router-view>

```

Partie 4 - L'interface en ligne de commande Vue-Cli

Jusqu'ici, vous avez appris à écrire des applications Vue "à la main", en définissant un fichier HTML qui contenait tout ce dont vous aviez besoin. Cette approche est concevable tant que le nombre de lignes du projet n'est pas trop important. Elle vous a permis de comprendre la philosophie du framework Vue.js.

Pour de plus gros projets, il sera intéressant d'avoir une approche "composant" dans laquelle chaque composant Vue.js est écrit dans un fichier séparé. Il serait également intéressant d'utiliser **Babel** pour coder en ES6. Pour cela, nous allons utiliser l'outil en ligne de commande **Vue-Cli**.

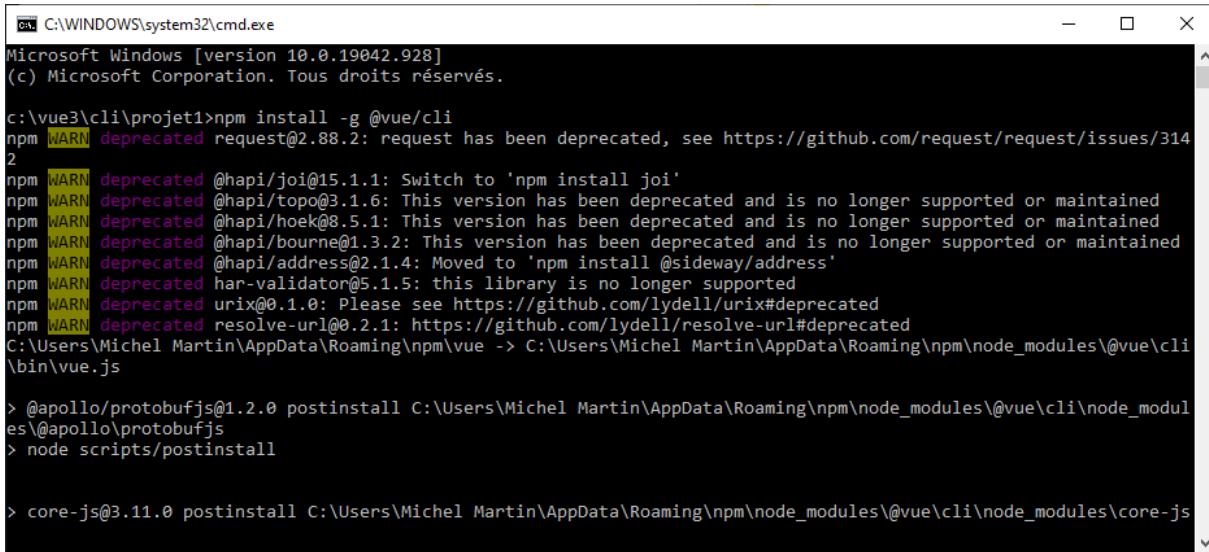
Vue-Cli intègre tout ce dont vous avez besoin pour créer une application Vue.js avec une approche composant. Nous utiliserons le *module bundler* (l'organisateur de modules) **Webpack**. Ceci afin d'organiser les fichiers utilisés (JavaScript, HTML, CSS, images, polices, etc.) en modules afin de simplifier le développement.

L'outil Vue-Cli

Nous allons créer un premier projet simple avec Vue-Cli.

Ouvrez une invite de commandes dans le dossier où vous voulez créer le nouveau projet. Lancez la commande suivante pour installer la dernière version de Vue-Cli :

```
npm install -g @vue/cli
```



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [version 10.0.19042.928]
(c) Microsoft Corporation. Tous droits réservés.

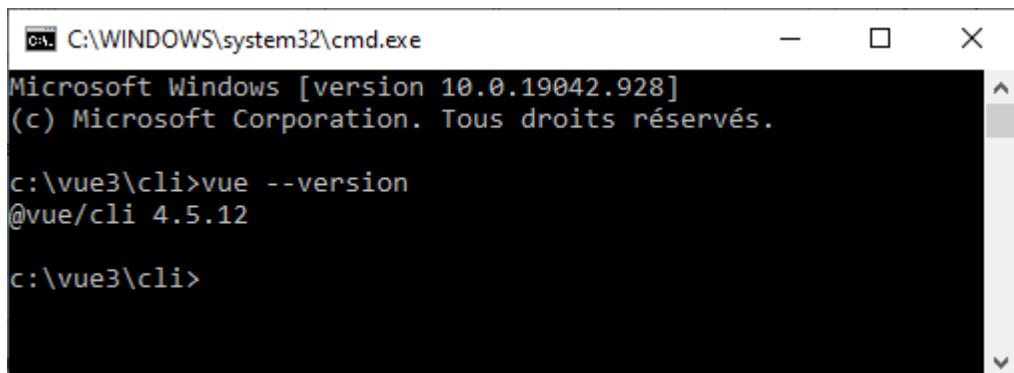
c:\vue3\cli\projet1>npm install -g @vue/cli
npm WARN deprecated request@2.88.2: request has been deprecated, see https://github.com/request/request/issues/3142
npm WARN deprecated @hapi/joi@15.1.1: Switch to 'npm install joi'
npm WARN deprecated @hapi/topo@3.1.6: This version has been deprecated and is no longer supported or maintained
npm WARN deprecated @hapi/hoek@8.5.1: This version has been deprecated and is no longer supported or maintained
npm WARN deprecated @hapi/bourne@1.3.2: This version has been deprecated and is no longer supported or maintained
npm WARN deprecated @hapi/address@2.1.4: Moved to 'npm install @sideway/address'
npm WARN deprecated har-validator@5.1.5: this library is no longer supported
npm WARN deprecated urix@0.1.0: Please see https://github.com/lydell/urix#deprecated
npm WARN deprecated resolve-url@0.2.1: https://github.com/lydell/resolve-url#deprecated
C:\Users\Michel Martin\AppData\Roaming\npm\vue -> C:\Users\Michel Martin\AppData\Roaming\npm\node_modules\@vue\cli\bin\vue.js

> @apollo/protobufjs@1.2.0 postinstall C:\Users\Michel Martin\AppData\Roaming\npm\node_modules\@vue\cli\node_modules\@apollo\protobufjs
> node scripts/postinstall

> core-js@3.11.0 postinstall C:\Users\Michel Martin\AppData\Roaming\npm\node_modules\@vue\cli\node_modules\core-js
```

Une fois Vue-Cli installé, vous pouvez vérifier sa version avec cette commande :

```
vue --version
```



```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [version 10.0.19042.928]
(c) Microsoft Corporation. Tous droits réservés.

c:\vue3\cli>vue --version
@vue/cli 4.5.12

c:\vue3\cli>
```

Si vous aviez déjà installé Vue-Cli, vous pouvez le mettre à jour avec cette commande :

```
npm update -g @vue/cli
```

Pour créer un projet, lancez la commande suivante :

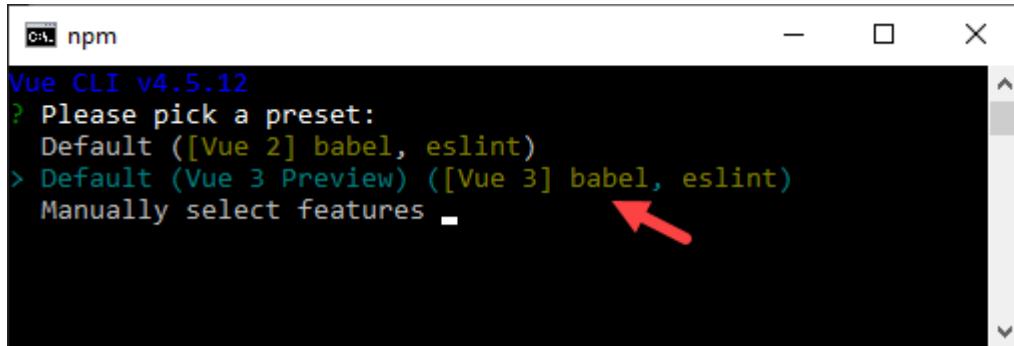
```
vue create projet
```

Où **projet** est le nom du projet à créer. Ce projet sera stocké dans un dossier dont le nom est celui du projet.

Pour créer le projet **projet1** dans le dossier c :\vue3\cli, créez ce dossier, allez dedans et lancez cette commande :

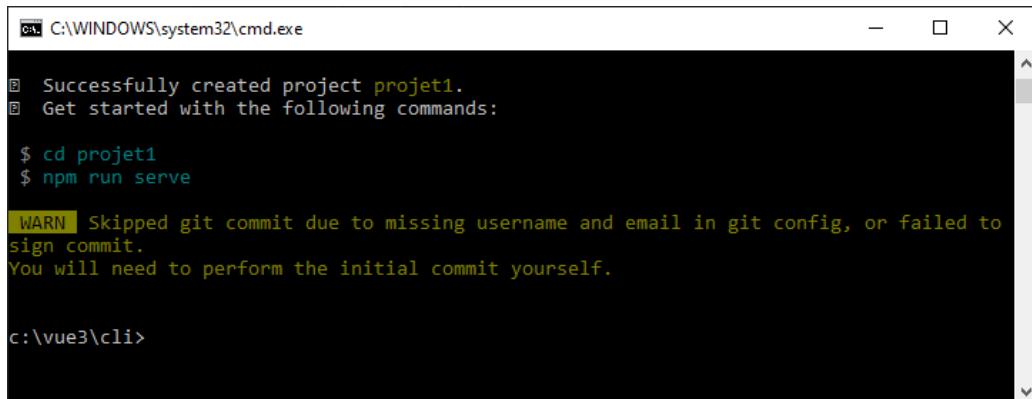
```
vue create projet1
```

Quelques instants plus tard, on vous demande si vous voulez créer un projet Vue 2 ou Vue 3. Sélectionnez **Vue 3** en appuyant sur la touche flèche *vers le bas*, puis validez en appuyant sur la touche *Entrée* :



```
npm
Vue CLI v4.5.12
? Please pick a preset:
  Default ([Vue 2] babel, eslint)
> Default (Vue 3 Preview) ([Vue 3] babel, eslint)
  Manually select features -
```

Patinez jusqu'à ce que l'application soit créée. Vous obtiendrez quelque chose comme ceci :



```
C:\WINDOWS\system32\cmd.exe
Successfully created project projet1.
Get started with the following commands:

$ cd projet1
$ npm run serve

WARN Skipped git commit due to missing username and email in git config, or failed to
sign commit.
You will need to perform the initial commit yourself.

c:\vue3\cli>
```

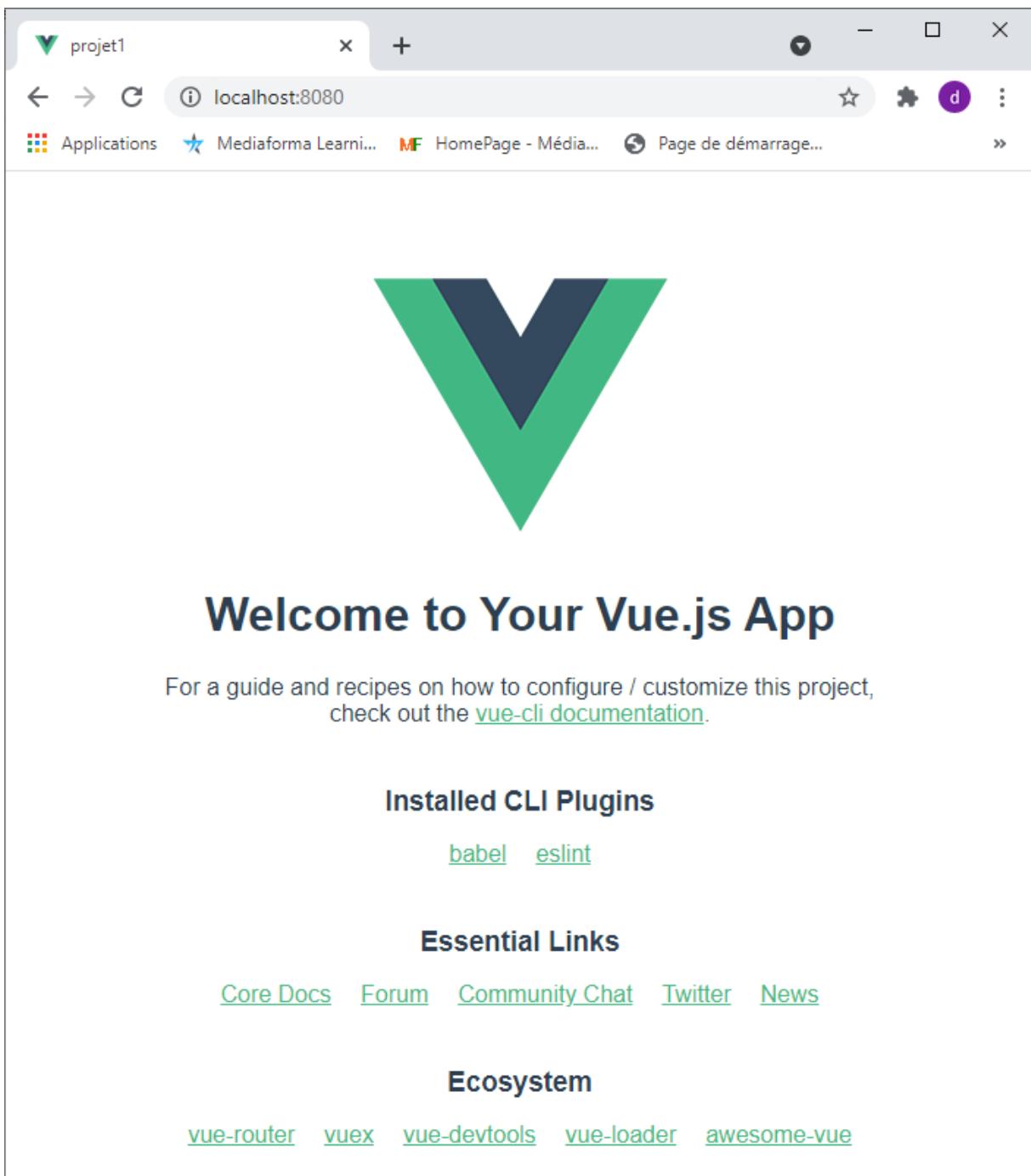
Une fois la commande **vue create** exécutée, lancez les commandes suivantes :

```
cd dossier (où dossier est le dossier du projet) :
npm run serve
```

L'application est accessible à cette adresse :

```
http://localhost:8080
```

Voici ce que vous devriez obtenir :



Bravo !

Vous venez de créer votre première application avec Vue-Cli.

Architecture de l'application

L'application est (entre autres) composée :

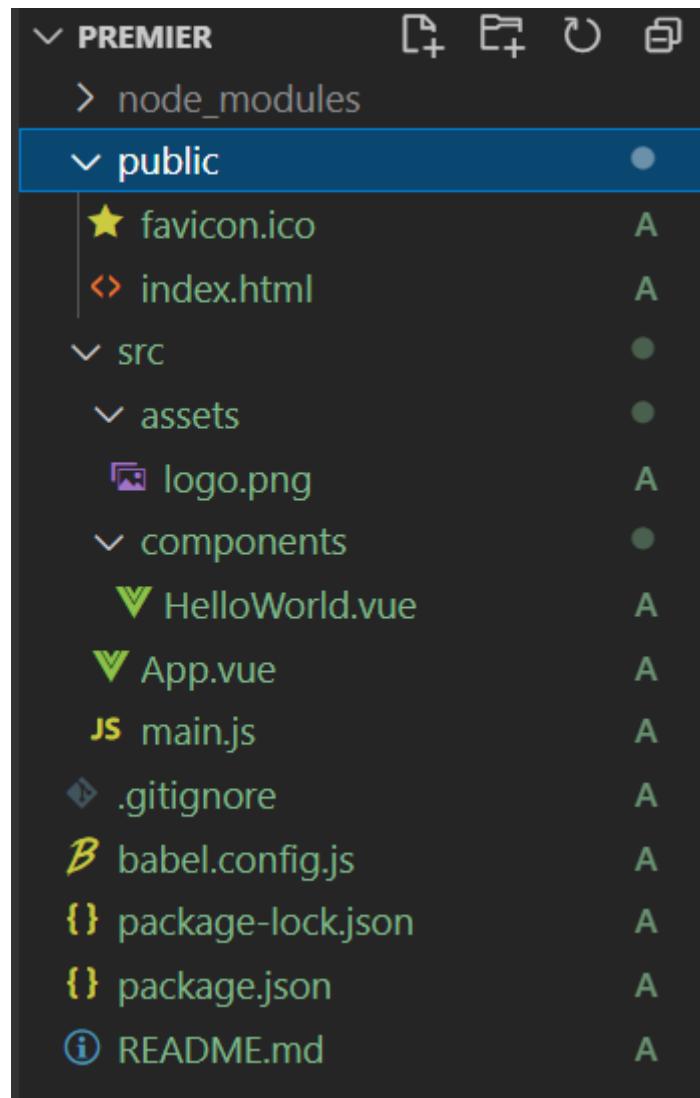
- d'un fichier **src/main.js** qui servira de point d'entrée
- d'un fichier **public/index.html** pour afficher l'application
- de composants sous la forme de fichiers d'extension **.vue** (dans les dossiers **src** et **src/components**)

Deux composants sont disponibles de base :

- **App.vue** : le root component
- **HelloWorld.vue** : le composant **Hello world**

Les fichiers d'extension **.vue** sont des SFC (*Single File Components*) réutilisables.

Examinons l'architecture de cette application :



Remarquez en particulier :

- Le dossier **src/components** qui contient les composants (ici **HelloWorld.vue**).
- Le dossier **src/assets** qui contient les fichiers annexes tels que images, CSS, etc. (ici logo.png).

Examinons les fichiers de l'application.

Le fichier public/index.html

```
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-
scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't work properly without JavaScript enabled. Please enable it to continue.</strong>
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

Ce fichier se contente de définir le conteneur de la vue :

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

Vous pourrez également l'utiliser pour faire référence à des bibliothèques JavaScript ou CSS externes en insérant les balises correspondantes entre les balises **<head>** et **</head>**.

Les fichiers .vue

Les fichiers d'extension **.vue** sont des composants. Ils encapsulent le code Javascript, le template et le CSS de leur composant à l'aide (respectivement) de balises **<script>**, **<template>** et **<style>**.

Par défaut, le code JavaScript des composants est développé en ES6. Ce code est transpilé avec **Babel** pour être compréhensible par tous les navigateurs.

Vous pouvez ajouter l'attribut **scoped** dans la balise **<style>** d'un composant. Dans ce cas, le CSS est restreint au composant. Si vous voulez gérer le CSS de l'application de façon globale, définissez-le dans le fichier **index.html**, sans l'attribut **scoped**.

Le fichier App.vue

```
<template>
  
  <HelloWorld msg="Welcome to Your Vue.js App"/>
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name : 'App',
  components : {
    HelloWorld
  }
}
</script>
<style>
#app {
  font-family : Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing : antialiased;
  -moz-osx-font-smoothing : grayscale;
  text-align : center;
  color : #2c3e50;
  margin-top : 60px;
}
```

```
}
```

```
</style>
```

Le template contient une image et le composant <**HelloWorld**/>.

Le script importe le composant **HelloWorld.vue** en utilisant une instruction import ES6 :

```
import HelloWorld from './components/HelloWorld.vue'
```

La syntaxe à utiliser est :

```
import nom from 'chemin'
```

Où :

- **nom** est l'alias du module vue à importer, c'est-à-dire le nom à utiliser dans le composant pour représenter le module. Dans notre exemple, vous auriez tout aussi bien pu écrire :

```
import Toto from './components/HelloWorld.vue'
```

Mais alors, vous auriez dû faire référence à l'alias Toto dans le template :

```
<template>
  
  <Toto msg="Welcome to Your Vue.js App"/>
</template>
```

- **chemin** est le chemin relatif de ce fichier dans l'arborescence de l'application, par rapport au fichier courant. Ici, **App.vue** se trouve dans le dossier **src** et le fichier **HelloWorld.vue** est dans le sous-dossier **components**. D'où le chemin **./components/HelloWorld.vue**.

L'instruction **export default** effectue deux actions :

- 1) Elle définit le nom du composant (**name**). Cette propriété est nécessaire si on veut pouvoir utiliser ce composant dans un autre composant.
- 2) Elle enregistre le composant avec la propriété **components**. Cette propriété est nécessaire pour pouvoir utiliser le composant **HelloWorld** dans le composant **App**. Un objet est affecté à la propriété **components** :

```
export default {
```

```
name : 'App',  
components : {  
    HelloWorld  
}  
}
```

Ici, **HelloWorld** est l'abréviation de :

'HelloWorld': HelloWorld

Le nom de la balise Le nom du composant

D'autres éléments peuvent être spécifiés dans la section export default : des props ou des méthodes par exemple.

La feuille de styles définit la mise en forme/mise en page de style des éléments affichés dans le composant **App.vue**.

Le fichier Main.js

```
import { createApp } from 'vue'  
import App from './App.vue'  
  
createApp(App).mount('#app')
```

Ce fichier importe la méthode `createApp()` et le composant `App.vue`. L'application est créée en passant le composant `App` à la fonction `createApp()`, puis elle est montée avec la fonction `mount()`. Le paramètre de cette fonction est un sélecteur CSS qui indique quel élément HTML va être contrôlé par l'application `Vue.js`.

Puis il crée l'application **App** et la monte en l'attachant à la balise d'id `app`.

Le fichier components/HelloWorld.vue

```
<template>  
  <div class="hello">
```

```
<h1>{{ msg }}</h1>
<p>
    For a guide and recipes on how to configure / customize this
project,<br>
    check out the
    <a href="https://cli.vue.js.org" target="_blank"
rel="noopener">vue-cli documentation</a>.
</p>
<h3>Installed CLI Plugins</h3>
<ul>
    <li><a href="https://github.com/Vue.js/vue-
cli/tree/dev/packages/%40vue/cli-plugin-babel" target="_blank"
rel="noopener">babel</a></li>
    <li><a href="https://github.com/Vue.js/vue-
cli/tree/dev/packages/%40vue/cli-plugin-eslint" target="_blank"
rel="noopener">eslint</a></li>
</ul>
<h3>Essential Links</h3>
<ul>
    <li><a href="https://Vue.js.org" target="_blank"
rel="noopener">Core Docs</a></li>
    <li><a href="https://forum.Vue.js.org" target="_blank"
rel="noopener">Forum</a></li>
    <li><a href="https://chat.Vue.js.org" target="_blank"
rel="noopener">Community Chat</a></li>
    <li><a href="https://twitter.com/Vue.js" target="_blank"
rel="noopener">Twitter</a></li>
    <li><a href="https://news.Vue.js.org" target="_blank"
rel="noopener">News</a></li>
</ul>
<h3>Ecosystem</h3>
<ul>
    <li><a href="https://router.Vue.js.org" target="_blank"
rel="noopener">vue-router</a></li>
    <li><a href="https://vuex.Vue.js.org" target="_blank"
rel="noopener">vuex</a></li>
    <li><a href="https://github.com/Vue.js/vue-devtools#vue-
devtools" target="_blank" rel="noopener">vue-devtools</a></li>
    <li><a href="https://vue-loader.Vue.js.org" target="_blank"
rel="noopener">vue-loader</a></li>
```

```

        <li><a href="https://github.com/Vue.js/awesome-vue"
target="_blank" rel="noopener">awesome-vue</a></li>
      </ul>
    </div>
</template>

<script>
export default {
  name : 'HelloWorld',
  props : {
    msg : String
  }
}
</script>

<!-- Add "scoped" attribute to limit CSS to this component only -->
<style scoped>
h3 {
  margin : 40px 0 0;
}
ul {
  list-style-type : none;
  padding : 0;
}
li {
  display : inline-block;
  margin : 0 10px;
}
a {
  color : #42b983;
}
</style>
```

La partie **<template>** définit ce qui s'affiche dans le composant.

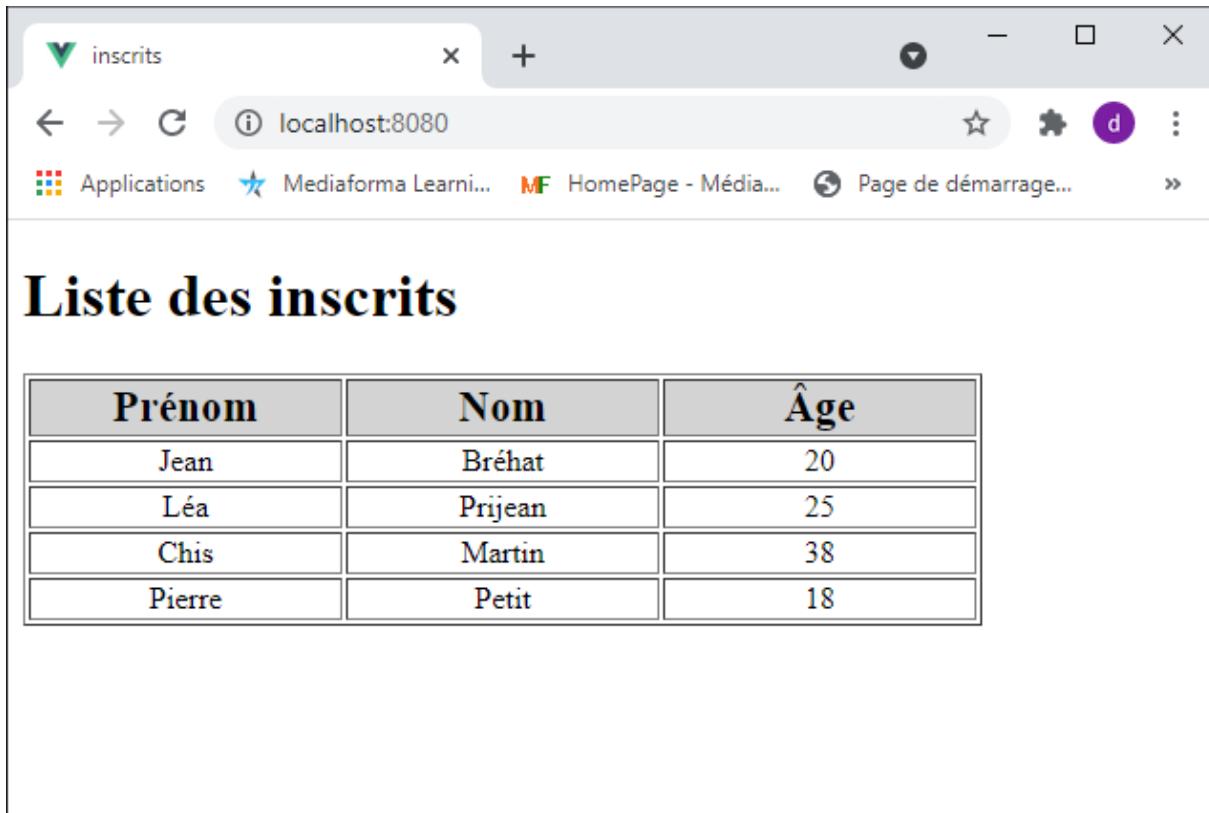
Dans l'instruction **export default**, on définit le nom du composant et on indique que la props **msg** sera utilisée dans le composant **HelloWorld**.

La partie `<style>` définit la mise en forme des éléments du composant (**scoped**).

Création et modification d'une application avec Vue-Cli

Définissez une nouvelle application avec Vue-Cli.

Cette application utilisera le composant **Inscrits** pour obtenir le résultat suivant :



The screenshot shows a web browser window titled "inscrits". The address bar displays "localhost:8080". Below the address bar, there are several tabs: "Applications", "Mediaforma Learni...", "MF HomePage - Média...", and "Page de démarrage...". The main content area of the browser shows a heading "Liste des inscrits" followed by a table with four rows of data:

Prénom	Nom	Âge
Jean	Bréhat	20
Léa	Prijean	25
Chis	Martin	38
Pierre	Petit	18

Les données du tableau seront définies dans `App.vue`, et c'est le composant **Inscrits** qui les affichera.

Solution :

Créez un nouveau projet (**inscrits** par exemple). Ouvrez une invite de commandes. Allez dans le dossier où l'appli **inscrits** doit être créée et lancer cette commande :

```
vue create inscrits
```

Définissez un projet Vue 3.

Quelques secondes plus tard, le dossier **inscrits** a été créé.

Lancez l'environnement de développement avec ces commandes :

```
cd inscrits
```

```
npm run serve
```

Supprimez tous les fichiers qui se trouvent dans le dossier **src**, à l'exception du fichier **main.js**. Dans le dossier **inscrits/public**, conservez le fichier **index.html** :

```
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't work properly without JavaScript enabled. Please enable it to continue.</strong>
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

La balise **<div>** définie dans le **<body>** contiendra l'application Vue.js.

Conservez également le fichier **main.js** dans le dossier **src** :

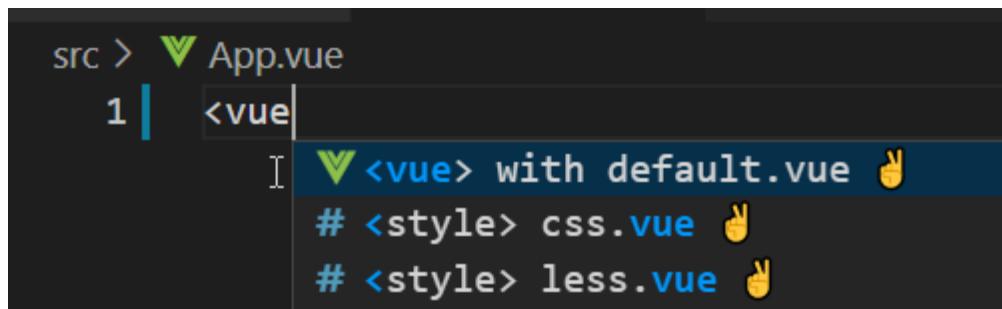
```
import { createApp } from 'vue'
import App from './App.vue'

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

Ici, on importe la fonction `createApp()` et le composant **App** pour pouvoir les utiliser dans la suite du code.

On crée ensuite une application Vue.js avec la fonction `createApp()`. L'application est montée et associée à l'élément d'id **app**.

Toujours dans le dossier **src**, vous allez définir le composant **App** (c'est-à-dire fichier **App.vue**). Si le plugin Vetur est installé, tapez `<vue` et appuyez sur la touche *Entrée* pour insérer un squelette standard de composant Vue dans le code :



```
src > App.vue
1 | <vue|
```

The screenshot shows a code editor window with the file path "src > App.vue". The cursor is positioned at the start of a tag, specifically "<vue|". A dropdown menu is open, displaying three suggestions: "<vue> with default.vue" (highlighted in blue), "# <style> css.vue", and "# <style> less.vue". Each suggestion is accompanied by a small yellow hand icon.

Voici ce que vous devez obtenir :

```
<template>

</template>

<script>
export default {

}
</script>
```

```
<style>
```

```
</style>
```

Complétez ce code comme ceci :

```
<template>
  <h1>Liste des inscrits</h1>
  <inscrits :liste="personnes"></inscrits>
</template>
```

```

<script>
import Inscrits from "./components/Inscrits.vue";
export default {
  name : "App",
  components : {
    Inscrits
  },
  data() {
    return {
      personnes : [
        { prenom : "Jean", nom : "Bréhat", age : 20 },
        { prenom : "Léa", nom : "Prijean", age : 25 },
        { prenom : "Chis", nom : "Martin", age : 38 },
        { prenom : "Pierre", nom : "Petit", age : 18 },
      ],
    };
  },
};
</script>
<style>
</style>

```

Le fichier **App.vue** contient trois balises : **<template>**, **<script>** et **<style>**.

- La balise **<template>** définit la partie HTML du composant. Ici, un titre **<h1>** et le composant **Inscrits** qui récupère les données via la props **liste**.
- La balise **<script>** est utilisée pour importer le composant **Inscrits**, pour exporter l'application et le composant **Inscrits**, et pour définir le tableau **personnes** qui contient les données.
- La balise **<style>** est utilisée pour définir les styles CSS utilisés dans le composant. Ici, on ne définit aucun style.

Le composant **Inscrits.vue** est très simple :

```

<template>
  <table border>
    <tr>
      <th>Prénom</th>
      <th>Nom</th>

```

```

        <th>Âge</th>
    </tr>
    <tr v-for="(personne, index) in liste" v-bind :key="index">
        <td>{{ personne.prenom }}</td>
        <td>{{ personne.nom }}</td>
        <td>{{ personne.age }}</td>
    </tr>
</table>
</template>

<script>
export default {
    name : "Inscrits",
    props : ["liste"],
}
</script>
<style scoped>
th {
    background : lightgrey;
    font-size : 140%;
}
td {
    width : 10rem;
    text-align : center;
}
</style>
```

Le fichier **App.vue** contient trois balises : **<template>**, **<script>** et **<style>**.

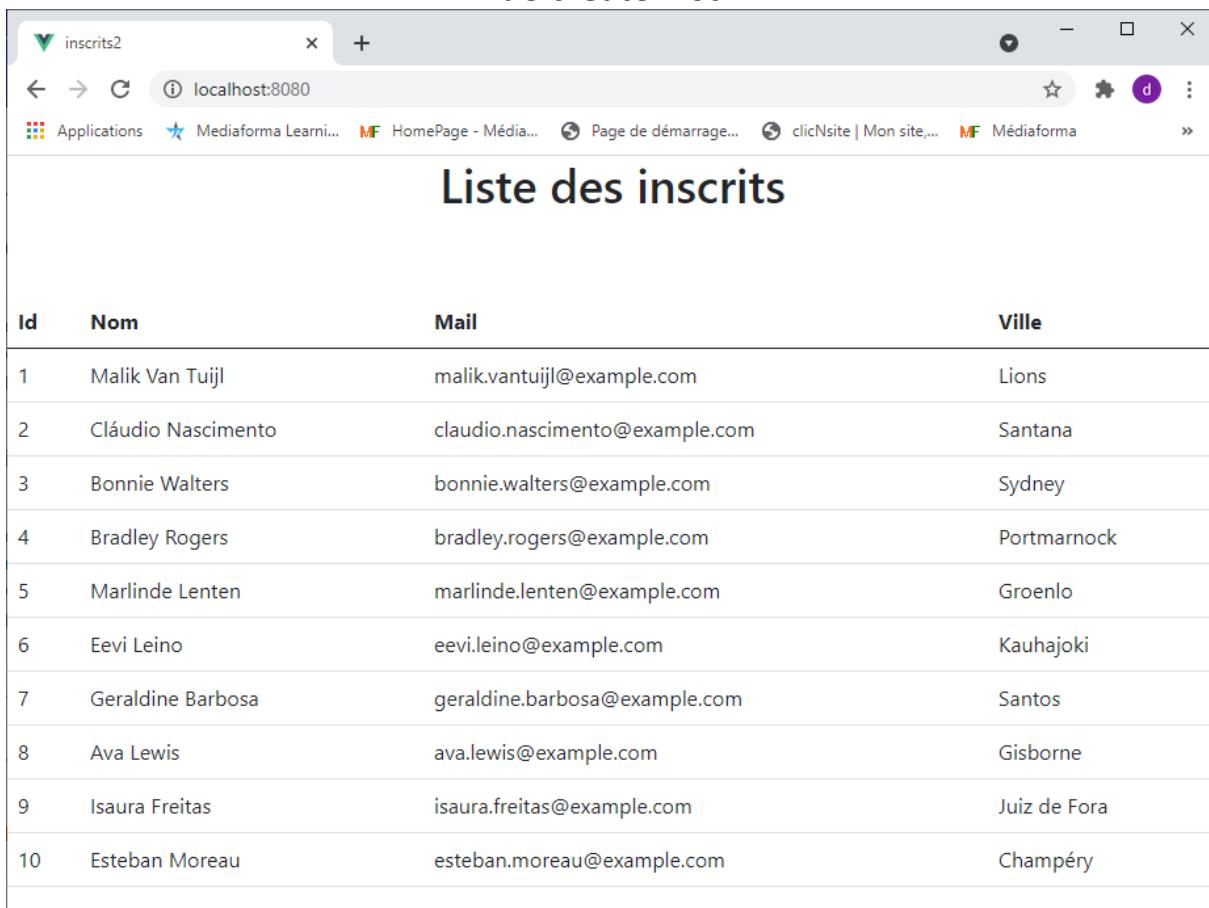
- La balise **<template>** définit le tableau et extrait les données à l'aide d'une directive **v-for**.
- La balise **<script>** exporte le composant **Inscrits** et la prop **liste**.
- La balise **<style>** définit le style des éléments du tableau.

Une deuxième application avec Vue-Cli

Définissez une nouvelle application avec Vue-Cli. Vous l'appellerez par exemple **inscrits2**.

Cette application utilisera le composant **Users** pour obtenir le résultat suivant :

Vue create inscri



Id	Nom	Mail	Ville
1	Malik Van Tuijl	malik.vantuijl@example.com	Lions
2	Cláudio Nascimento	claudio.nascimento@example.com	Santana
3	Bonnie Walters	bonnie.walters@example.com	Sydney
4	Bradley Rogers	bradley.rogers@example.com	Portmarnock
5	Marlinde Lenten	marlinde.lenten@example.com	Groenlo
6	Eevi Leino	eevi.leino@example.com	Kauhajoki
7	Geraldine Barbosa	geraldine.barbosa@example.com	Santos
8	Ava Lewis	ava.lewis@example.com	Gisborne
9	Isaura Freitas	isaura.freitas@example.com	Juiz de Fora
10	Esteban Moreau	esteban.moreau@example.com	Champéry

Les données seront obtenues dans le composant **Users** avec l'API REST **randomuser.me**, à partir de l'URL suivante : <https://randomuser.me/api/?results=10>

Les données seront lues en AJAX avec la bibliothèque **Axios** et mises en forme dans un tableau en utilisant la bibliothèque **Bootstrap**.

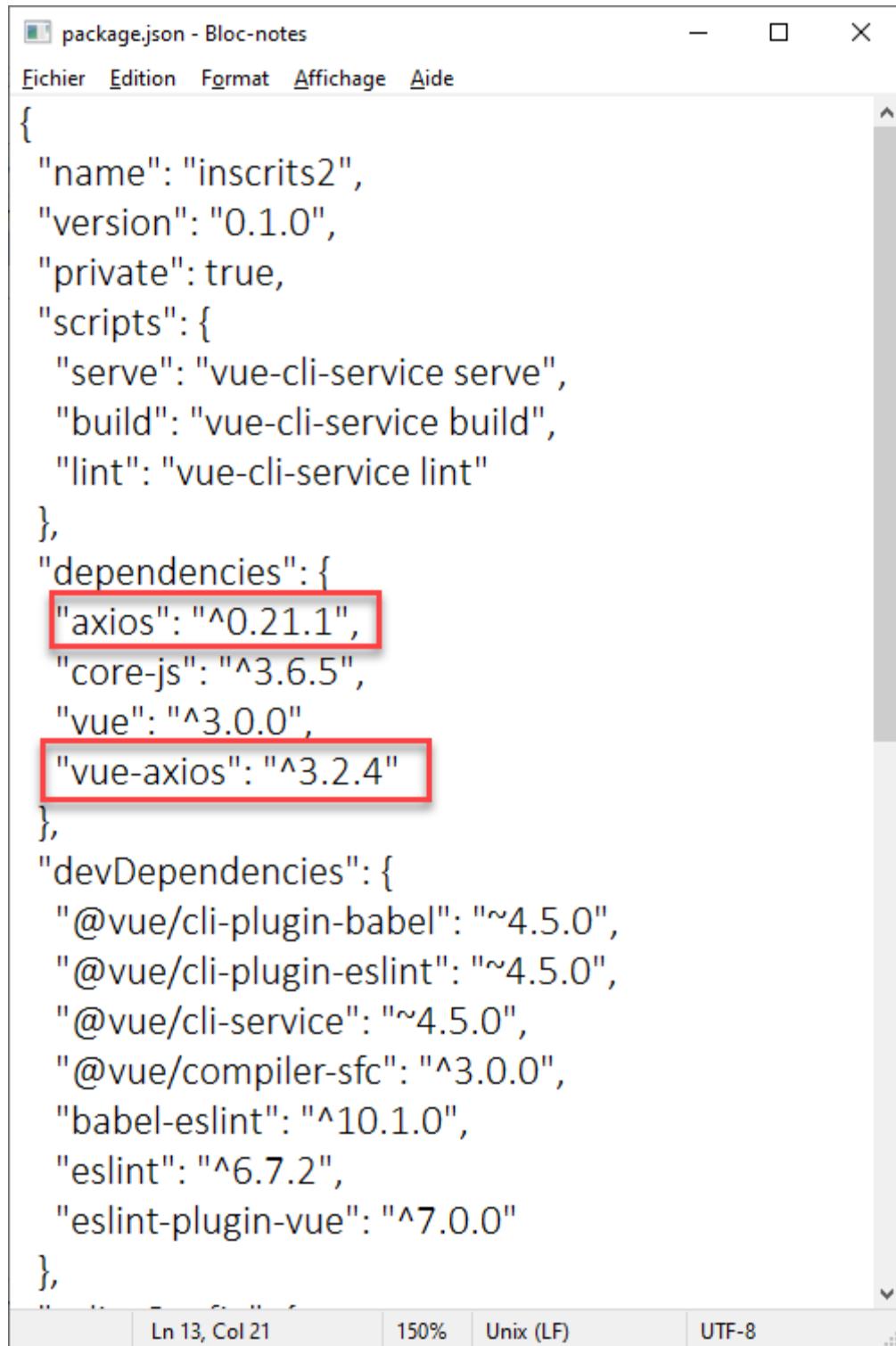
Quelques indices pour vous aider à utiliser Axios.

- 1) Pour pouvoir accéder à Axios dans Vue.js, vous devez installer le module **axios** avec ces commandes :

```
npm install axios
```

```
npm install vue-axios
```

Après avoir exécuté ces deux commandes, **axios** et **vue-axios** devraient apparaître dans la section **dependencies** du fichier **package.json** :



```
{  
  "name": "inscrits2",  
  "version": "0.1.0",  
  "private": true,  
  "scripts": {  
    "serve": "vue-cli-service serve",  
    "build": "vue-cli-service build",  
    "lint": "vue-cli-service lint"  
  },  
  "dependencies": {  
    "axios": "^0.21.1",  
    "core-js": "^3.6.5",  
    "vue": "^3.0.0",  
    "vue-axios": "^3.2.4"  
  },  
  "devDependencies": {  
    "@vue/cli-plugin-babel": "~4.5.0",  
    "@vue/cli-plugin-eslint": "~4.5.0",  
    "@vue/cli-service": "~4.5.0",  
    "@vue/compiler-sfc": "^3.0.0",  
    "babel-eslint": "^10.1.0",  
    "eslint": "^6.7.2",  
    "eslint-plugin-vue": "^7.0.0"  
  },  
}  
Ln 13, Col 21 | 150% | Unix (LF) | UTF-8
```

2) Dans le fichier **main.js**, ajoutez une instruction **import** pour

donner un accès global à Axios :

```
import axios from 'axios'
```

3) Montez la dépendance Axios avec cette instruction :

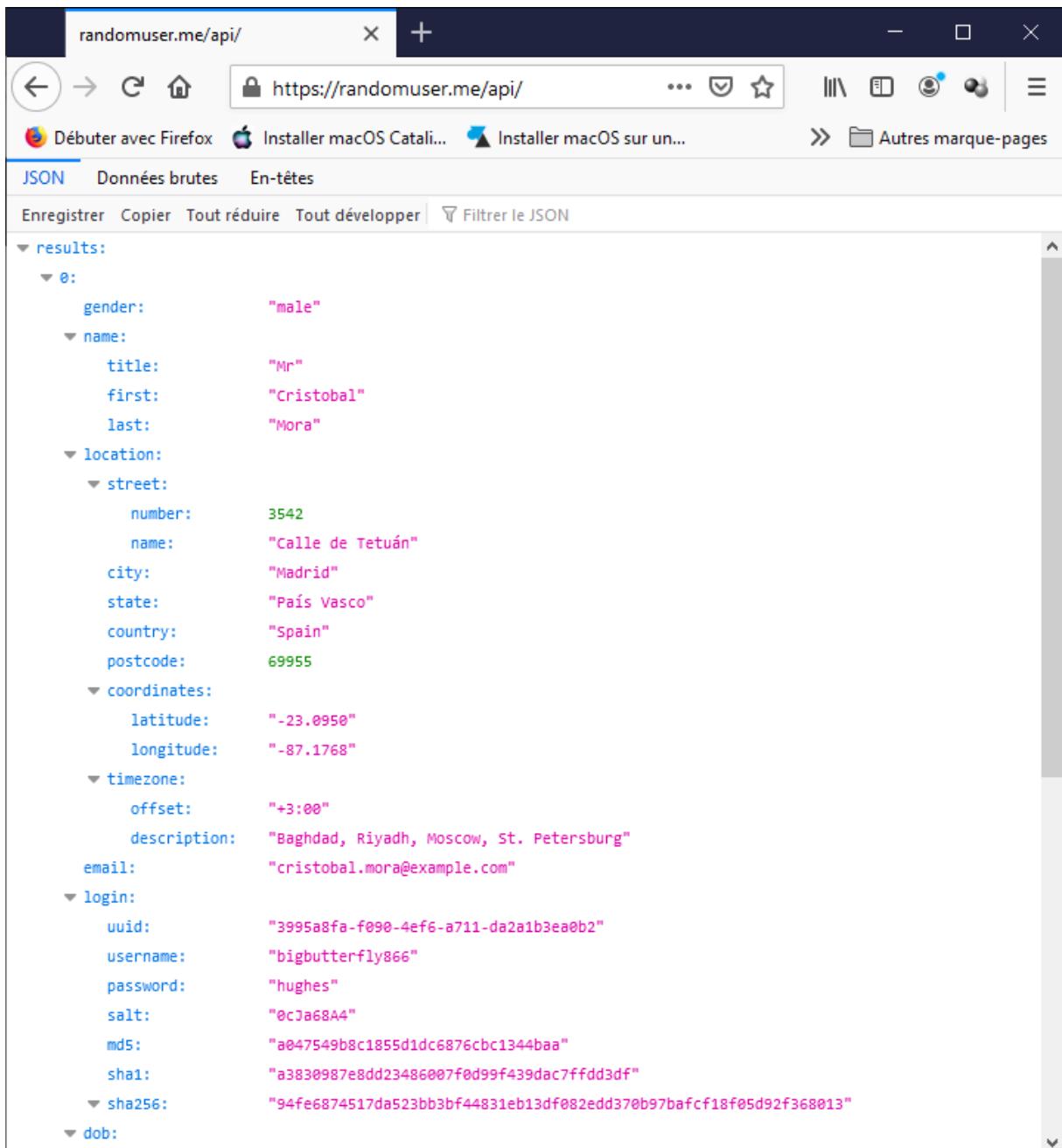
```
app.config.globalProperties.axios=axios
```

Voici à quoi devrait ressembler votre fichier **main.js** :

```
import { createApp } from 'vue'  
import App from './App.vue'  
import axios from 'axios'  
const app = createApp(App)  
app.config.globalProperties.axios=axios  
app.mount('#app')
```

4) Enfin, vous pouvez accéder à Axios dans un des composants de l'application avec **this.axios.**

A titre d'information, voici les informations retournées par l'API **randomuser.me** :



```
randomuser.me/api/ https://randomuser.me/api/ Débuter avec Firefox Installer macOS Catali... Installer macOS sur un... Autres marque-pages JSON Données brutes En-têtes Enregistrer Copier Tout réduire Tout développer Filtrer le JSON results: 0: gender: "male" name: title: "Mr" first: "Cristobal" last: "Mora" location: street: number: 3542 name: "Calle de Tetuán" city: "Madrid" state: "País Vasco" country: "Spain" postcode: 69955 coordinates: latitude: "-23.0950" longitude: "-87.1768" timezone: offset: "+3:00" description: "Baghdad, Riyadh, Moscow, St. Petersburg" email: "cristobal.mora@example.com" login: uid: "3995a8fa-f090-4ef6-a711-da2a1b3ea0b2" username: "bigbutterfly866" password: "hughes" salt: "0cJa68A4" md5: "a047549b8c1855d1dc6876cbc1344baa" sha1: "a3830987e8dd23486007f0d99f439dac7ffdd3df" sha256: "94fe6874517da523bb3bf44831eb13df082edd370b97bafcf18f05d92f368013" dob:
```

Solution :

Générez l'application **inscrits2** avec Vue-Cli, comme vous l'avez fait auparavant.

Pour mettre en forme les données avec Bootstrap, allez sur le site <https://getbootstrap.com/> et cliquez sur **Get started** pour récupérer la feuille de styles de Bootstrap :

Bootstrap · The most popular H X

https://getbootstrap.com

Débuter avec Firefox Installer macOS Catali... Installer macOS sur un... Acheter un poisson tr... Autres marque-pages

Build fast, responsive sites with Bootstrap

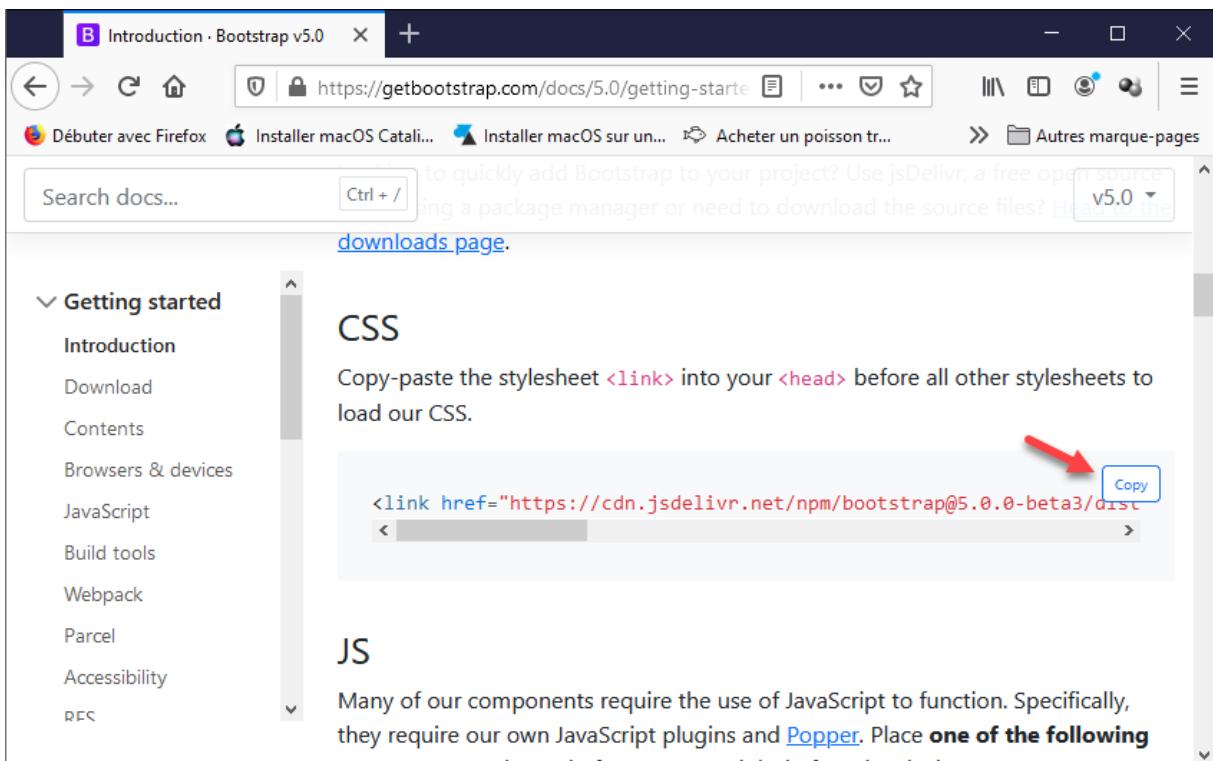
Quickly design and customize responsive mobile-first sites with Bootstrap, the world's most popular front-end open source toolkit, featuring Sass variables and mixins, responsive grid system, extensive prebuilt components, and powerful JavaScript plugins.

[Get started](#) [Download](#)

Currently v5.0.0-beta3 · v4.6.x docs · All releases

https://getbootstrap.com/docs/5.0/getting-started/introduction/

Copiez collez cette URL dans l'en-tête du fichier **index.html** :



Introduction · Bootstrap v5.0

Search docs... Ctrl + /

to quickly add Bootstrap to your project? Use jsDelivr, a free open source package manager or need to download the source files? Use [v5.0](#)

[downloads page.](#)

Getting started

- Introduction
- Download
- Contents
- Browsers & devices
- JavaScript
- Build tools
- Webpack
- Parcel
- Accessibility
- RFCs

CSS

Copy-paste the stylesheet `<link>` into your `<head>` before all other stylesheets to load our CSS.

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-eOJMYsd53ii+sc0/bJGFsiCZc+5NDVN2yr8+0RDqr0Q10h+rP48ckxlpbzKgwra6" crossorigin="anonymous">
```

JS

Many of our components require the use of JavaScript to function. Specifically, they require our own JavaScript plugins and [Popper](#). Place **one of the following**

Voici à quoi devrait ressembler votre fichier **index.html** :

```
<!DOCTYPE html>

<html lang="">

  <head>

    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <link rel="icon" href="<%= BASE_URL %>/favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
    <script src="https://unpkg.com/axios/dist/axios.min.js">
    </script>

    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.0-beta3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-eOJMYsd53ii+sc0/bJGFsiCZc+5NDVN2yr8+0RDqr0Q10h+rP48ckxlpbzKgwra6" crossorigin="anonymous">

  </head>
```

```

<body>
  <noscript>
    <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't work properly without JavaScript enabled. Please enable it to continue.</strong>
  </noscript>
  <div id="app"></div>
  <!-- built files will be auto injected -->
</body>
</html>

```

Le fichier **App.vue** se contente d'appeler le composant **Users** :

```

<template>
  <h1>Liste des inscrits</h1>
  <users></users>
</template>

<script>
  import Users from './components/Users.vue'
  export default {
    name : 'App',
    components : {
      Users
    }
  }
</script>

<style>
h1 {
  text-align : center;
  margin-bottom : 4rem;
}
</style>

```

Axios permet de faire des requêtes XMLHttpRequest. Il est basé sur les promises. Vous utiliserez donc la fonction **then()** pour récupérer la réponse :

```
this.axios.get(adresse).then(utilisateurs => {
  console.log(utilisateurs);
}) ;
```

Où :

- **adresse** est l'adresse qui délivre les données. Ici, <https://randomuser.me/api/?results=10>
- **utilisateurs** est la réponse de l'API

La documentation sur Axios se trouve ici :

<https://github.com/axios/axios>

Le gros du travail se fera dans le composant **Users**.

Vous allez utiliser le hook **created** pour lire les données retournées par l'API :

```
<script>
export default {
  name : 'Users',
  data() {
    return {
      donnees : []
    }
  },
  created() {
    this.axios
      .get('https://randomuser.me/api/?results=10')
      .then(utilisateurs => {
        this.donnees = utilisateurs.data.results;
      })
  }
}
```

```
};  
</script>
```

Attention : dans le hook **created**, on utilise la **notation ES6** pour faciliter l'écriture. En effet, en ES6, **this** s'accorde au contexte lexical. C'est-à-dire à celui de **created**. L'objet **this** donne donc accès à la propriété **donnees**.

Dans le template, insérez un tableau Bootstrap. Pour cela, sélectionnez **Content/Tables** sur le site de Bootstrap et copiez-collez le code du premier tableau dans le template :

The screenshot shows a Firefox browser window with the URL <https://getbootstrap.com/docs/5.0/content/tables/>. The sidebar on the left has a tree view with categories like 'Getting started', 'Customize', 'Layout', 'Content' (which is expanded), 'Tables' (highlighted with a red arrow), 'Figures', 'Forms', 'Components', 'Helpers', 'Utilities', 'Extend', and 'About'. Below these is a 'Migration' section. The main content area displays the HTML code for a table, which is enclosed in a red box. A 'Copy' button is located in the top right corner of this red box. The page title is 'Tables - Bootstrap v5.0'.

```
<table class="table">  
  <thead>  
    <tr>  
      <th scope="col">#</th>  
      <th scope="col">First</th>  
      <th scope="col">Last</th>  
      <th scope="col">Handle</th>  
    </tr>  
  </thead>  
  <tbody>  
    <tr>  
      <th scope="row">1</th>  
      <td>Mark</td>  
      <td>Otto</td>  
      <td>@mdo</td>  
    </tr>  
    <tr>  
      <th scope="row">2</th>  
      <td>Jacob</td>  
      <td>Thornton</td>  
      <td>@fat</td>  
    </tr>  
    <tr>  
      <th scope="row">3</th>  
      <td colspan="2">Larry the Bird</td>  
      <td>@twitter</td>  
    </tr>  
  </tbody>  
</table>
```

Modifiez le contenu des cellules de titre et des cellules de données pour obtenir l'effet désiré. Ici, le tableau **donnees** contient toutes les données des utilisateurs. Une simple boucle **v-for** est utilisée pour extraire le prénom, le nom, le mail et la ville et afficher ces informations dans les cellules du tableau :

Voici ce que vous devez avoir dans le template :

```
<template>
  <table class="table">
    <thead>
      <tr>
        <th>Id</th>
        <th>Nom</th>
        <th>Mail</th>
        <th>Ville</th>
      </tr>
    </thead>
    <tbody>
      <tr v-for="(un, index) in donnees" v-bind :key="index">
        <td>{{ index+1 }}</td>
        <td>{{ un.name.first }} {{ un.name.last }}</td>
        <td>{{ un.email }}</td>
        <td>{{ un.location.city }}</td>
      </tr>
    </tbody>
  </table>
</template>
```

La section **<style></style>** reste vide. La mise en forme du tableau est en effet réalisée par Bootstrap.

Fonctions de rendu (*Render Functions*)

Chaque composant Vue implémente une fonction de rendu. La plupart du temps, cette fonction est créée par le compilateur Vue. Lorsque vous définissez un template dans un composant, son contenu est converti en une fonction de rendu par le compilateur Vue.js.

Cette fonction retourne un DOM virtuel qui est injecté dans le DOM du navigateur pour être rendu. Le DOM virtuel est créé en mémoire et injecté en une seule fois dans le DOM dans un souci de performances.

Les pages qui suivent vont vous montrer comment créer vos propres fonctions de rendu.

Examinez ce code (projet4-01) :

```
<div id="app">
  <art>
    <template v-slot :titre>
      <h1>Titre de l'article</h1>
    </template>
    <p>
      Un simple texte en dehors d'un slot nommé.
    </p>
    <template v-slot :bas>
      <p>Bas de l'article</p>
    </template>
    <p>
      Un autre texte en dehors d'un slot nommé.
    </p>
  </art>
</div>
<script>
  const Art = {
    template : `<div>
      <header><slot name='titre'></slot></header>
      <main><slot></slot></main>
      <footer><slot name='bas'></slot></footer>
    </div>`
  };
  const app = Vue.createApp({
    components : {
      'art' : Art
    }
  );
</script>
```

```
let vm = app.mount('#app');  
</script>
```

Voici le rendu :



D'une façon traditionnelle, les slots nommés **titre** et **bas** sont injectés dans le template, et les deux paragraphes sont injectés dans le slot sans nom.

Dans un composant, il est possible de récupérer :

- Le contenu d'un slot nommé **monslot** avec **this.\$slots.monslot()**
- Le contenu d'un slot sans nom avec **this.\$slots.default()**

Plutôt que de définir un template dans le composant, vous pouvez définir une fonction de rendu avec cette syntaxe :

```
const Art = {  
  render() {  
    return h('tag', {objet}, [enfants]);  
  };  
}  
}
```

Où :

- **tag** est un élément HTML ou un composant,
- **objet** est un objet optionnel qui contient les attributs, props et évènements à utiliser dans le template,
- **enfants** correspond aux enfants du premier argument. Ces enfants sont spécifiés à l'aide de la fonction **h()** dans un simple

tableau. La fonction **h()** retourne un "nœud virtuel" (*Vnode*) qui indique à Vue.js quel type de nœud doit être rendu.

Challenge

Remplacez le template du composant **Art** par la fonction de rendu équivalente.

Vous devez obtenir ce rendu :



Voici le code de départ à utiliser (projet4-01.html) :

```
<div id="app">
  <art>
    <template v-slot :titre>
      <h1>Titre de l'article</h1>
    </template>
    <p>
      Un simple texte en dehors d'un slot nommé.
    </p>
    <template v-slot :bas>
      <p>Bas de l'article</p>
    </template>
    <p>
      Un autre texte en dehors d'un slot nommé.
    </p>
```

```

        </p>
    </art>
</div>
<script>
    var Art = {
        template :
            "<div>\
                <header><slot name='titre'></slot></header>\
                <main><slot></slot></main>\
                <footer><slot name='bas'></slot></footer>\
            </div>"
    };
    var vm = new Vue({
        el : "#app",
        components : { art : Art }
    });
</script>

```

Solution :

La partie HTML reste inchangée :

```

<div id="app">
    <art>
        <template v-slot :titre>
            <h1>Titre de l'article</h1>
        </template>
        <p>
            Un simple texte en dehors d'un slot nommé.
        </p>
        <template v-slot :bas>
            <p>Bas de l'article</p>
        </template>
        <p>
            Un autre texte en dehors d'un slot nommé.
        </p>
    </art>
</div>

```

La balise parente est un `<div></div>`. Les balises enfants sont `<header></header>`, `<main></main>` et `<footer></footer>`. Les slots `titre` et `bas` sont récupérés avec `this.$slots.titre()` et `this.$slots.bas()`. Le slot sans nom est récupéré avec `this.$slots.default()`. Ils sont passés comme troisième argument des trois fonctions `h()` spécifiées dans le troisième argument de la fonction `h()` principale :

```
<script>
  const Art = {
    render() {
      return h('div', {}, [
        h('header', {}, this.$slots.titre()),
        h('main', {}, this.$slots.default()),
        h('footer', {}, this.$slots.bas())
      ])
    }
  };
  const { createApp, h } = Vue;
  const app = createApp({
    components : {
      'art' : Art
    }
  );
  let vm = app.mount('#app');
</script>
```

Templates vs render functions & JSX

Dans la plupart des cas, vous définirez des templates pour construire la partie visuelle de vos composants. Parfois, cependant, il est préférable d'utiliser des fonctions de rendu (*render functions*).

Nous allons raisonner sur un exemple simple.

Supposons que vous définissiez le composant **Titre** qui affiche un titre de niveau `<h1>`, `<h2>` ou `<h3>` en fonction de l'attribut **niveau**. Pour afficher un titre de niveau 1, vous écririez ceci dans l'application :

```
<titre niveau="1">Titre de niveau 1</titre>
```

Pour afficher un titre de niveau 2, vous écririez ceci dans le template de l'application :

```
<titre niveau="2">Titre de niveau 2</titre>
```

Et pour afficher un titre de niveau 3, vous écririez ceci dans le template de l'application :

```
<titre niveau="3">Titre de niveau 3</titre>
```

Voici ce que devraient donner ce code :



Voici le code du composant local **Titre** (projet4-03.html) :

L'attribut **niveau** est récupéré dans la prop **niveau**, et les composants **<slot></slot>** permettent de récupérer ce qui est dans l'innerHTML des composants **<titre></titre>**.

Comme vous pouvez le constater, le code est très répétitif.

```
const Titre = {
  template : `<div>
    <h1 v-if='niveau==1'>
      <slot></slot>
    </h1>
    <h2 v-else-if="niveau==2">
      <slot></slot>
    </h2>
    <h3 v-else-if="niveau==3">
      <slot></slot>
    </h3>
  </div>`,
```

```
    props : ['niveau']
}
```

L'application est très simple. Elle cible la balise d'id **app** et utilise le composant **<titre></titre>**

Voici le code de l'application :

```
const app = Vue.createApp({
  components : {
    'titre' : Titre
  }
});
let vm = app.mount('#app');
```

Challenge

Simplifiez le code du composant **Titre** en utilisant une fonction de rendu. Le résultat doit être le même qu'avec un template.



Solution (projet4-04) :

```
<div id="app">
  <titre niveau="1">Titre de niveau 1</titre>
  <titre niveau="2">Titre de niveau 2</titre>
  <titre niveau="3">Titre de niveau 3</titre>
</div>
<script>
```

```

const { createApp, h } = Vue;
const Titre = {
  render() {
    return h('h' + this.niveau, {}, this.$slots.default())
  },
  props: ['niveau']
}
const app = Vue.createApp({
  components: {
    'titre': Titre
  }
});
let vm = app.mount('#app');
</script>

```

Composants fonctionnels

Les composants fonctionnels sont une alternative sans état des composants traditionnels. Ils sont rendus sans créer d'instance, en contournant le cycle de vie habituel des composants.

Dans cette formation, nous allons voir comment créer des composants fonctionnels en utilisant la fonction `render()`.

Créer un composant fonctionnel

Pour créer un composant fonctionnel, utilisez une simple fonction JavaScript et passez-lui deux arguments :

```
const ComposantFonctionnel = (props, context) => {
}
```

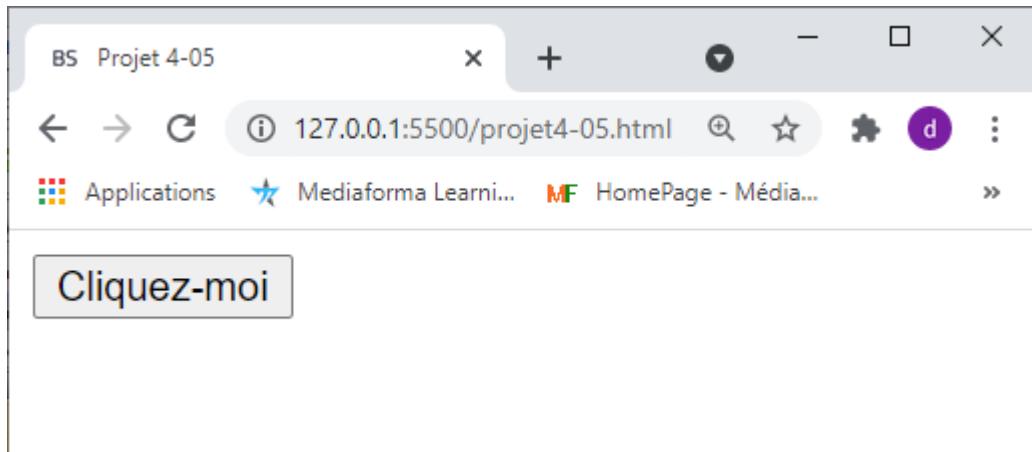
Où :

- `props` sont les props à utiliser dans le composant fonctionnel,
- `context` contient trois propriétés : `attrs`, `emit` et `slots`. Il s'agit des équivalents des propriétés d'instance `$attrs`, `$emit` et `$slots`.

Un composant fonctionnel retourne un "nœud virtuel" (*Vnode*) qui indique à Vue.js quel type de nœud doit être rendu :

```
const ComposantFonctionnel = (props, context) => {
    return h('balise', context.attrs, context.slots);
}
```

Voici le code d'un premier composant fonctionnel très simple qui affiche un bouton "**Cliquez-moi**" (projet4-05.html) :



```
<div id="app">
    <bouton></bouton>
</div>
<script>
    const { createApp, h } = Vue;
    const Bouton = (props, context) => {
        return h('button', 'Cliquez-moi')
    }
    const app = Vue.createApp({
        components : {
            'bouton' : Bouton
        }
    });
    let vm = app.mount('#app');
</script>
```

Utilisation des paramètres props et context

Voyons maintenant comment utiliser les paramètres **props** et **context** passés au composant fonctionnel.

```
const Titre = (props, context) => {
    return h('tag', context.attrs, context.slots);
}
```

Où :

- `context.attrs` fait référence aux attributs, props et évènements à utiliser dans le template
- `context.slots` correspond aux slots utilisés dans le template.

À titre d'exemple, nous allons transformer le composant **Titre** ci-dessous (projet4-04.html) en un composant fonctionnel (sans utiliser **this** ni **\$slots**).

```
<div id="app">
    <titre niveau="1">Titre de niveau 1</titre>
    <titre niveau="2">Titre de niveau 2</titre>
    <titre niveau="3">Titre de niveau 3</titre>
</div>
<script>
    const { createApp, h } = Vue;
    const Titre = {
        render() {
            return h('h' + this.niveau, {}, this.$slots.default())
        },
        props : ['niveau']
    }
    const app = Vue.createApp({
        components : {
            'titre' : Titre
        }
    });
    let vm = app.mount('#app');
</script>
```

Solution (projet4-06.html) :

```
<div id="app">
    <titre niveau="1">Titre de niveau 1</titre>
```

```

<titre niveau="2">Titre de niveau 2</titre>
<titre niveau="3">Titre de niveau 3</titre>
</div>
<script>
    const { createApp, h } = Vue;
    const Titre = (props, context) => {
        return h('h'+props.niveau, context.attrs, context.slots)
    }
    const app = Vue.createApp({
        components : {
            'titre' : Titre
        }
    });
    let vm = app.mount('#app');
</script>

```

Le rendu est le même :



Supposons maintenant que vous vouliez effectuer une action lorsque l'utilisateur clique sur un des titres. Pour cela, remplacez le deuxième paramètre par :

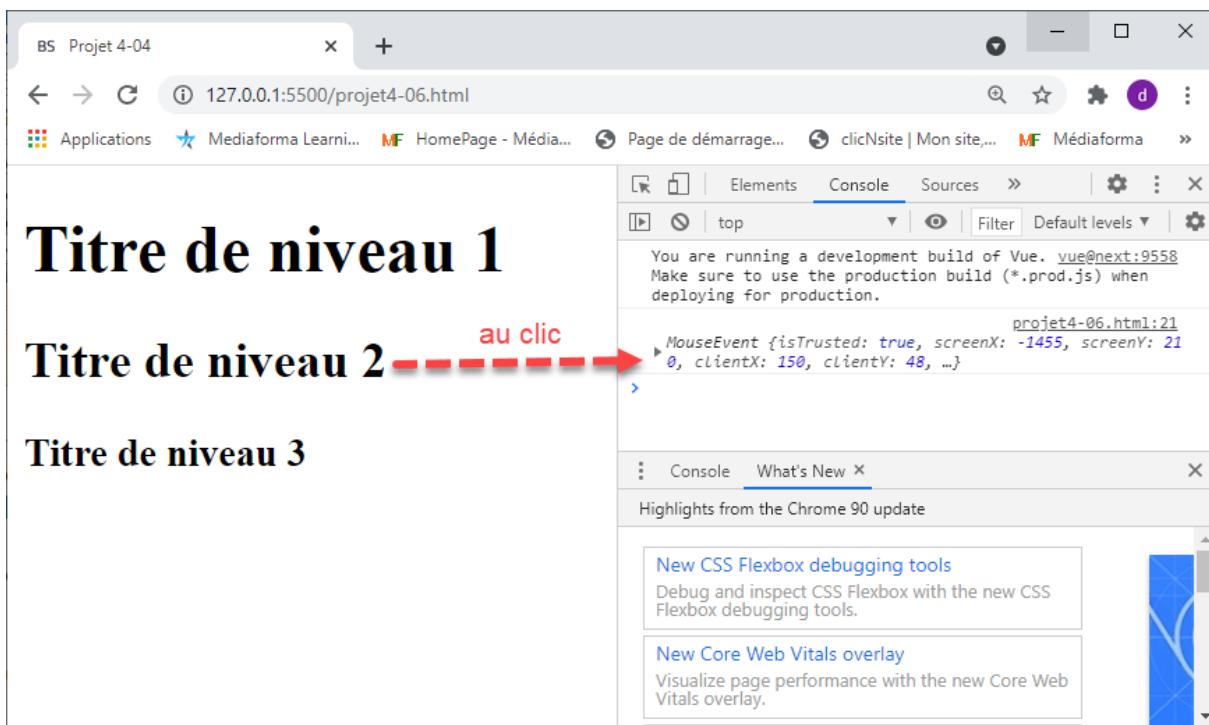
```
{ onClick : (event) => console.log(event) }
```

Pour réagir aux évènements **click**, on utilise la prop **onClick**. Par extension, pour réagir aux évènements **ev**, vous utiliserez la prop **onEv**.

Ce qui donne :

```
const Titre = (props, context) => {  
  return h('h'+props.niveau, { onClick : (event) =>  
    console.log(event) }, context.slots)  
}
```

Et voici le résultat :



Partie 5 - Le gestionnaire d'état Vuex

Vuex est un gestionnaire d'état (*state management pattern*). Il sert de zone de stockage de données centralisée pour tous les composants dans une application. Il fournit des méthodes permettant aux composants de l'application d'accéder à ces données.

Vuex a un gros avantage : toutes les transactions sont enregistrées.

La documentation en ligne sur Vuex se trouve ici : <https://vuex.vue.js.org/>

Installation de Vuex

Lorsque vous codez en Vue.js 3, vous devez utiliser la version 4 de Vuex. Pour cela, insérez cette ligne dans le **<head></head>** de votre application :

```
<script src="https://unpkg.com/vuex@4"></script>
```

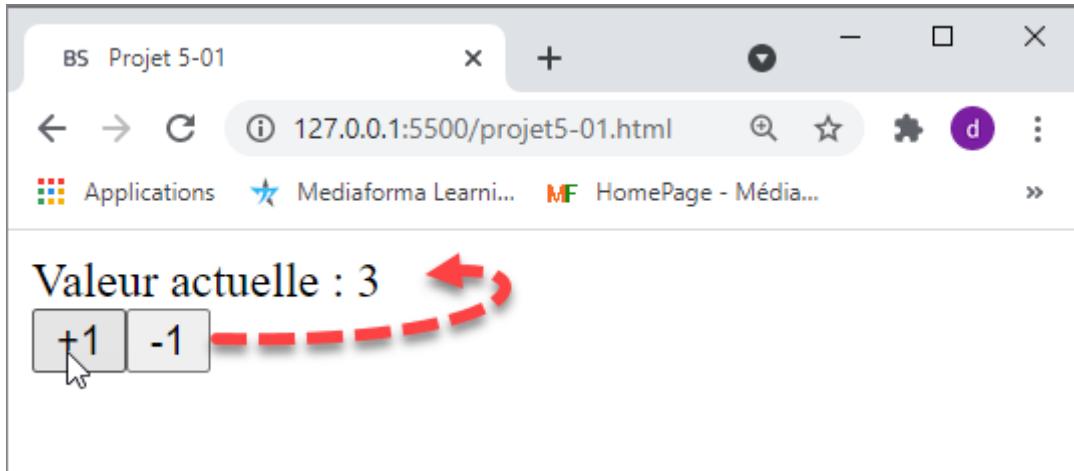
Si vous préférez, vous pouvez également télécharger le code qui se trouve sur la page <https://unpkg.com/vuex@4>, lui donner (par exemple) le nom **vuex.js** et insérer cette ligne dans le **<head></head>** de votre application :

```
<script src="vuex.js"></script>
```

Une première application Vuex

Nous allons définir une application élémentaire qui stocke une valeur numérique dans un état du store Vuex.

Cette valeur pourra être incrémentée ou décrémentée à l'aide de deux boutons :



Fichier projet5-01.html :

```
<div id="app">
    Valeur actuelle : {{actuel}}<br />
    <button v-on :click="plusUn">+1</button>
    <button v-on :click="moinsUn">-1</button>
</div>
<script>
    const store = new Vuex.Store({
        state : {
            compteur : 0
        },
        mutations : {
            increment(state) {
                state.compteur++;
            },
            decrement(state) {
                state.compteur--;
            }
        }
    });
    const app = Vue.createApp({
        computed : {
            actuel() {
                return store.state.compteur;
            }
        },
    },
```

```

methods : {
    plusUn() {
        store.commit('increment');
    },
    moinsUn() {
        store.commit('decrement');
    }
}
);

app.use(store);
let vm = app.mount('#app');
</script>

```

La valeur calculée **actuel** est affichée au début du code. Lorsqu'ils sont cliqués, les boutons exécutent les fonctions **plusUn()** et **moinsUn()** définies dans le ViewModel.

Le store est créé en implémentant la classe **Vuex.store**. La propriété **compteur** est définie dans l'objet **state**.

Les fonctions **increment()** et **decrement()** agissent sur l'état **compteur** du store. Elles sont définies dans l'objet **mutations**.

La fonction **actuel()** retourne la valeur de l'état **compteur** du store.

Les fonctions **plusUn()** et **moinsUn()** sont appelées quand l'utilisateur clique respectivement sur les boutons **+1** et **-1**. Elles exécutent respectivement les fonction **increment()** et **decrement()** du store pour modifier la valeur de l'état compteur.

Pour terminer, n'oubliez pas d'indiquer que vous voulez utiliser le store avec la fonction **use()** :

```
app.use(store);
```

Accès aux méthodes du store dans les composants

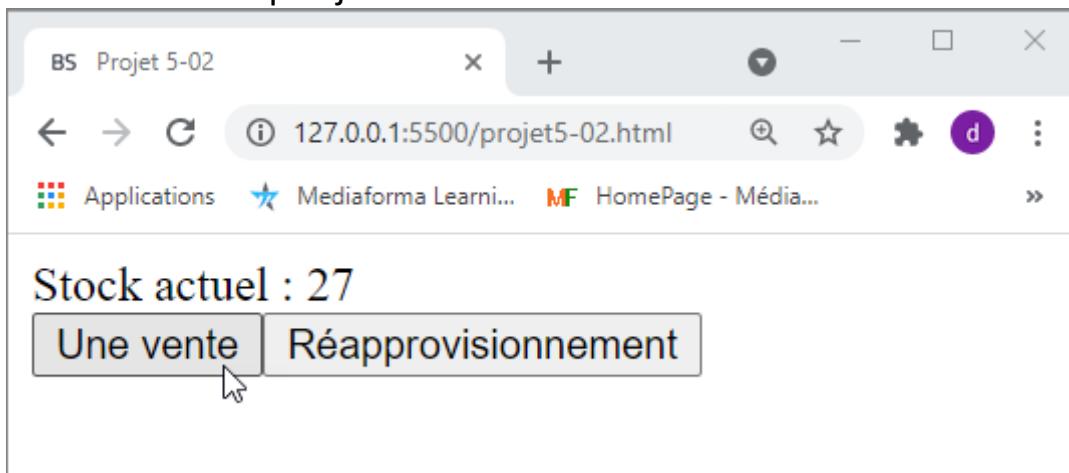
Les méthodes définies dans la propriété **mutations** du store sont accessibles dans le modèle de l'application, mais également dans tous les composants de l'application. Par leur intermédiaire, il est donc possible de modifier les états (*state*) mémorisés dans le store.

Challenge

Un épicier se lance dans la vente de salades. Il utilise Vue.js pour gérer son stock de salades (!) Définissez une application Vue.js qui mémorise le nombre de salades en stock dans un store Vuex. Le nombre de salades au départ est égal à **20**.

Ajoutez deux composants à cette application :

- **Vente** qui affiche le message "**Une salade a été vendue**" et qui soustrait une salade du stock.
- **Reappro** qui affiche le message "**Dix salades ont été ajoutées au stock**" et qui ajoute dix salades au stock.



Solution (projet5-02.html) :

```
<div id="app">
  Stock actuel : {{actuel}}<br>
  <vente></vente>
  <reappro></reappro>
</div>
<script>
  const store = new Vuex.Store({
    state : {
```

```

        salades : 20
    },
    mutations : {
        uneVente(state) {
            state.salades--;
        },
        plusDix(state) {
            state.salades += 10;
        }
    }
});

const Vente = {
    template : "<button @click='vente'>Une vente</button>",
    methods : {
        vente() {
            store.commit("uneVente");
        }
    }
};

const Reappro = {
    template : "<button
@click='reapp'>Réapprovisionnement</button>",
    methods : {
        reapp() {
            store.commit("plusDix");
        }
    }
};

const app = Vue.createApp({
    components : { vente : Vente, reappro : Reappro },
    computed : {
        actuel : function () {
            return store.state.salades;
        }
    }
});
app.use(store);

```

```
let vm = app.mount('#app');  
</script>
```

Accès à l'état du store dans les composants

Reprenez le composant **Vente** défini précédemment :

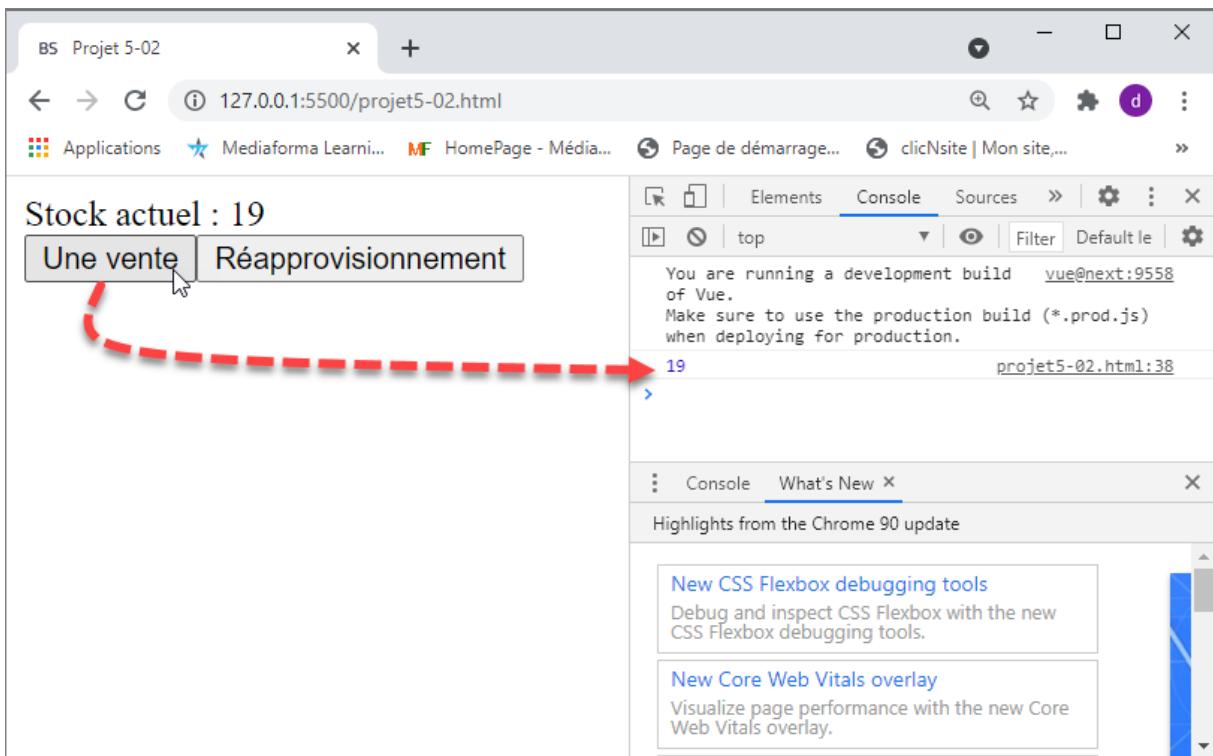
```
const Vente = {  
  template : "<button @click='vente'>Une vente</button>",  
  methods : {  
    vente() {  
      store.commit("uneVente");  
    }  
  }  
};
```

Comment ferez-vous pour afficher la valeur du state **salades** dans ce composant ?

La première idée consiste à insérer l'instruction suivante dans la fonction **vente()** :

```
console.log(store.state.salades);
```

Essayez cette technique :



Une deuxième possibilité vous est offerte.

Tous les composants enfants d'une application Vue.js ont accès au store avec **this.\$store**.

Challenge

Affichez la valeur du state salades dans les composants **Vente** et **Reapro**.

Solution (projet5-03.html) :

```
<div id="app">
  Stock actuel : {{actuel}}<br>
  <vente></vente>
  <reapro></reapro>
</div>
<script>
  const store = new Vuex.Store({
    state : {
      salades : 20
    },
    mutations : {
```

```

        uneVente(state) {
            state.salades--;
        },
        plusDix(state) {
            state.salades += 10;
        }
    }
});

const Vente = {
    template : "<button @click='vente'>Une vente</button>",
    methods : {
        vente() {
            console.log("Avant le commit : " +
this.$store.state.salades);
            store.commit("uneVente");
            console.log("Après le commit : " +
this.$store.state.salades);
        }
    }
};

const Reappro = {
    template : "<button
@click='reapp'>Réapprovisionnement</button>",
    methods : {
        reapp() {
            console.log("Avant le commit : " +
this.$store.state.salades);
            store.commit("plusDix");
            console.log("Après le commit : " +
this.$store.state.salades);
        }
    }
};

const app = Vue.createApp({
    components : { vente : Vente, reappro : Reappro },
    computed : {
        actuel : function () {
            return store.state.salades;
        }
    }
});

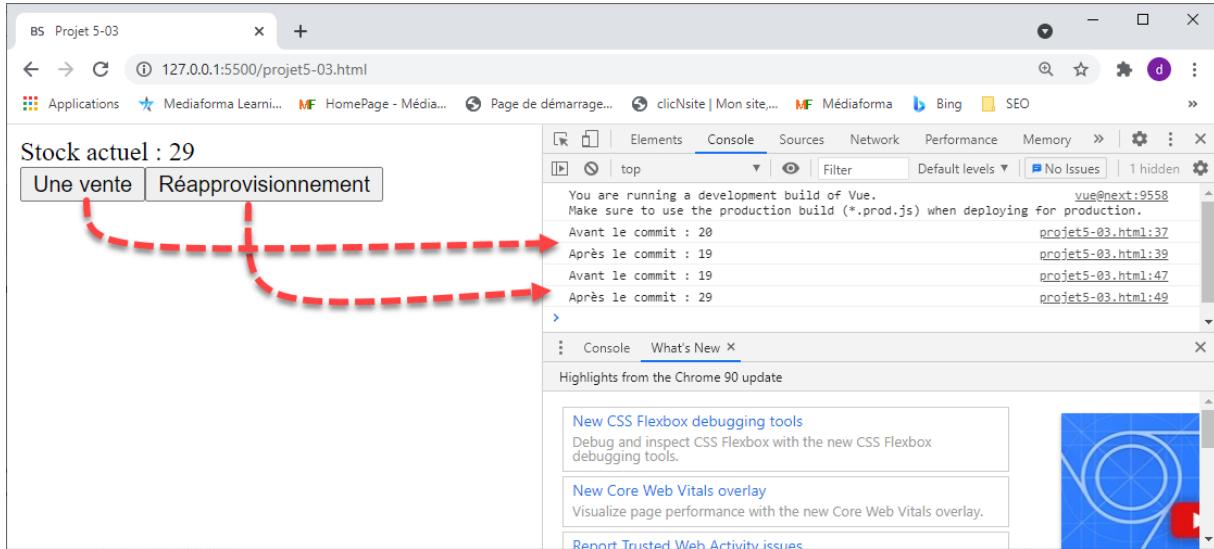
```

```

        }
    }
}) ;
app.use(store);
let vm = app.mount('#app');
</script>

```

Voici le résultat :



L'option **plugins** de Vuex

Le store Vuex a une option **plugins** qui donne accès à des hooks pour chaque mutation :

```
const plug = (store) => {
}
```

A l'intérieur de cette fonction, appelez **store.subscribe()** pour exécuter du code après chaque mutation.

Passez à cette fonction une fonction anonyme avec deux paramètres : **mutation** et **store**.

```
const plug = (store) => {
  store.subscribe(function(mutation, state) {
    // Les objets mutation et state sont accessibles ici
  });
}
```

};

Modifiez le code projet5-02.html comme ceci (projet5-04.html) :

```
<div id="app">
    Stock actuel : {{actuel}}<br>
    <vente></vente>
    <reappro></reappro>
</div>
<script>
    const plug = (store) => {
        console.log("Appelé à l'initialisation du store");
        store.subscribe((mutation, state) => {
            console.log("Appelé à chaque mutation");
            console.log(mutation);
            console.log(state);
        });
    };
    const store = new Vuex.Store({
        plugins : [plug],
        state : {
            salades : 20
        },
        mutations : {
            uneVente(state) {
                state.salades--;
            },
            plusDix(state) {
                state.salades += 10;
            }
        }
    });
    const Vente = {
        template : "<button @click='vente'>Une vente</button>",
        methods : {
            vente() {
                console.log("Avant le commit : " +
this.$store.state.salades);
                store.commit("uneVente");
            }
        }
    };
</script>
```

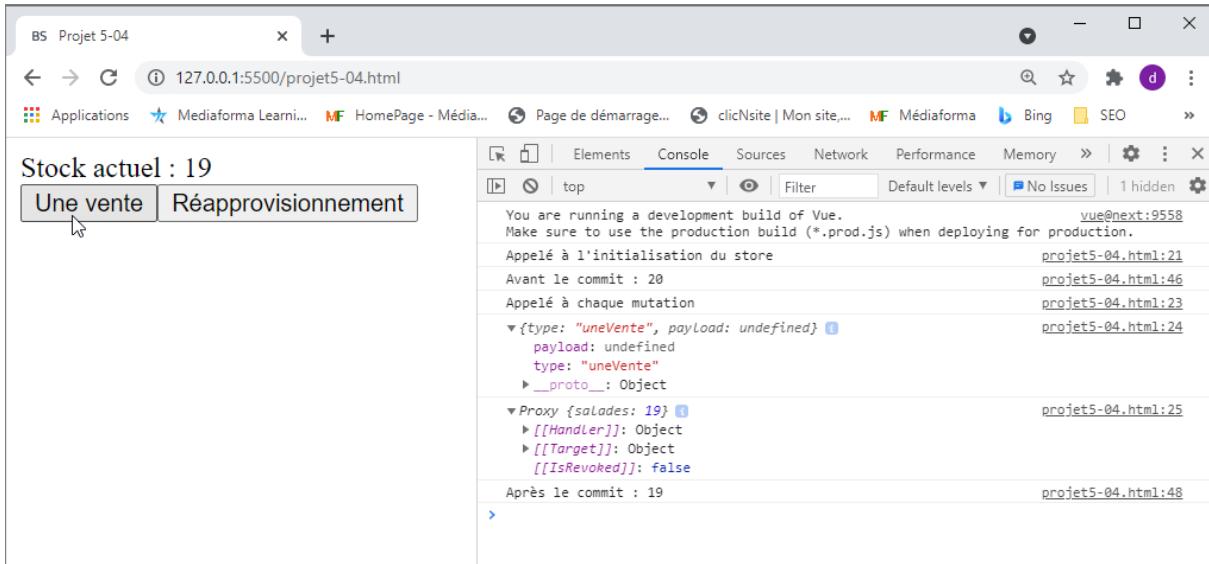
```

        console.log("Après le commit : " +
this.$store.state.salades);
    }
}
};

const Reappro = {
    template : "<button
@click='reapp'>Réapprovisionnement</button>",
    methods : {
        reapp() {
            console.log("Avant le commit : " +
this.$store.state.salades);
            store.commit("plusDix");
            console.log("Après le commit : " +
this.$store.state.salades);
        }
    }
};

const app = Vue.createApp({
    components : { vente : Vente, reappro : Reappro },
    computed : {
        actuel : function () {
            return store.state.salades;
        }
    }
});
app.use(store);
let vm = app.mount('#app');
</script>
```

Voici ce que vous devez obtenir :



Alternative à Vuex

Dans des cas simples, la méthode **reactive()** peut être utilisée comme un store minimaliste accessible aux composants de l'application.

La méthode **reactive()** prend un objet comme argument. Elle retourne le «même» objet, mais **réactif**.

L'objet retourné peut être utilisé directement à l'intérieur des fonctions de rendu et des propriétés calculées.

Chaque mutation déclenche les mises à jour appropriées.

Voici un exemple d'utilisation :

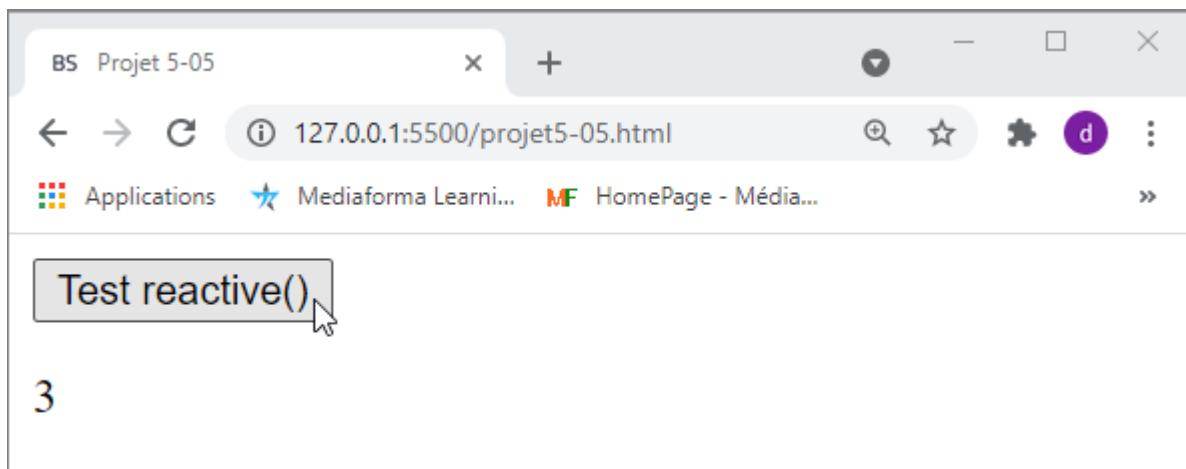
```
<div id="app">
  <button @click="plusUn">Test reactive()</button>
  <p>{{etat.state.count}}</p>
</div>
<script>
  let state = Vue.reactive({ count : 0 });
  const app = Vue.createApp({
    computed : {
      etat() {
        return { state };
      }
    }
  });
  app.mount('#app');
</script>
```

```

        }
    },
    methods : {
        plusUn() {
            {
                state.count++;
            }
        }
    }
);
let vm = app.mount('#app');
</script>

```

Comme vous pouvez le constater, la propriété count est bien mise à jour à chaque clic sur le bouton :



Challenge

Reprenez l'exemple du vendeur de salades en remplaçant le store par un état réactif. Voici le code de départ (projet5-03.html) :

```

<div id="app">
    Stock actuel : {{actuel}}<br>
    <vente></vente>
    <reappro></reappro>
</div>
<script>
    const store = new Vuex.Store({

```

```

state : {
    salades : 20
},
mutations : {
    uneVente(state) {
        state.salades--;
    },
    plusDix(state) {
        state.salades += 10;
    }
}
);

const Vente = {
    template : "<button @click='vente'>Une vente</button>",
    methods : {
        vente() {
            console.log("Avant le commit : " +
this.$store.state.salades);
            store.commit("uneVente");
            console.log("Après le commit : " +
this.$store.state.salades);
        }
    }
};

const Reappro = {
    template : "<button
@click='reapp'>Réapprovisionnement</button>",
    methods : {
        reapp() {
            console.log("Avant le commit : " +
this.$store.state.salades);
            store.commit("plusDix");
            console.log("Après le commit : " +
this.$store.state.salades);
        }
    }
};

const app = Vue.createApp({

```

```

        components : { vente : Vente, reappro : Reappro },
        computed : {
            actuel : function () {
                return store.state.salades;
            }
        }
    });
    app.use(store);
    let vm = app.mount('#app');
</script>

```

Solution (projet5-06.html) :

```

<div id="app">
    Stock actuel : {{actuel.state.salades}}<br>
    <vente></vente>
    <reappro></reappro>
</div>
<script>
    let state = Vue.reactive({ salades : 20 });
    const Vente = {
        template : "<button @click='vente'>Une vente</button>",
        methods : {
            vente() {
                state.salades--;
            }
        }
    };
    const Reappro = {
        template : "<button
@click='reapp'>Réapprovisionnement</button>",
        methods : {
            reapp() {
                state.salades+=10;
            }
        }
    };
    const app = Vue.createApp({
        components : { vente : Vente, reappro : Reappro },

```

```

computed : {
    actuel : function () {
        return {state};
    }
}
});

let vm = app.mount('#app');
</script>

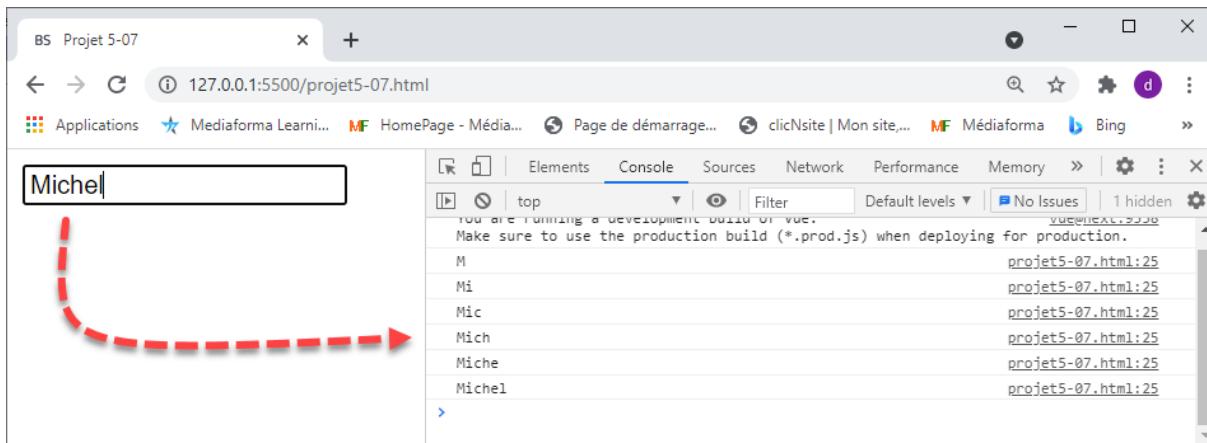
```

Vuex et les formulaires

Vous savez que les données d'une application peuvent être stockées dans le modèle, c'est-à-dire dans la propriété **data** du ViewModel. Si l'application utilise plusieurs composants et que ces composants doivent se partager les données, vous devez passer par **Vuex** (ou par **reactive()** si l'application n'est pas trop complexe).

Les données sont alors mises à jour par des fonctions définies dans la propriété **mutations** du store, ou directement en agissant sur l'objet passé à la fonction **reactive()**.

Supposons que votre application comporte un **<input>** de type **text** et que vous voulez mémoriser le contenu de ce champ dans un store Vuex. Comment feriez-vous ?



Conseil :

Passez la valeur **event.target.value** à la fonction de commit et récupérez cette valeur en définissant un deuxième argument dans le mutateur :

```
this.$store.commit("maj", event.target.value);  
...  
mutations : {  
    maj : function(state, payload) {
```

Solution (projet5-07.html) :

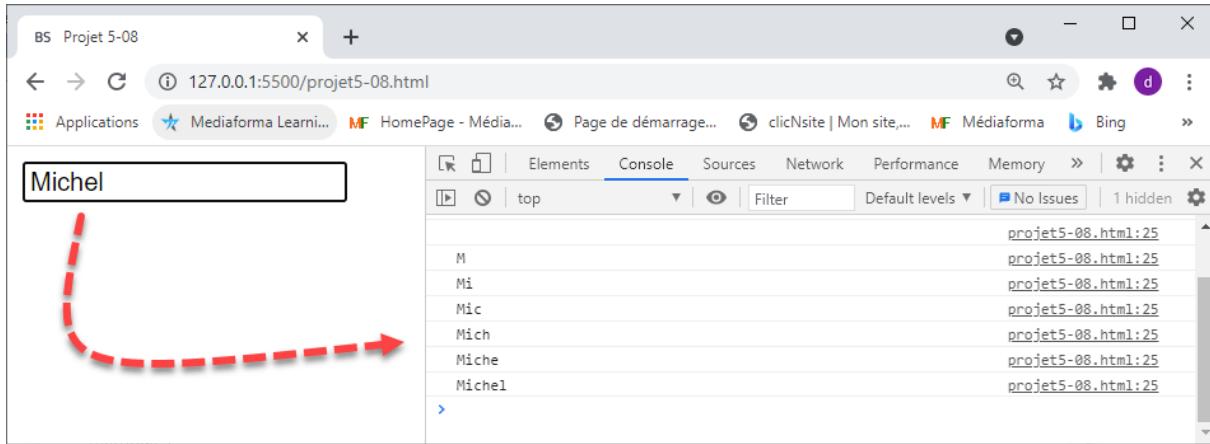
```
<div id="app">  
    <input @input="updateMessage">  
</div>  
<script>  
    const store = new Vuex.Store({  
        state : {  
            message : ''  
        },  
        mutations : {  
            maj : function (state, payload) {  
                state.message = payload;  
                console.log(state.message);  
            }  
        }  
    });  
    const app = Vue.createApp({  
        methods : {  
            updateMessage(event) {  
                this.$store.commit("maj", event.target.value);  
            }  
        }  
    });  
    app.use(store);  
    let vm = app.mount('#app');  
</script>
```

Une autre technique consiste à utiliser un **getter** et un **setter** dans la Vue. Le **setter** effectue un commit en passant par un mutateur dans le store.

Voici le code :

```
<div id="app">
    <input v-model="message">
</div>
<script>
    const store = new Vuex.Store({
        state : {
            texte : ''
        },
        mutations : {
            maj(state, payload) {
                state.message = payload;
                console.log(state.message);
            }
        }
    });
    const app = Vue.createApp({
        computed : {
            message : {
                get() {
                    return this.$store.state.texte;
                },
                set(value) {
                    this.$store.commit("maj", value);
                }
            }
        }
    });
    app.use(store);
    let vm = app.mount('#app');
</script>
```

Le comportement est bien le même qu'avec le code précédent :



Validation d'un formulaire dans Vue.js

La validation de formulaires dans Vue.js est très simple :

- 1) Définissez une balise `<form>` avec une directive `v-on :submit` qui pointe vers une fonction du modèle. Cette fonction retourne `true` si le formulaire est valide.
- 2) Définissez les champs nécessaires en les reliant à des données du modèle avec des directives `v-model`.
- 3) Définissez la fonction de validation dans la propriété `methods` du modèle. Cette fonction retourne `true` si le formulaire est valide. Dans le cas contraire, elle applique la fonction `preventDefault` à l'objet `event` reçu en paramètre.

Vous voyez, rien de bien compliqué et rien de bien différent à la validation d'un formulaire en pur JavaScript.

Challenge

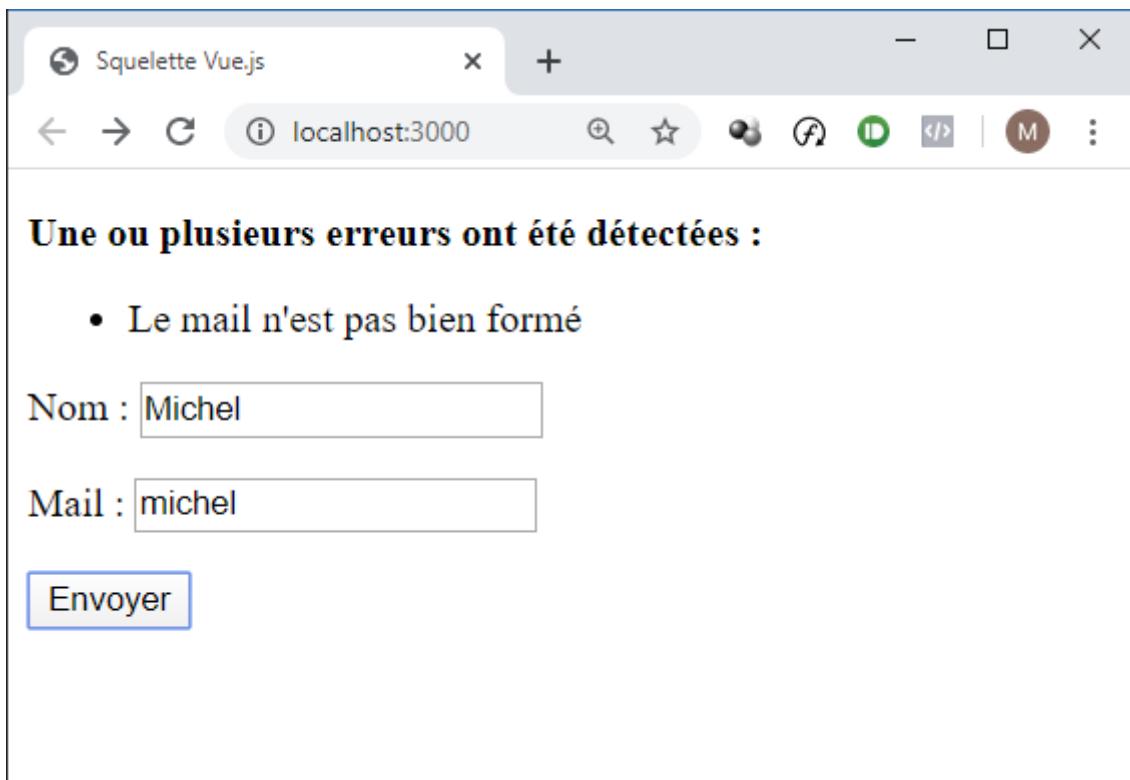
Définissez ce formulaire et le code nécessaire pour obtenir les comportements suivants dans une application Vue.js :

Screenshot of a browser window titled "Squelette Vue.js" showing a simple form. The URL is "localhost:3000". The form contains three fields: "Nom : ", "Mail : ", and a blue "Envoyer" button.

Screenshot of a browser window titled "Squelette Vue.js" showing validation errors. The URL is "localhost:3000". The page displays the message "**Une ou plusieurs erreurs ont été détectées :**" followed by a bulleted list of validation messages:

- Le nom est obligatoire
- Le mail est obligatoire

The form fields are identical to the one above: "Nom : " and "Mail : ", along with a blue "Envoyer" button.



Solution (projet5-09.html) :

```
<div id="app">
  <p v-if="errors.length">
    <strong>Une ou plusieurs erreurs ont été détectées :</strong>
    <ul>
      <li v-for="error in errors">{{error}}</li>
    </ul>
  </p>
  <form @submit="verif" action="https://www.bing.com"
        method="post">
    <p><label for="name">Nom : </label>
       <input type="text" name="name" id="name" v-
model="name">
    </p>
    <p><label for="mail">Mail : </label>
```

```

        <input type="text" name="mail" id="mail" v-
model="mail">

    </p>
    <p><input type="submit" value="Envoyer"></p>

</form>

</div>

<script>

const app = Vue.createApp({


    data() {
        return {
            name : '',
            mail : '',
            errors : []
        }
    },
    methods : {

        valide(mail) {
            var ex = new RegExp(/^[a-z0-9.-]+@[a-z0-9._-]{2,}\.[a-z]{2,8}$/);
            return ex.test(mail);
        },
        verif(ev) {
            this.errors = [];
            if (!this.name)
                this.errors.push('Le nom est obligatoire');
            if (!this.mail)
                this.errors.push('Le mail est obligatoire')
            else if (!this.valide(this.mail))
                this.errors.push('Le mail n\'est pas bien
formé');
        }
    }
});
```

```
        if (this.errors.length == 0)
            return true;
        ev.preventDefault();
    }
}

};

let vm = app.mount('#app');

</script>
```

Partie 6 - Plus loin avec Vue.js

Mixins

Les **mixins** permettent de créer des fonctionnalités réutilisables dans un ou plusieurs composants Vue.

On distingue deux types de mixins :

- **Local** : limité au composant dans lequel il est enregistré.
- **Global** : affecte tous les composants Vue d'une application.

Mixin local

Voici un exemple de mixin local :

```
<script>
    const mixinBonjour = {
        created() {
            this.bonjour();
        },
        methods : {
            bonjour() {
                console.log('Le mixin dit bonjour');
            }
        }
    };
</script>
```

Dans le mixin **mixinBonjour**, le hook **created** est appelé dès la création du mixin. Il affiche un message texte dans la console.

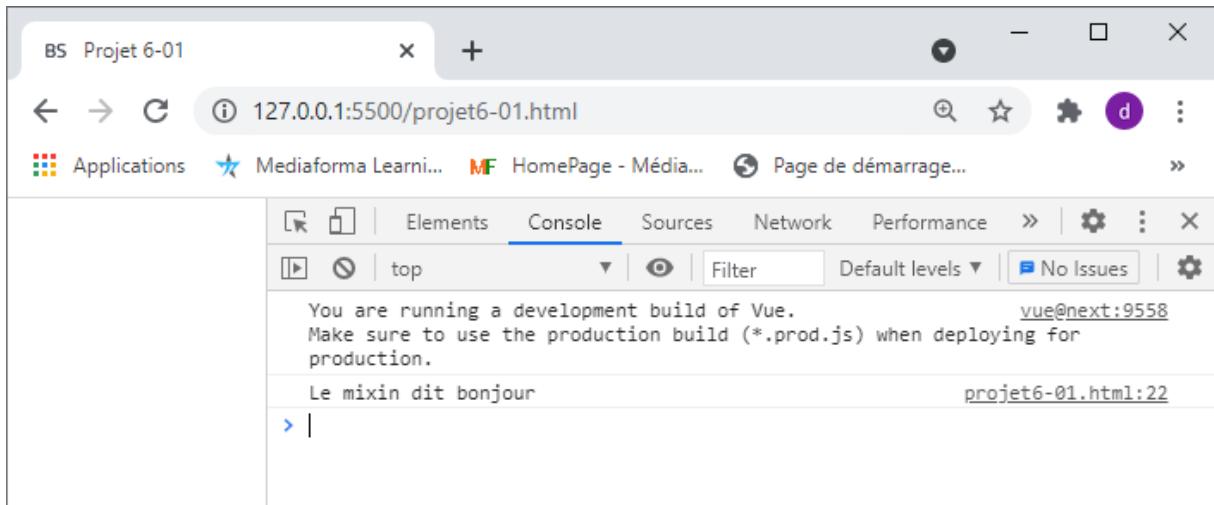
Pour mettre à disposition le mixin **mixinBonjour** dans l'application, il suffit d'y faire référence dans la propriété **mixins** de l'application (projet6-01.html) :

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

```

</div>
<script>
    const mixinBonjour = {
        created() {
            this.bonjour();
        },
        methods : {
            bonjour() {
                console.log('Le mixin dit bonjour');
            }
        }
    };
    const app = Vue.createApp({
        mixins : [mixinBonjour]
    });
    let vm = app.mount('#app');
</script>

```



Vérifions que le mixin est accessible dans les composants de l'application. Nous définissons le composant Test et nous incluons le mixin mixinBonjour dans sa définition :

```

<div id="app">
    <test></test>
</div>
<script>

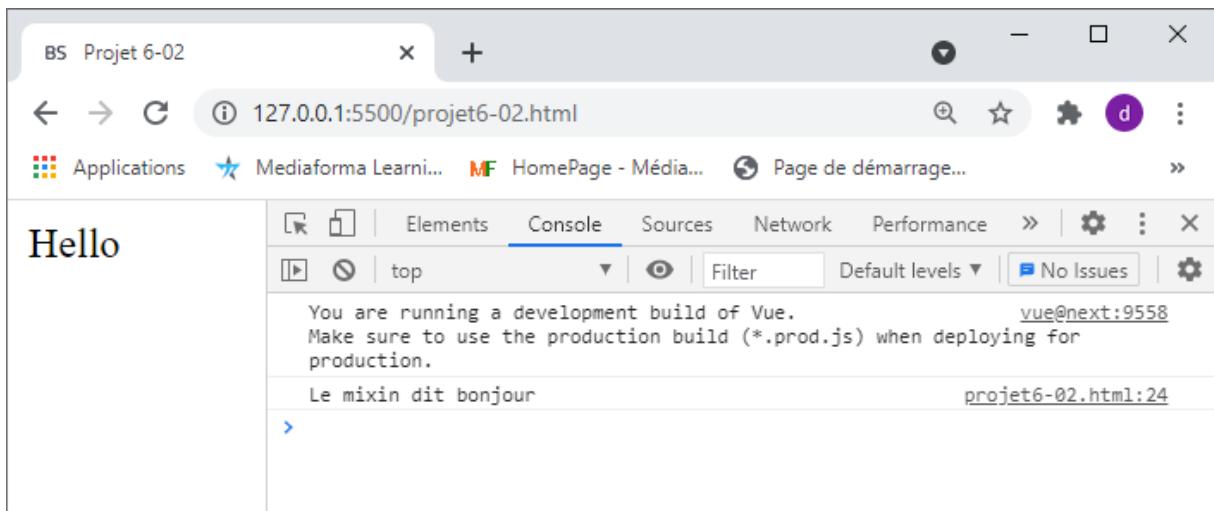
```

```
const mixinBonjour = {
    created() {
        this.bonjour();
    },
    methods : {
        bonjour() {
            console.log('Le mixin dit bonjour');
        }
    }
};

const Test = {
    template : '<div>Hello</div>',
    mixins : [mixinBonjour]
}

const app = Vue.createApp({
    components : {
        test : Test
    }
});
let vm = app.mount('#app');
</script>
```

Voici le résultat (projet6-02.html) :



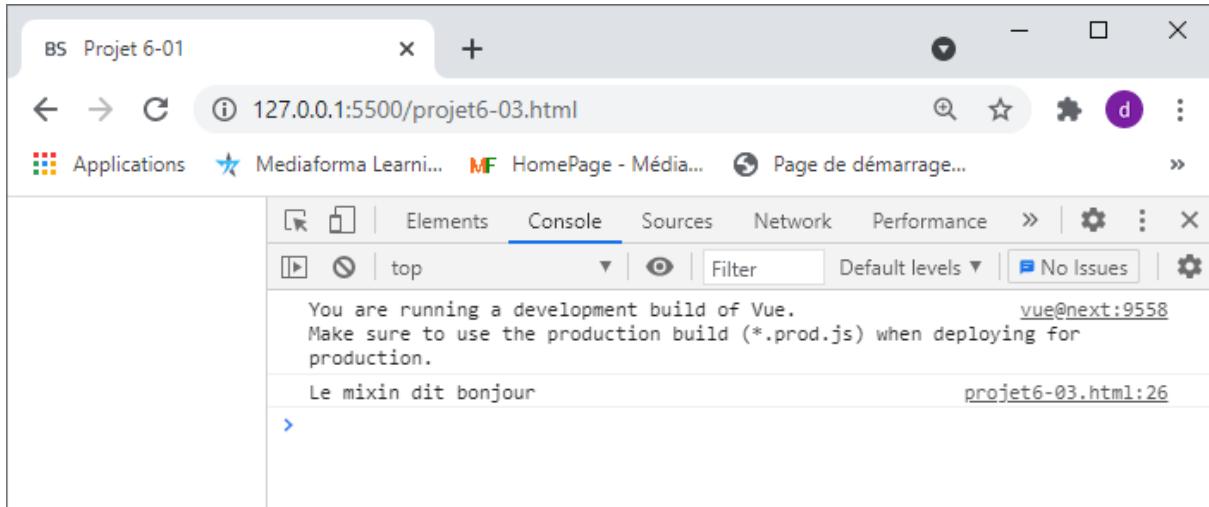
Mixin global

Voici le même exemple, mais cette fois-ci, le mixin est global (projet6-03.html) :

```
<div id="app">
</div>
<script>
    const app = Vue.createApp({
        optionPerso : 'Bonjour'
    });
    app.mixin({
        created() {
            const optionPerso = this.$options.optionPerso;
            this.bonjour(optionPerso);
        },
        methods : {
            bonjour() {
                console.log('Le mixin dit ' + optionPerso);
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

Le hook **created** est appelé dès que la création du mixin global. Il affiche un message texte dans la console. Le simple fait d'instancier la classe Vue met le mixin global à disposition de l'application.

Voici le résultat :



Il n'est pas conseillé d'utiliser des mixins globaux, car ils affectent toutes les instances de Vue d'une application, ... y compris celles des bibliothèques tierces. Cependant, vous pourriez les utiliser pour créer des options personnalisées, comme dans cet exemple.

Plugins

Vue.js peut utiliser des plugins pour étendre ses possibilités. Le code d'un plugin est défini dans un fichier séparé avec la fonction **install()**. Lorsque le plugin est intégré dans une application Vue.js, la fonction **install()** est automatiquement appelée.

Vous avez déjà utilisé les plugins suivants :

- **vue-router** pour gérer le routage de l'application
- **vuex** pour définir une zone de stockage de données centralisée pour tous les composants de l'application

Voici quelques autres plugins :

- **vue-resource** pour réaliser des requêtes AJAX sur une API
- **vue-validator** pour valider vos formulaires

- et beaucoup d'autres encore...

Cette section va vous montrer comment créer un plugin.

Pour créer un plugin, vous devez définir la méthode **install** et lui fournir deux paramètres :

- L'objet app retourné par la fonction `createApp()`.
- L'objet options (cet objet est optionnel) passé par l'utilisateur.

```
const Plugin1 = {
  install(app, options) {
    // Le code du plugin se trouve ici
  }
}
```

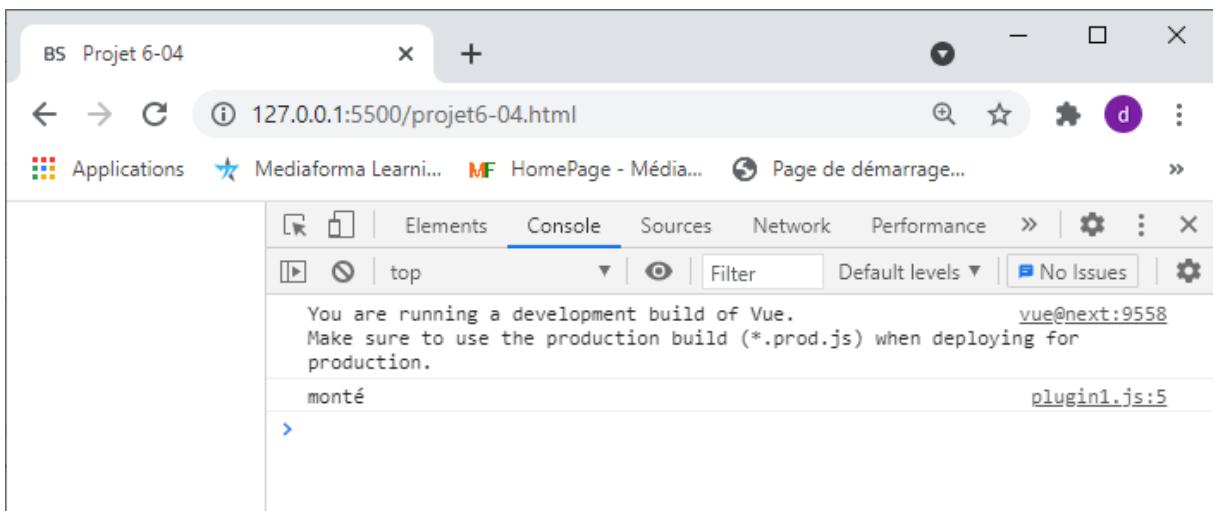
Vous pouvez ajouter au plugin une méthode ou une propriété globale, une ressource, une option ou une méthode globale ou encore une méthode d'instance.

Pour utiliser le plugin **MonPlugin**, ajoutez cette instruction :

```
app.use(MonPlugin);
```

Un premier challenge simple

Définissez le plugin **Plugin1** qui contient un mixin global dans lequel le hook **mounted** affiche la chaîne "**monté**" dans la console chaque fois qu'un composant est monté.



Solution

Voici le code du fichier plugin1.js :

```
const Plugin1 = {
  install : (app) => {
    app.mixin({
      mounted() {
        console.log('monté');
      }
    })
  }
}
```

Et le code de l'application projet6-04.html :

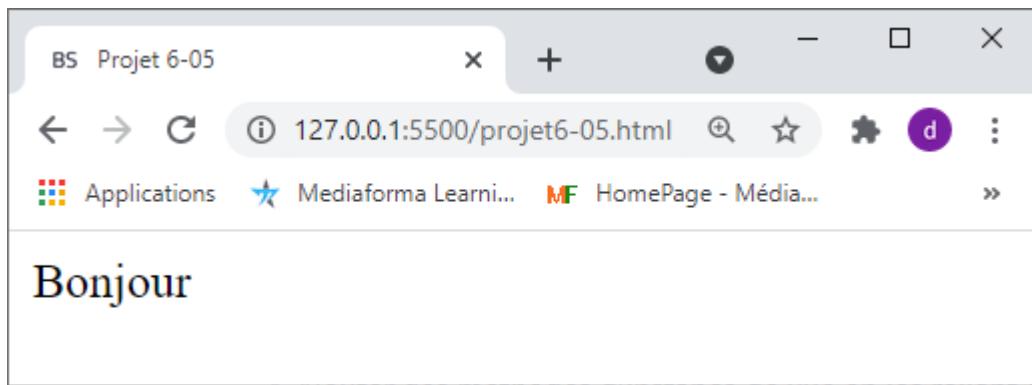
```
<script src="plugin1.js"></script>
...
<div id="app">
</div>
<script>
  const app = Vue.createApp({
  });
  app.use(Plugin1);
  let vm = app.mount('#app');
</script>
```

En utilisant la fonction `use()` appliquée à l'application, on indique qu'on veut utiliser le plugin **Plugin1**. Dès que son hook **mounted** est exécuté, le message "**monté**" s'affiche dans la console.

Un deuxième challenge simple

Définissez le plugin **Plugin2** qui package le composant global **Bonjour**. Ce composant se contente d'afficher "**bonjour**" dans une `<div></div>`.

Utilisez le composant `<bonjour></bonjour>` dans votre application.



Solution

Voici le fichier plugin2.js :

```
const Plugin2 = {
  install : (app) => {
    app.component('bonjour', {
      template : '<div>Bonjour</div>'
    })
  }
}
```

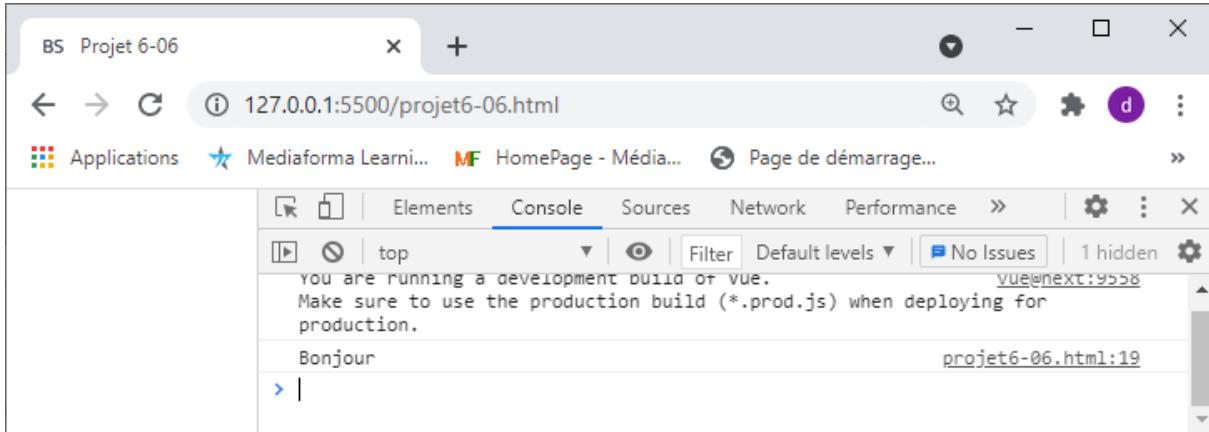
Et le fichier de l'application projet6-05.html :

```
<script src="plugin2.js"></script>
...
<div id="app">
  <bonjour></bonjour>
</div>
<script>
  const app = Vue.createApp({
  });
  app.use(Plugin2);
  let vm = app.mount('#app');
</script>
```

Un troisième challenge simple

Définissez le plugin **MonPlugin** qui ajoute la méthode d'instance **bonjour()**. Cette méthode retourne la chaîne "**Bonjour**". Utilisez ce plugin dans

l'application et appelez la méthode **bonjour()** pour afficher la chaîne "Bonjour" dans la console.



Remarque

Dans Vue.js 2, on utilisait cette syntaxe pour définir une méthode d'instance :

```
Vue.prototype.$maMethode = function(options) {}
```

Dans Vue.js 3, on définit plutôt une propriété globale avec cette syntaxe :

```
app.config.globalProperties.$maMethode = () => {}
```

Solution

Voici le fichier plugin3.js :

```
const Plugin3 = {
  install : (app) => {
    app.config.globalProperties.$bonjour = () => {
      return "Bonjour";
    };
  }
}
```

Et le fichier de l'application projet6-06.html :

```
<script src="plugin3.js"></script>
...
<script>
  const app = Vue.createApp({
    mounted() {
      console.log(this.$bonjour());
    }
  })
  app.mount('#app')
</script>
```

```
    }
  }) ;
app.use(Plugin3) ;
let vm = app.mount('#app') ;
</script>
```

Directives personnalisées

Arrivés à ce point dans la formation, vous savez parfaitement utiliser les directives **v-model**, **v-bind**, **v-on**, etc. Vous allez maintenant apprendre à définir des directives personnalisées, c'est-à-dire qui ne font pas partie des directives de base de Vue.js.

Pour définir une directive personnalisée, utilisez cette instruction :

```
Vue.directive('nom', {});
```

Où **nom** est le nom de la directive sans le "v-".

Entre les accolades, vous pouvez utiliser plusieurs fonctions de hook. Entre autres :

- **beforeMount()** : appelée une fois, à l'initialisation de la directive.
- **mounted()** : appelé une fois, lorsque l'élément lié a été inséré dans son nœud parent.
- **updated()** : appelé après la mise à jour du conteneur.

Les hooks ont accès à ces arguments :

- **el** : élément auquel la directive est liée.
- **binding** : objet qui contient (entre autres) le nom de la directive et les arguments qui lui sont passés.

Directives personnalisées – Un premier exemple

Nous allons définir la directive **v-h1bleu** qui affiche un titre **h1** de couleur **lightblue**. Le texte à afficher est affecté à la directive **v-h1bleu** dans la vue (projet6-07.html) :

```
<div id="app">
    <div v-h1bleu="'Un titre H1 surligné en lightblue'"></div>
</div>
<script>
    const app = Vue.createApp({
        });
        app.directive('h1bleu', {
            beforeMount(el, binding, vnode) {
                el.innerHTML = '<h1>' + binding.value + '</h1>';
                el.style.background = 'lightblue';
            }
        });
        let vm = app.mount('#app');
</script>
```

Remarquez que la chaîne affectée à **v-h1bleu** dans la vue est entourée de **guillemets**(à l'extérieur) et **d'apostrophes** (à l'intérieur). Sans les apostrophes, la valeur passée à **v-h1bleu** ne serait pas considérée comme une chaîne de caractères.



Directive personnalisée avec arguments

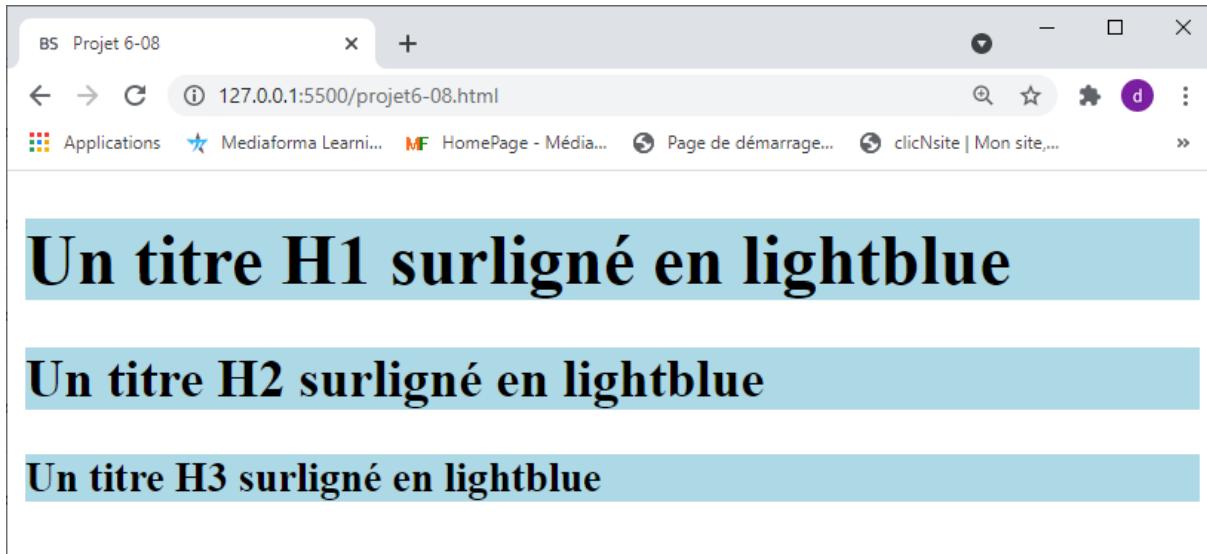
Vous pouvez utiliser des arguments dans une directive personnalisée :

```
<div v-perso :a="message"></div>
```

L'argument **a** est récupéré dans la propriété **binding.arg**.

Challenge

Définissez la directive **v-hbleu**. Passez-lui un niveau de titre en paramètre (**1 à 3**) et affichez le message en conséquence.



Solution (projet6-08.html) :

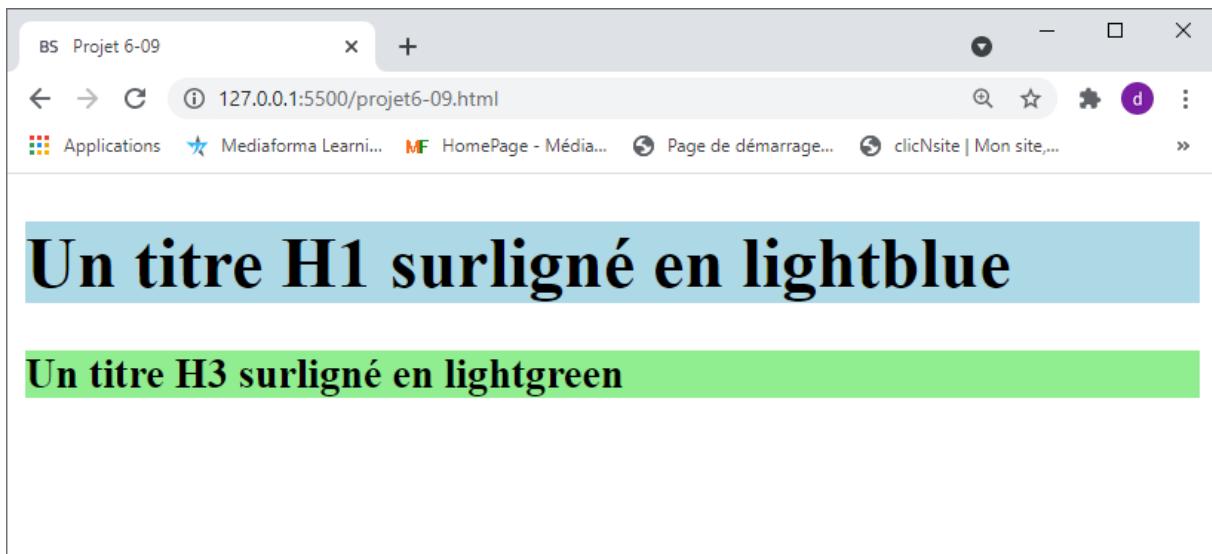
```
<div id="app">
    <div v-hbleu :1="'Un titre H1 surligné en lightblue'"></div>
    <div v-hbleu :2="'Un titre H2 surligné en lightblue'"></div>
    <div v-hbleu :3="'Un titre H3 surligné en lightblue'"></div>
</div>
<script>
    const app = Vue.createApp({
        });
        app.directive('hbleu', {
            beforeMount(el, binding, vnode) {
                el.innerHTML = '<h' + binding.arg + '>' +
                    binding.value +
                    '</h' + binding.arg + '1>';
                el.style.background = 'lightblue';
            }
        });
    
```

```
let vm = app.mount('#app');  
</script>
```

Lorsqu'une directive personnalisée a besoin de plusieurs arguments, vous pouvez lui passer un objet JSON littéral. Les composantes de l'objet se trouvent dans l'objet **binding.value**.

Challenge

Définissez la directive personnalisée **hcouleur**. Affectez-lui un objet JSON qui contient le niveau de titre, la couleur d'arrière-plan et le texte à afficher.



Solution (projet6-09.html) :

```
<div id="app">  
    <div v-hcouleur="{titre : 1, couleur : 'lightblue', texte : 'Un  
titre H1 surligné en lightblue'}"></div>  
    <div v-hcouleur="{titre : 3, couleur : 'lightgreen', texte : 'Un  
titre H3 surligné en lightgreen'}"></div>  
</div>  
<script>  
    const app = Vue.createApp({  
        } );  
        app.directive('hcouleur', {  
            beforeMount(el, binding, vnode) {  
                el.innerHTML = '<h' + binding.value.titre + '>'  
            }  
        })  
    }  
</script>
```

```

        + binding.value.texte
        + '</h' + binding.value.titre +
'1>';
        el.style.background = binding.value.couleur;
    }
}) ;
let vm = app.mount('#app');
</script>

```

Rendre son code robuste grâce aux props typées

Vous savez qu'il est possible de passer des données aux composants enfants avec des props. Jusqu'ici, les props n'ont jamais été typées.

Pour rendre votre code plus robuste, vous pouvez typer les props. Pour cela, remplacez le tableau des **props** par un objet JSON :

Remplacez ce code (projet6-10.html) :

```

<div id="app">
    <taille param="grand"></taille>
</div>
<script>
    const app = Vue.createApp({
    });
    app.component("taille", {
        template : "<div>La taille est : {{param}}</div>",
        props : ["param"]
    });
    let vm = app.mount('#app');
</script>

```

Par celui-ci (projet6-11.html) :

```

<div id="app">
    <taille param="grand"></taille>
</div>

```

```
<script>
  const app = Vue.createApp({
    });
    app.component("taille", {
      template : "<div>La taille est : {{param}}</div>",
      props : {
        param : {
          type : String
        }
      }
    });
    let vm = app.mount('#app');
</script>
```

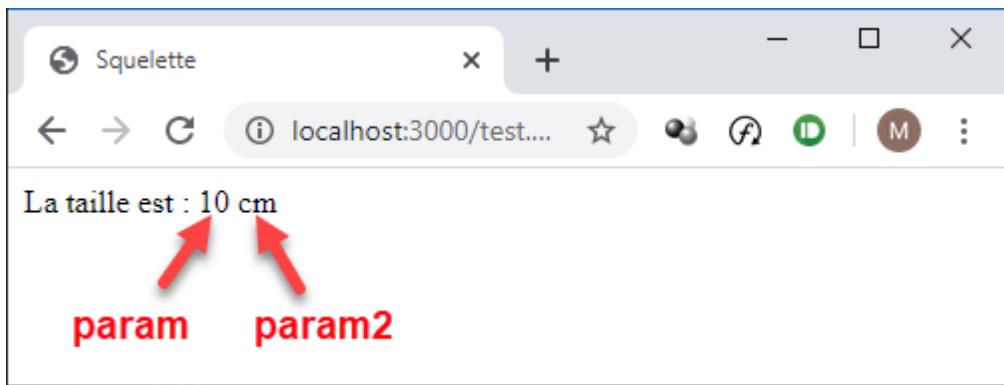
Les props peuvent avoir un des types suivants :

- String
- Number
- Boolean
- Array
- Object
- Date
- Function
- Symbol

Les props passées de façon statique à une instance d'un composant sont toujours de type **String**. Elles peuvent être d'un autre type si vous les bindez avec des données définies dans la propriété **data** du composant.

Challenge

Définissez la prop **param** de type **Number** et la prop **param2** de type **String** dans le composant **taille**. L'affichage doit être le suivant :



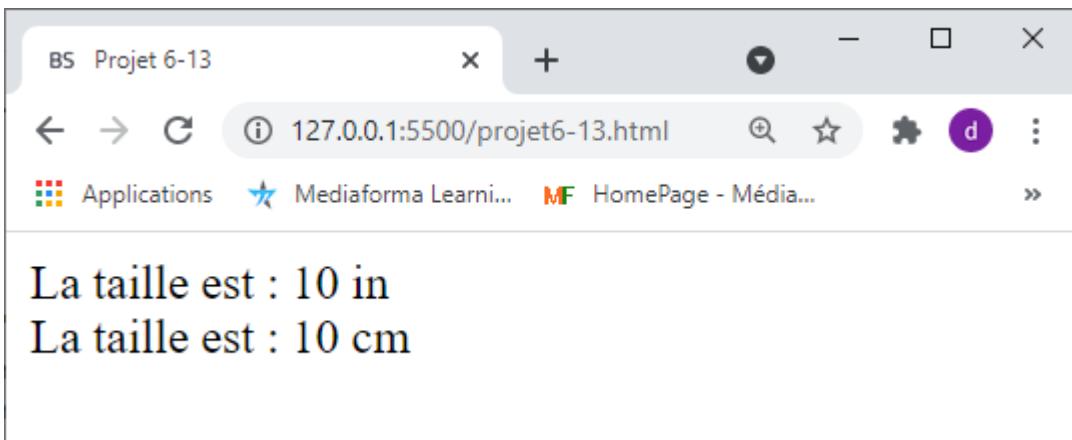
Solution (projet6-12.html) :

Le premier paramètre est obligatoirement bindé au composant car son type est **Number**.

```
<div id="app">
    <taille :param1="valeur" param2="cm"></taille>
</div>
<script>
    const app = Vue.createApp({
        data() {
            return {
                valeur : 10
            }
        }
    });
    app.component("taille", {
        template : "<div>La taille est : {{param1}} {{param2}}</div>",
        props : {
            param1 : {
                type : Number
            },
            param2 : {
                type : String
            }
        }
    });
    let vm = app.mount('#app');
</script>
```

Il est possible de définir la valeur par défaut des propriétés. Voici un exemple (code6-13.html) :

```
<div id="app">
  <taille :param1="valeur"></taille>
  <taille :param1="valeur" param2="cm"></taille>
</div>
<script>
  const app = Vue.createApp({
    data() {
      return {
        valeur : 10
      }
    }
  );
  app.component("taille", {
    template : "<div>La taille est : {{param1}} {{param2}}</div>",
    props : {
      param1 : {
        type : Number
      },
      param2 : {
        type : String,
        default : 'in'
      }
    }
  );
  let vm = app.mount('#app');
</script>
```



Dans le premier composant, la propriété **param2** n'est pas définie dans l'instance. C'est donc la valeur par défaut qui est utilisée. Dans le deuxième composant, la propriété **param2** est définie. Elle surcharge donc la valeur par défaut.

NuxtJS

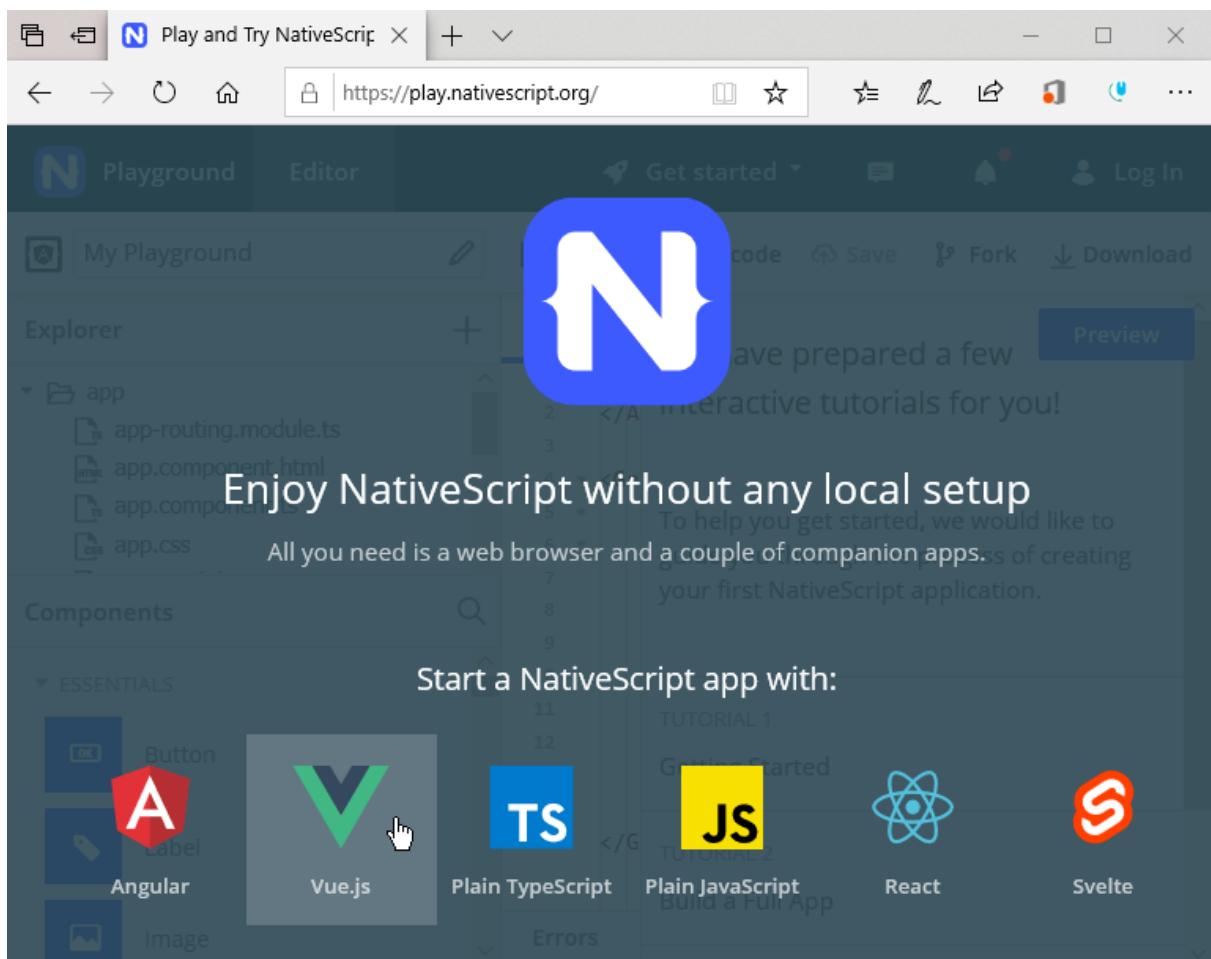
Le framework Nuxt.js permet de créer des applications Vue.js avec un rendu de la page côté serveur (SSR) puis un rendu en SPA côté client. NuxtJS améliore le SEO, car les crawlers ont accès au contenu pré-rendu de vos pages. Il améliore également les performances sur les périphériques d'affichage peu puissants. Alors que j'écris ces lignes, NuxtJS n'est pas encore utilisable avec Vue3. Une section lui sera consacrée lorsque ce sera le cas.

Développement d'applications mobiles

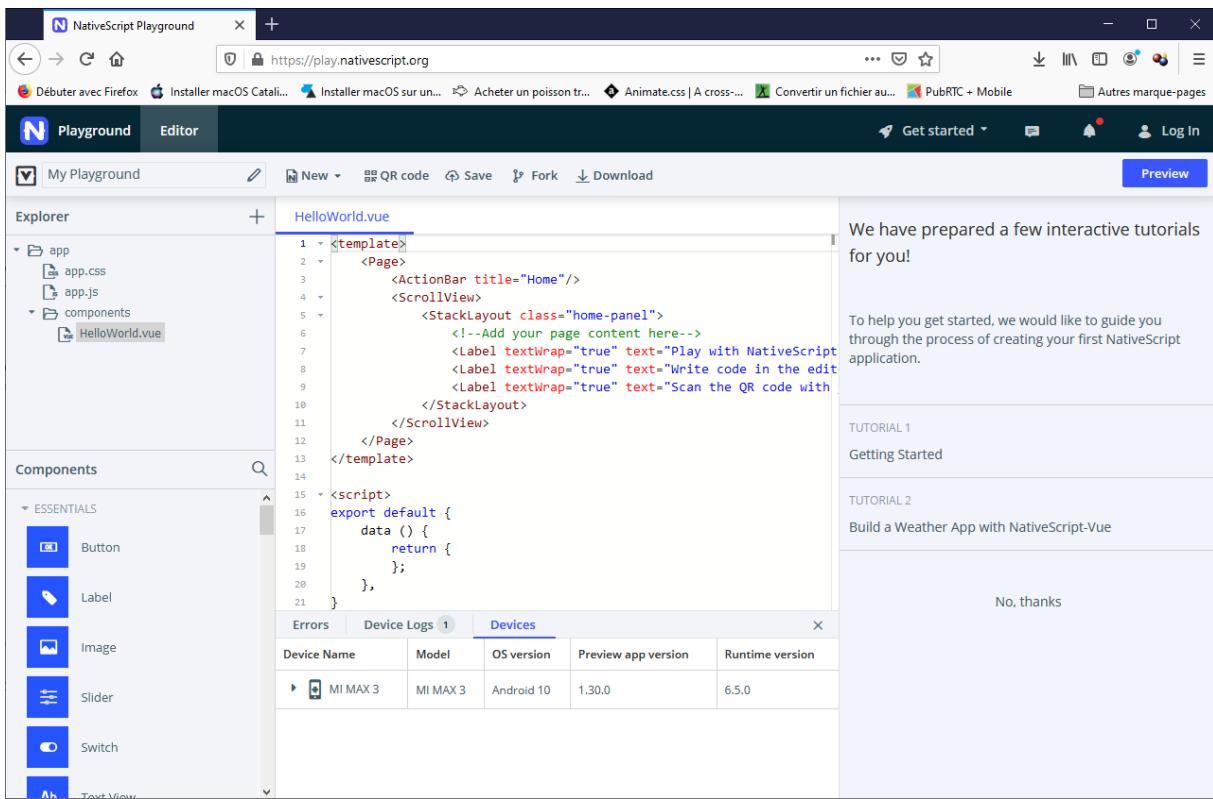
Plusieurs frameworks permettent de créer des applications Vue pour mobiles iOS et Android.

Dans cette formation, nous allons utiliser le framework **VueNative**.

Pour tester facilement ses fonctionnalités, allez sur la page <https://play.nativescript.org/> et cliquez sur l'icône **Vue.js** :



Cet écran va alors s'afficher :



Vous pouvez faire vos premiers pas dans le développement d'applications mobiles sur cette page.

Le volet gauche contient deux zones :

- **Explorer** qui donne accès aux fichiers du projet
- **Components** qui donne accès aux composants préconfigurés utilisables dans l'application

Le volet central contient le code de l'application. Il est entièrement modifiable :

- En glissant-déposant des composants depuis le volet **Components** dans le fichier **HelloWorld.vue**
- En tapant directement des instructions dans le fichier **HelloWorld.vue**

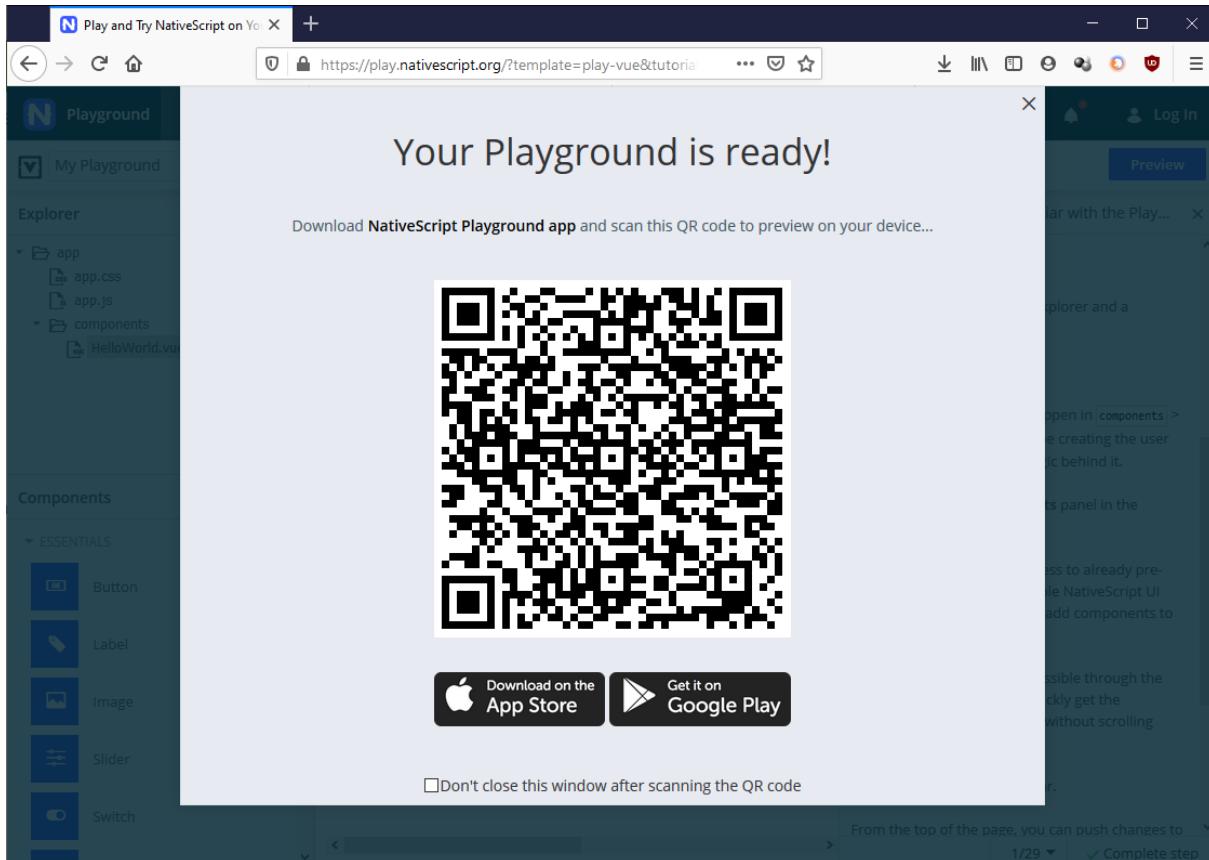
Pour pouvoir tester votre code, vous devrez installer l'application **NativeScript Playground** sur votre téléphone :

- Sur App Store (iOS) :
<https://itunes.apple.com/us/app/nativescript-playground/id1263543946?mt=8&ls=1>
- Sur Google Play (Android) :

<https://play.google.com/store/apps/details?id=org.nativescript.play>

Allez-y, installez l'application qui correspond à votre téléphone.

Cliquez sur le bouton bleu **Preview**, en haut et à droite de la page. Un QR code s'affiche. Scannez ce QR-code avec l'application **NativeScript Playground** :

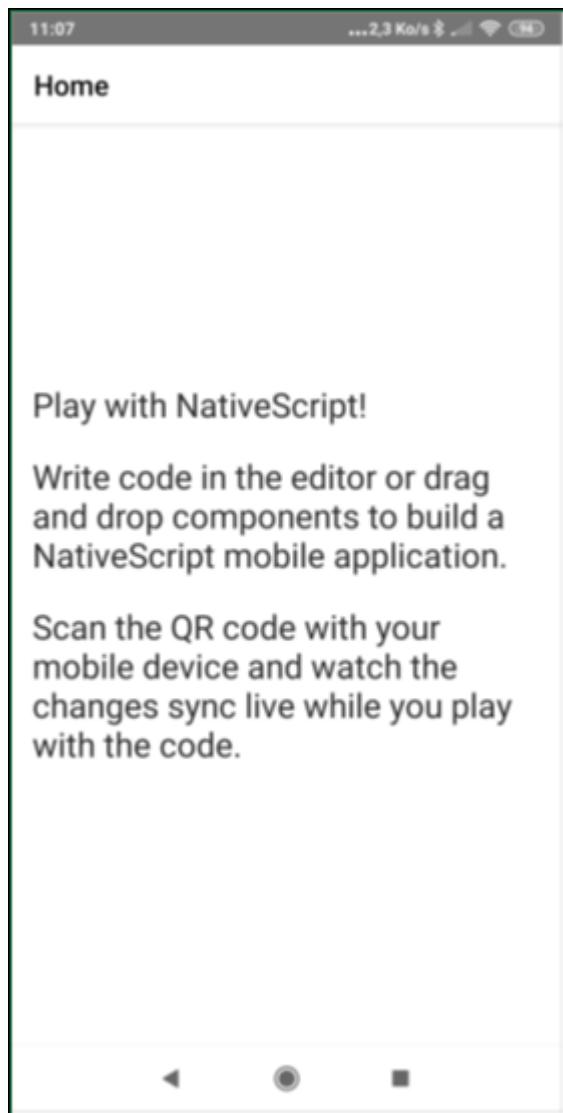


Lorsque vous utilisez pour la première fois l'application **NativeScript Playground**, on vous demande d'installer l'application **NativeScript Preview**.

Procédez à son installation.

Scannez à nouveau le QR Code dans l'application **NativeScript Playground** pour l'exécuter.

Voici ce que vous devriez obtenir (ici sur un téléphone Android) :



Challenge

Supprimez les trois composants **<Label />** dans **HelloWorld.vue** et glissez-déposez un bouton pour les remplacer.

La méthode **onButtonTap()** est automatiquement ajoutée dans la section **<script></script>** pour gérer le clic sur le bouton.

Remplacez l'appel à **console.log()** par un **alert()** et sauvegardez le nouveau code en appuyant sur *Contrôle + S*. L'application est mise à jour sur votre mobile (fichier `vuenative1.html`).

```
<template>
```

```
<Page>
    <ActionBar title="Home" />
    <ScrollView>
        <StackLayout class="home-panel">
            <Button text="Button" @tap="onButtonTap" />
        </StackLayout>
    </ScrollView>
</Page>
</template>
<script>
    export default {
        methods : {
            onButtonTap() {
                alert("Button was pressed");
            }
        },
        data() {
            return {};
        }
    };
</script>
<style scoped>
    .home-panel {
        vertical-align : center;
        font-size : 20;
        margin : 15;
    }
    .description-label {
        margin-bottom : 15;
    }
</style>
```

Challenge

Affichez une image A cliquable.

Lorsque l'utilisateur clique dessus, remplacez-la par une image **B**, également cliquable.

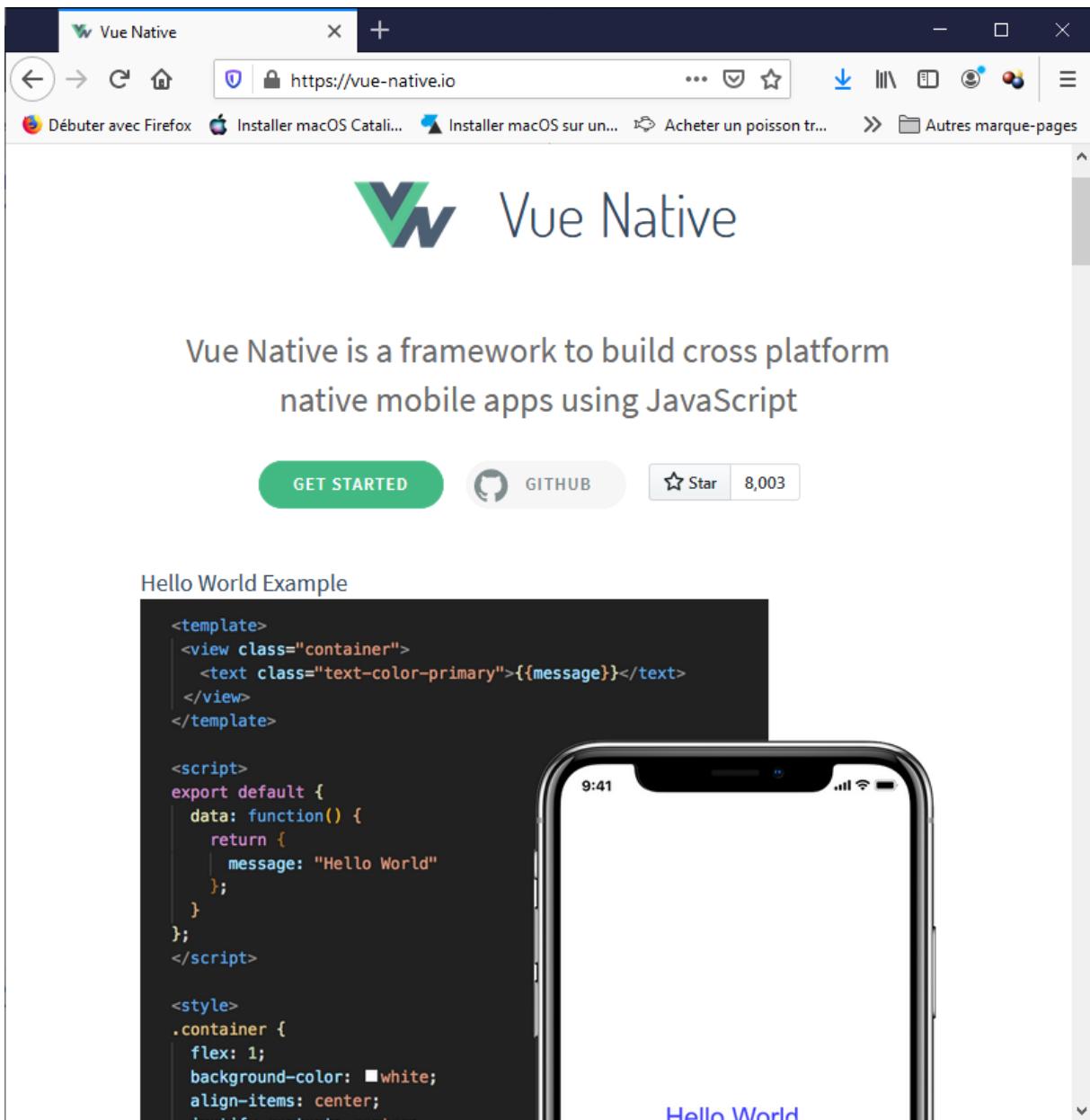
Lorsque l'utilisateur clique dessus, remplacez-la par l'image **A**. Ainsi de suite...

Solution (vuenative2.html) :

```
<template>
    <Page>
        <ActionBar title="Home" />
        <ScrollView>
            <StackLayout class="home-panel">
                <Image :src="im" @tap="change">
            </StackLayout>
        </ScrollView>
    </Page>
</template>
<script>
    export default {
        data() {
            return {
                im :
'https ://static3.depositphotos.com/thumbs/1005412/image/218/218603
8/api_thumb_450.jpg'
            } ;
        },
        methods : {
            change() {
                if (this.im.indexOf("218") != -1) {
                    this.im =
'https ://st2.depositphotos.com/thumbs/2222024/image/5609/56093859/
api_thumb_450.jpg'
                }
                else {
                    this.im =
'https ://static3.depositphotos.com/thumbs/1005412/image/218/218603
8/api_thumb_450.jpg';
                }
            }
        }
    }
</script>
```

```
        }
    };
</script>
<style scoped>
    .home-panel {
        vertical-align : center;
        font-size : 20;
        margin : 15;
    }
    .description-label {
        margin-bottom : 15;
    }
</style>
```

Vous voulez aller plus loin dans le développement d'applications natives iOS et Android avec VueNative ? Je vous suggère de suivre les instructions données sur le site <https://vue-native.io/> :



Le composant <teleport></teleport>

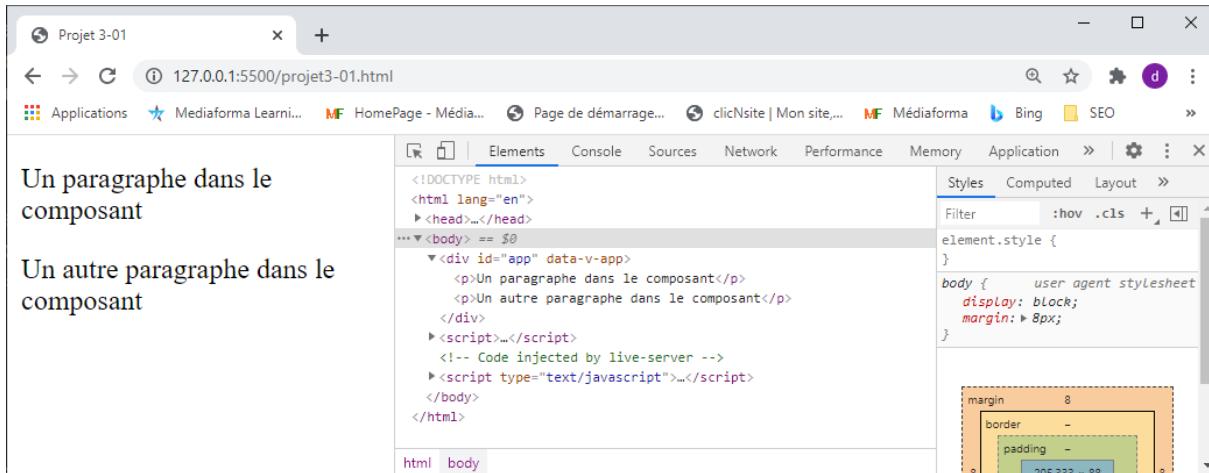
En règle générale, les interfaces utilisateur créées avec Vue.js consistent en un ensemble de composants. Chaque composant embarque les comportements (méthodes) associés. L'application est créée dans une arborescence de composants, en imbriquant les composants les uns dans les autres.

Parfois, il paraît plus logique de déplacer une partie du modèle d'un composant dans une autre partie du DOM (à la racine de l'application par exemple), voire même à l'extérieur de la balise ciblée par Vue.js. Pour cela, vous utiliserez le composant **Teleport**, fourni avec Vue.js 3.

Nous allons examiner une application très simple qui contient un seul composant. Voici le code de départ :

```
<div id="app">  
    <bonjour></bonjour>  
</div>  
  
<script>  
    const app = Vue.createApp({  
        }) ;  
        app.component('bonjour', {  
            template : `<p>Un paragraphe dans le composant</p>  
                      <p>Un autre paragraphe dans le composant</p>`  
        })  
        let vm = app.mount('#app')  
</script>
```

Comme on pouvait s'y attendre, les deux paragraphes définis dans le template du composant sont insérés dans l'application :



Supposons maintenant que l'on veuille "remonter" le deuxième paragraphe à la fin du body. Pour cela, on va utiliser un composant

Teleport en utilisant cette syntaxe :

```
<teleport to="destination">  
</teleport>
```

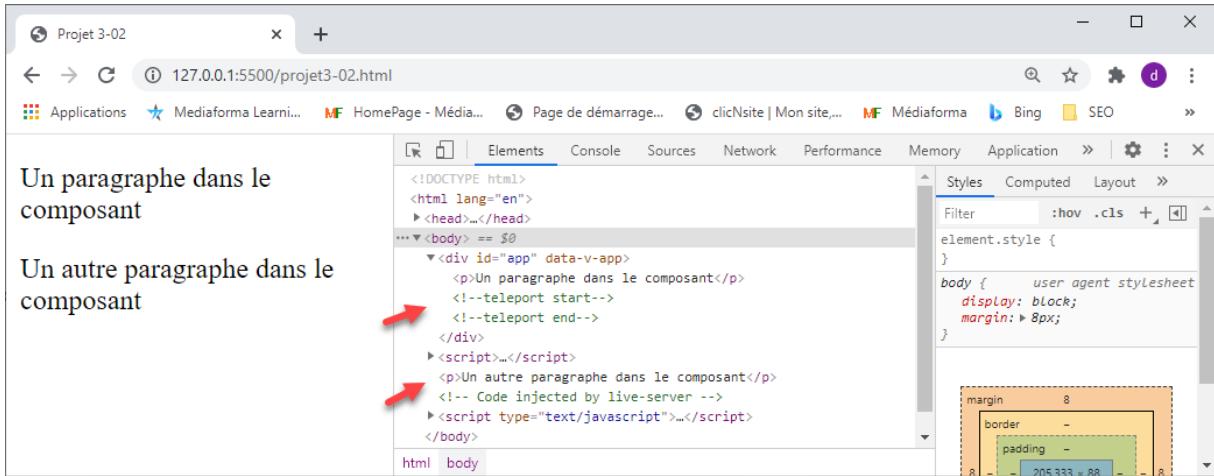
Où **destination** représente un sélecteur CSS quelconque :

- Le nom d'un élément : **<teleport to="body">**
- Un id : **<teleport to="#unId">**
- Une classe : **<teleport to=".uneClasse">**
- Un attribut : **<teleport to="[unAttribut]">**
- Une propriété réactive : **<teleport :to="unePropriétéRéactive">**

Voici le code utilisé :

```
<div id="app">  
  <bonjour></bonjour>  
</div>  
  
<script>  
  const app = Vue.createApp({  
    } );  
  app.component('bonjour', {  
    template : `<p>Un paragraphe dans le composant</p>  
              <teleport to="body">  
                <p>Un autre paragraphe dans le composant</p>  
              </teleport>`  
  })  
  let vm = app.mount('#app');  
</script>
```

Comme vous pouvez le voir, le deuxième paragraphe a été déplacé après la balise de l'application Vue.js, à la fin du body.



La balise `<teleport>` peut avoir un attribut **disabled**. Lorsqu'il vaut **true**, ce qui se trouve entre `<teleport>` et `</teleport>` reste dans le composant. Lorsqu'il vaut **false**, ce qui se trouve entre `<teleport>` et `</teleport>` est déplacé à l'endroit spécifié dans l'attribut **to**.

Définissons la propriété **deplace** et affectons-lui la valeur **true**. Ajoutons un attribut **disabled** bindé à la propriété **deplace**. Enfin, ajoutons un bouton dans le template. Lorsqu'il sera cliqué, la valeur de la propriété **deplace** sera inversée :

```
<div id="app">
  <bonjour></bonjour>
</div>
<script>
  const app = Vue.createApp({
    });
    app.component('bonjour', {
      template : `<p>Un paragraphe dans le composant</p>
      <teleport to="body" :disabled="deplace">
        <p>Un autre paragraphe dans le composant</p>
      </teleport>
      <button
        @click="deplace=!deplace">Change</button>`,
      data() {
        return {
      
```

```

        deplace : true
    }
}

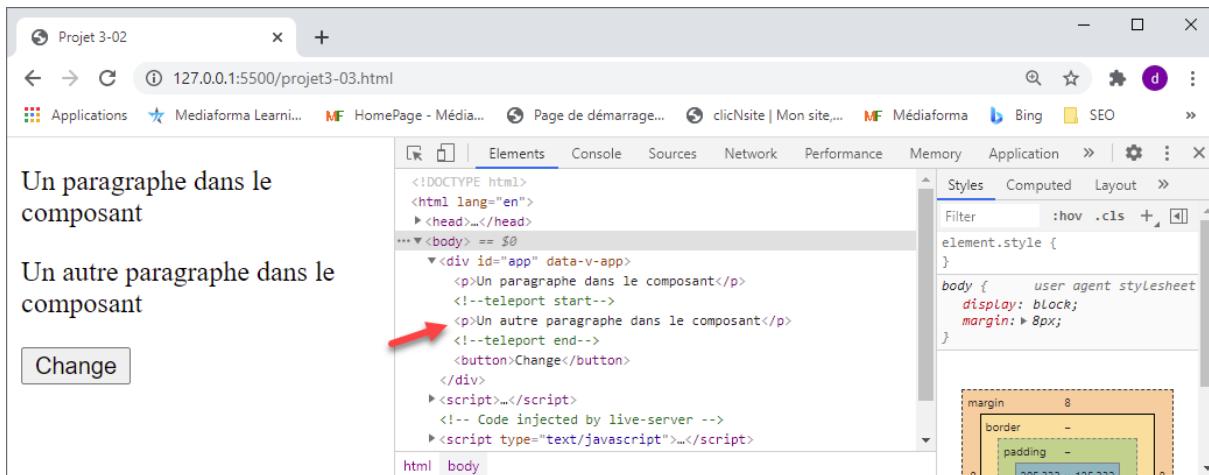
}

let vm = app.mount('#app');

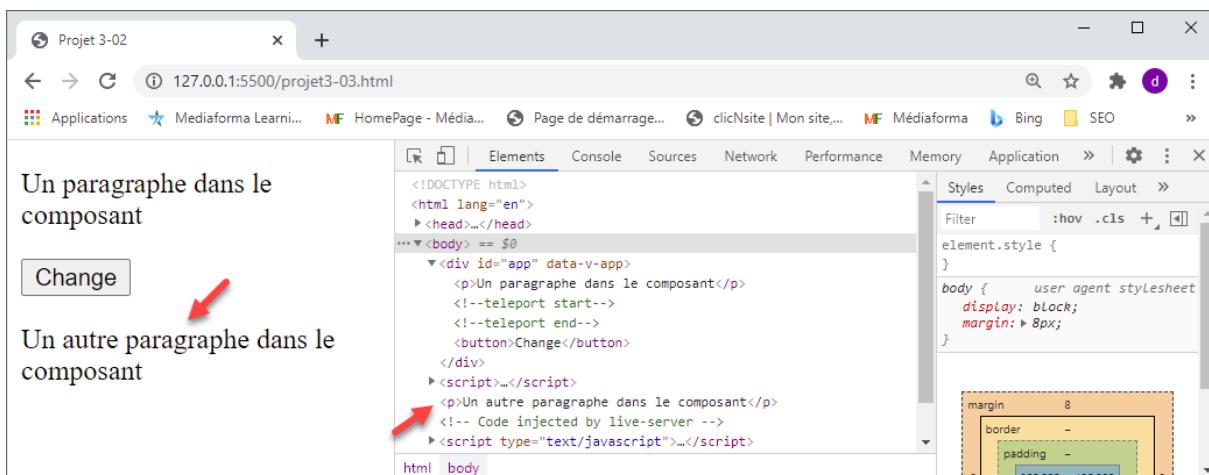
</script>

```

Par défaut, la propriété **deplace** (et donc l'attribut **disabled**) vaut **true**. Le deuxième paragraphe reste donc dans le composant :



Lorsqu'on clique sur le bouton, **deplace** s'inverse. Il vaut donc **false**. Cette valeur est répercutée sur l'attribut **disabled** et le deuxième paragraphe est déplacé à la fin du body :



L'API de composition

Une application Vue.js 2 est implémentée selon un modèle "par options" (*Options API*). Dans cette implémentation, différentes propriétés sont définies dans le modèle :

- **data** qui contient les données initiales ;
- **methods** qui contient des fonctions utilisables dans le template ;
- **computed** qui définit des données calculées mises à jour dynamiquement à partir des variables définies dans **data** et **props** ;
- **props** qui liste les données reçues par les composants ;
- les propriétés liées au cycle de vie du composant : **beforeCreate**, **created**, **beforeMount**, **mounted**, **beforeUpdate**, **updated**, **beforeDestroy** et **destroyed**.

Voici un exemple d'application Vue.js 2 :

```
<script>

var app = new Vue({
  el : '#app',
  data : {
    },
  methods : {
    },
  computed : {
    },
  props : {
    },
  components : {
    }
  })
</script>
```

L'approche **Options API** comporte quelques limitations.

- 1)** Le support du langage TypeScript est très limité dans une application Vue.js 2.
- 2)** Lorsqu'un composant grossit, vous devez répartir le code qui correspond à ses différentes fonctionnalités dans les différentes sections (**data**, **methods**, etc.). Le code devient moins lisible, et donc, moins facile à maintenir. Ceci est d'autant plus le cas lorsqu'un composant doit gérer plusieurs problématiques. Les lignes de codes sont réparties dans les différentes sections, et séparées entre elles par des dizaines ou des centaines de lignes de code.
- 3)** Il est difficile de réutiliser du code dans les différents composants d'une application. Si plusieurs composants doivent avoir des données ou de la logique en commun, vous devez passer par des **mixins**. Vous pouvez :
 - Créer les mixins qui correspondent aux fonctionnalités à réutiliser et inclure les mixins dans le composant qui en a besoin.
 - Créer un **mixin factories** (fonctions qui retournent des versions personnalisées des mixins) et inclure les mixins dans le composant qui en a besoin en précisant un namespace pour chaque fonction.
 - Créer des **scoped slots** et les utiliser dans le composant qui en a besoin.

Ces trois techniques sont complexes et ne permettent pas de partager facilement du code dans les différents composants d'une application.

L'approche **Options API** de Vue.js 2 est toujours utilisable dans Vue.js 3. Mais Vue.js 3 offre la possibilité d'organiser le code autour des fonctionnalités logiques de l'application plutôt de ses composants, ce qui facilite la maintenance du code :

Options API



Composition API



Je vais vous montrer plusieurs façons d'utiliser l'API de composition à l'aide d'une application très simple qui contiendra une zone de texte `<input type="text">` et une balise ``. Le texte entré dans la zone de texte sera affiché en majuscules dans la balise ``. Je suis sûr

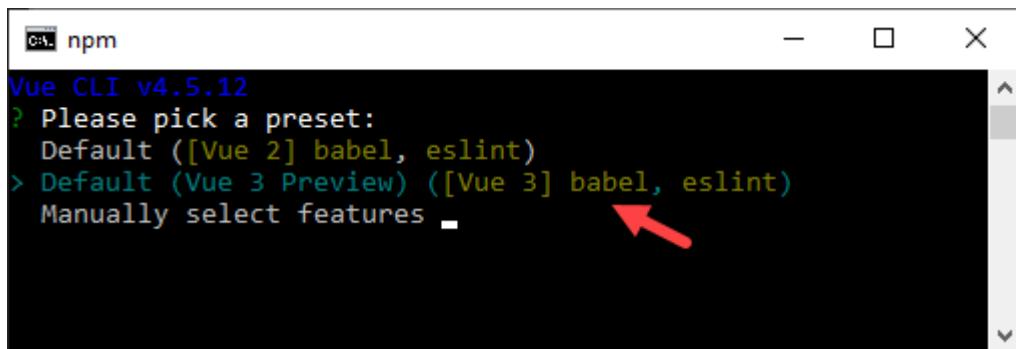
que vous savez comment faire en utilisant l'Options API. Mais ici, je vais vous montrer comment utiliser la composition API.

Nous allons partir d'une l'application standard créée par Vue-Cli.

Dans un premier temps, définissons le projet **composition1** dans le dossier **c:\vue3\cli**. Ouvrez une invite de commande. Allez dans le dossier **c:\vue3\cli** et lancez cette commande :

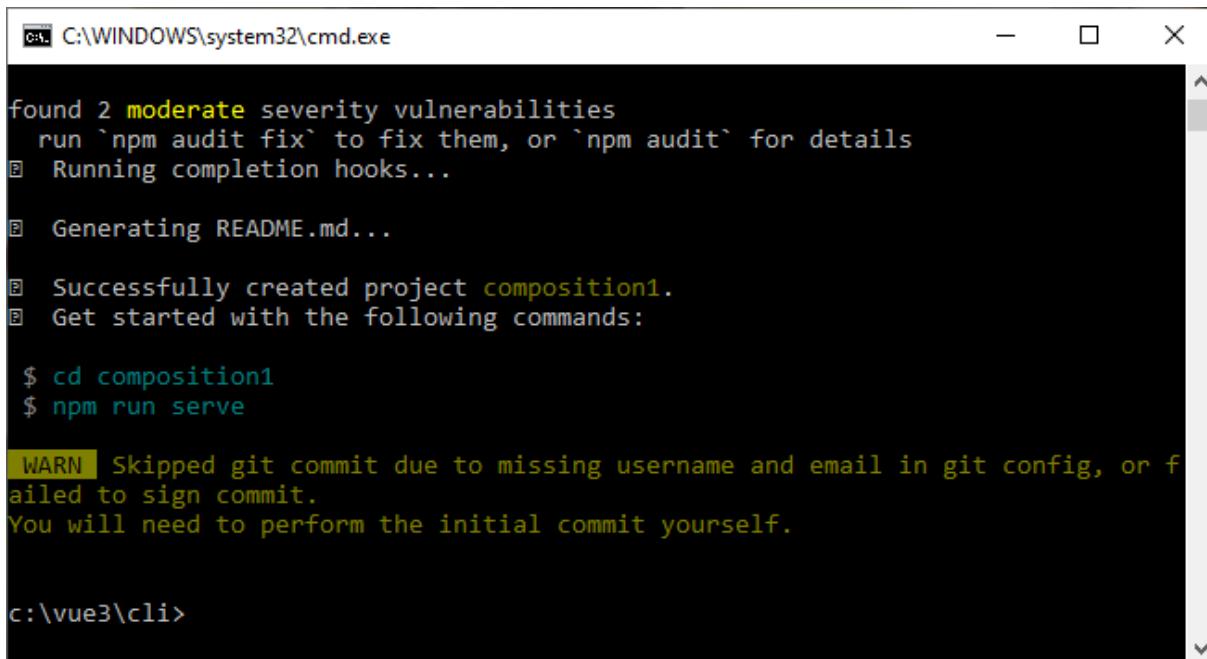
```
vue create composition1
```

Quelques instants plus tard, on vous demande si vous voulez créer un projet Vue 2 ou Vue 3. Sélectionnez **Vue 3** en appuyant sur la touche *flèche vers le bas*, puis validez en appuyant sur la touche *Entrée* :



```
Vue CLI v4.5.12
? Please pick a preset:
  Default ([Vue 2] babel, eslint)
> Default (Vue 3 Preview) ([Vue 3] babel, eslint)
  Manually select features -
```

Patinez jusqu'à ce que l'application soit créée. Vous obtiendrez quelque chose comme ceci :



```
C:\WINDOWS\system32\cmd.exe

Found 2 moderate severity vulnerabilities
  run `npm audit fix` to fix them, or `npm audit` for details
  Running completion hooks...

  Generating README.md...
  Successfully created project composition1.
  Get started with the following commands:

  $ cd composition1
  $ npm run serve

  WARN Skipped git commit due to missing username and email in git config, or failed to sign commit.
  You will need to perform the initial commit yourself.

c:\vue3\cli>
```

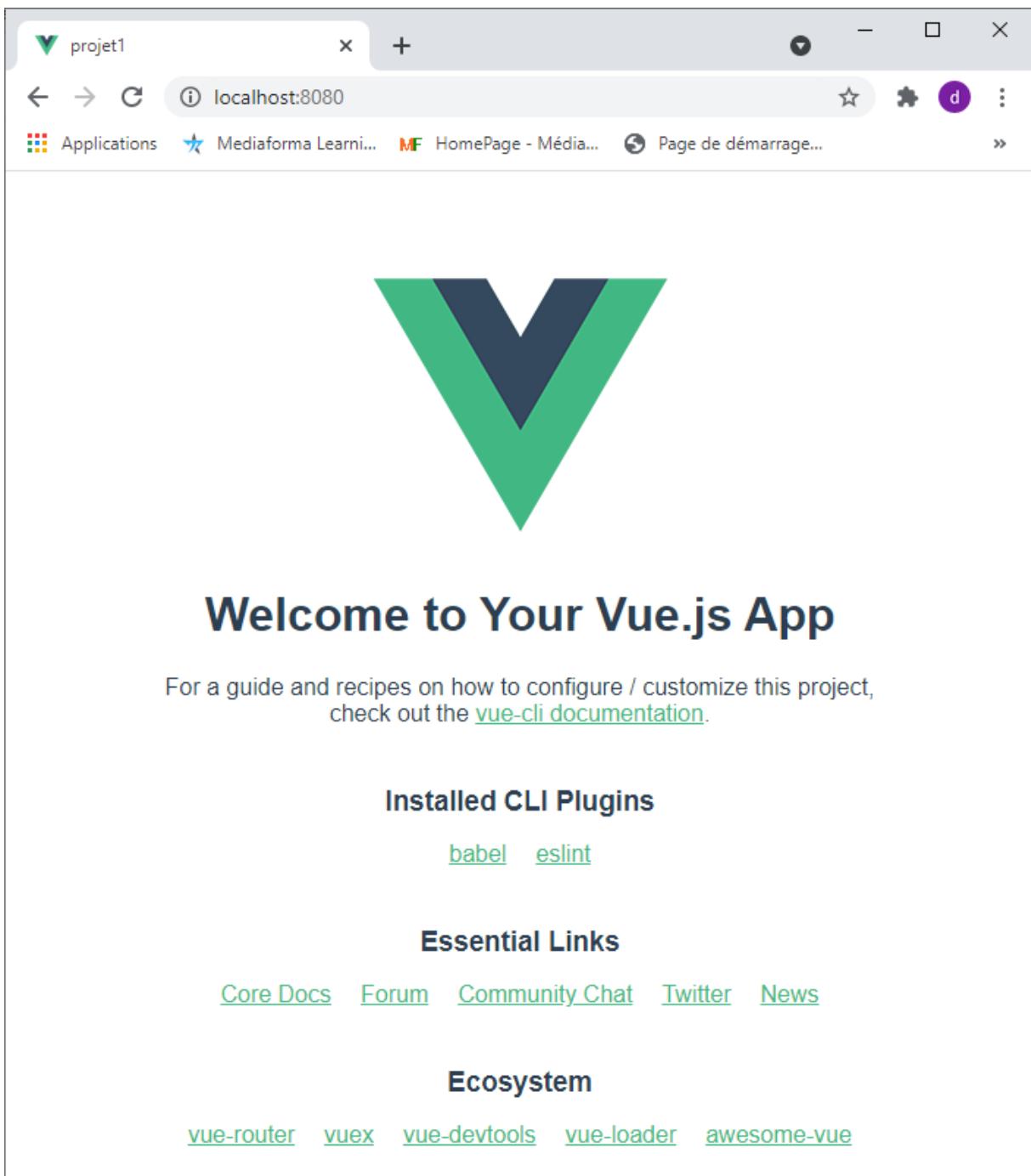
Une fois la commande **vue create** exécutée, lancez les commandes suivantes :

```
cd composition1  
npm run serve
```

L'application est accessible à cette adresse :

[http://localhost :8080](http://localhost:8080)

Voici ce que vous devriez obtenir :



Vous allez maintenant modifier le code de ce projet dans Visual Studio Code.

Lancez la commande **Open Folder** dans le menu **File** de Visual Studio Code et ouvrez le dossier **Composition1**.

Editez le fichier **App.vue**.

Ici, nous n'allons pas utiliser le composant **HelloWorld**, mais plutôt le composant **Majuscules** (fichier App.vue) :

```
<template>
  
  <Majuscules></Majuscules>
</template>
<script>
  import Majuscules from './components/Majuscules.vue'
  export default {
    name : 'App',
    components : {
      Majuscules
    }
  }
</script>
<style>
#app {
  font-family : Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing : antialiased;
  -moz-osx-font-smoothing : grayscale;
  text-align : center;
  color : #2c3e50;
  margin-top : 60px;
}
</style>
```

Dans le dossier **components**, supprimez le fichier **HelloWorld.vue** et remplacez-le par le fichier **Majuscules.vue**. Je vais supposer que le plugin **Vetur** est installé. Tapez `<vue` et appuyez sur la touche *Entrée* pour insérer un squelette standard de composant Vue dans le code. Complétez ce code comme ceci :

```
<template>
```

```

<div class="maj">
  <h2>Composition API</h2>
  <input type="text">
  <span></span>
</div>
</template>
<script>
export default {
  name : 'Majuscules'
}
</script>
<style>
</style>

```

Je vous rappelle que le but de cette application est d'afficher la version majuscule de la zone de texte dans le span.

Première approche – Utilisation de la fonction ref()

Dans cette première approche, nous allons utiliser la fonction **ref()** pour créer des variables réactives. Ceci est une nouveauté de l'API de composition.

Ajoutez cette ligne au-dessus de l'instruction export :

```
import {ref} from 'vue'
```

Vous allez maintenant ajouter la méthode **setup()** dans la section **export**. C'est dans cette méthode que vous utiliserez la fonction **ref()**. La méthode **setup()** sera appelée après le hook **beforeCreate** et avant le hook **created**. Vous ne pourrez donc pas utiliser **this** dans cette fonction car les données ne seront pas encore disponibles.

Nous allons associer la variable entrée à la zone de texte et la variable sortie au span. Ces deux variables seront réactives grâce à la fonction **ref()**

et elles seront initialisées avec une chaîne vide :

```
import {ref} from 'vue';

export default {
    name : 'Majuscules',
    setup() {
        let entree = ref('');
        let sortie = ref('');
    }
}
```

La fonction **setup()** doit retourner un objet JSON qui contient les données réactives :

```
setup() {
    let entree = ref('');
    let sortie = ref('');
    return {
        entree,
        sortie
    }
}
```

Vous allez maintenant relier les variables **entree** et **sortie** avec la vue.

```
<div class="maj">
    <h2>Composition API</h2>
    <input type="text" v-model="entree" />
    <span>{{sortie}}</span>
</div>
```

Pour mettre à jour le span à chaque fois que la zone de texte est modifiée, ajoutez un gestionnaire évènementiel **keyup** dans la zone de texte :

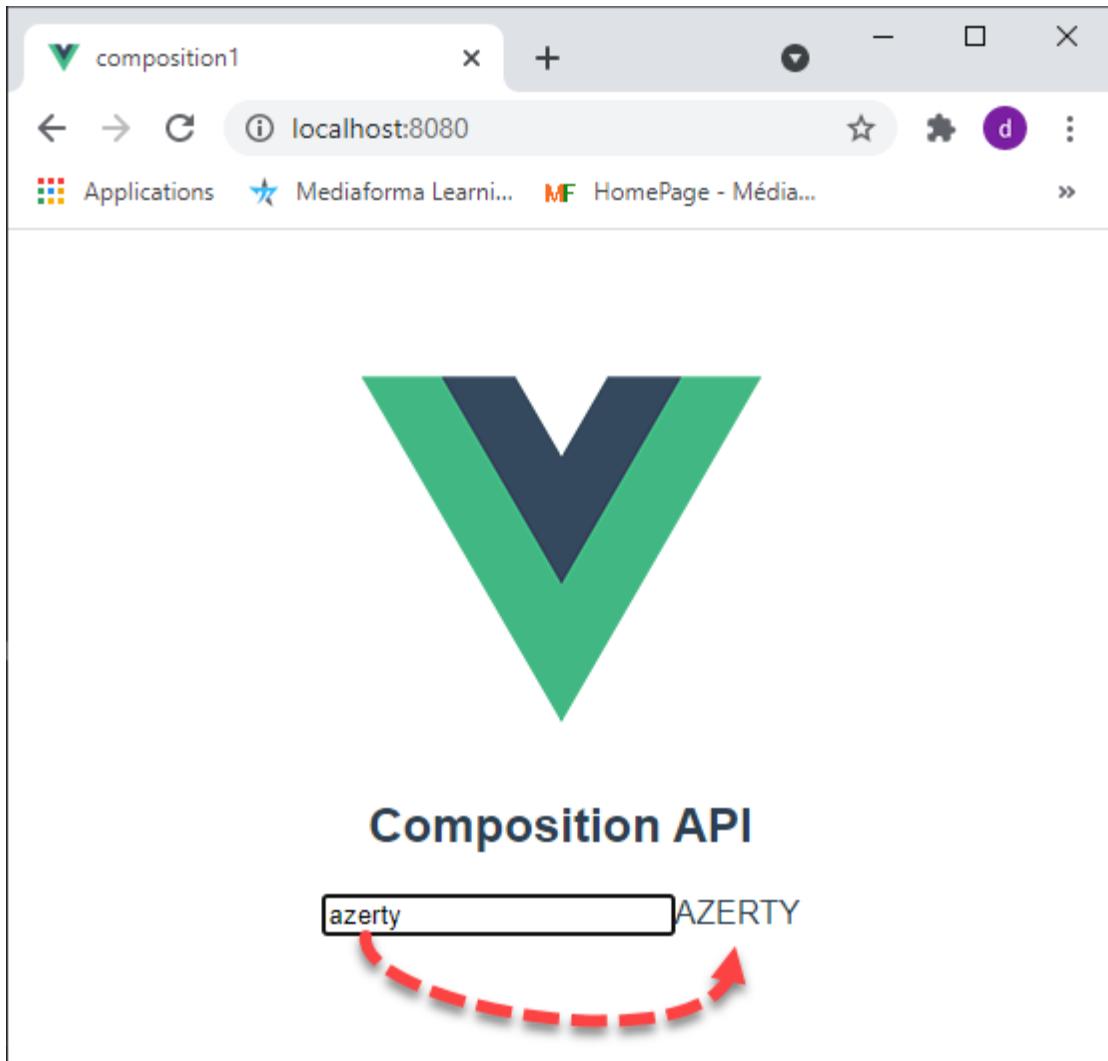
```
<input type="text" v-model="entree" @keyup="enMaj" />
```

Ajoutez la méthode **enMaj** dans le setup. Cette méthode modifie la variable **sortie** en fonction de la variable **entree**. Pensez à ajouter la

fonction dans le return de la fonction **setup()**. Voici le code complet de **Majuscules.vue** (fichier **Majuscules-ref.vue**) :

```
<template>
  <div class="maj">
    <h2>Composition API</h2>
    <input type="text" v-model="entree" @keyup="enMaj" />
    <span>{{sortie}}</span>
  </div>
</template>
<script>
  import { ref } from "vue";
  export default {
    name : "Majuscules",
    setup() {
      let entree = ref("");
      let sortie = ref("");
      function enMaj() {
        sortie.value = entree.value.toUpperCase();
      }
      return {
        entree,
        sortie,
        enMaj
      };
    },
  };
</script>
<style>
</style>
```

Désormais, lorsque vous entrez du texte dans la zone de texte, il est recopié en majuscules dans le span :



Deuxième approche – Utilisation de la fonction reactive()

Dans cette deuxième approche, nous allons utiliser la fonction **reactive()** pour créer des variables réactives. Ceci est une nouveauté de l'API de composition. Avec l'approche **ref()**, vous avez dû appeler autant de fois **ref()** que vous vouliez créer de variables réactives. Avec l'approche **reactive()**, vous allez placer toutes les propriétés à rendre réactives dans un seul objet JSON.

Dupliquez le fichier **Majuscules.vue** et renommez la copie **Majuscules-ref.vue**. Ainsi, vous conserverez la précédente approche de l'API de composition dans le fichier **Majuscules-ref.vue**.

Vous allez maintenant modifier le fichier **Majuscules.vue**.

Remplacez cette ligne :

```
import {ref} from 'vue'
```

Par celle-là :

```
import {reactive} from 'vue'
```

Supprimez les variables **entree** et **sortie**. Remplacez-les par la variable **state** et affectez-lui la fonction **reactive()** à laquelle vous passerez un objet JSON qui définit les variables **entree** et **sortie**. Retournez la variable **state** et remplacez les références à **entree** et **sortie** par **state.entree** et **state.sortie**. Voici le code complet de **Majuscules.vue** (fichier **Majuscules-reactive.vue**) :

```
<template>
  <div class="maj">
    <h2>Composition API</h2>
    <input type="text" v-model="state.entree" @keyup="enMaj" />
    <span>{{ state.sortie }}</span>
  </div>
</template>
<script>
  import { reactive } from "vue";
  export default {
    name : "Majuscules",
    setup() {
      let state = reactive({
        entree : '',
        sortie : ''
      });
      function enMaj() {
        state.sortie = state.entree.toUpperCase();
      }
    }
  }
</script>
```

```

        state.sortie = state.entree.toUpperCase();
    }

    return {
        state,
        enMaj,
    };
},
};

</script>
<style>
</style>

```

Troisième approche – Utilisation de la fonction computed()

Dans cette troisième approche, nous allons utiliser la fonction `computed()` pour mettre à jour la propriété `sortie` chaque fois que la propriété `entree` change. La fonction évènementielle ne sera donc plus nécessaire.

Dupliquez le fichier **Majuscules.vue** et renommez la copie **Majuscules-reactive.vue**. Ainsi, vous conserverez la précédente approche de l'API de composition dans le fichier **Majuscules-reactive.vue**.

Vous allez maintenant modifier le fichier **Majuscules.vue**.

Supprimez la fonction évènementielle `enMaj()`, sa référence dans le `return` et sa référence `@keyup` dans la balise `input`. Ajoutez une référence à `computed` dans l'instruction `import`. Modifiez la valeur affectée à la propriété `sortie` dans la variable `state`. Affectez-lui une fonction anonyme qui retourne la mise en majuscules de la zone de texte.

Voici le code complet (fichier **Majuscules-computed.vue**) :

```

<template>
<div class="maj">
    <h2>Composition API</h2>

```

```

<input type="text" v-model="state.entre" />
<span>{{ state.sortie }}</span>
</div>
</template>
<script>
import { reactive, computed } from "vue";
export default {
  name : "Majuscules",
  setup() {
    let state = reactive({
      entre : '',
      sortie : computed(()=>{
        return state.entre.toUpperCase();
      })
    });
    return {
      state,
    };
  },
};
</script>
<style>
</style>

```

Quatrième approche – Externalisation du calcul réactif

Dans cette quatrième approche, nous allons placer la logique qui calcule sortie à partir de entrée dans une fonction à l'extérieur de **setup()**. Cela facilitera la réutilisabilité du code.

Dupliquez le fichier **Majuscules.vue** et renommez la copie **Majuscules-computed.vue**. Ainsi, vous conserverez la précédente approche de l'API de composition dans le fichier **Majuscules-computed.vue**.

Vous allez maintenant modifier le fichier **Majuscules.vue**.

Définissez la fonction **enMaj()** et déplacez la logique de calcul de sortie dans cette fonction. Pour que la variable **state** soit réactive, ajoutez une référence à **toRefs** dans l'instruction **import** et appliquez la fonction **toRefs** à la variable **state** dans le **return**.

Dans le **setup**, affectez par déstructuration ce qui est retourné par la fonction **enMaj()** aux variables **entree** et **sortie** et retournez ces deux variables.

Voici le code complet (fichier **Majuscules.vue**) :

```
<template>
  <div class="maj">
    <h2>Composition API</h2>
    <input type="text" v-model="entree" />
    <span>{{ sortie }}</span>
  </div>
</template>
<script>
  import { reactive, computed, toRefs } from "vue";
  function enMaj() {
    let state = reactive({
      entree : "",
      sortie : computed(() => {
        return state.entree.toUpperCase();
      }),
    });
    return toRefs(state);
  }
  export default {
```

```
name : "Majuscules",
setup() {
    let { entree, sortie } = enMaj();
    return {
        entree,
        sortie,
    };
},
</script>
<style>
</style>
```

Vue 3 et TypeScript

Vue 3 a été écrit en TypeScript. Il semble donc logique d'utiliser TypeScript pour créer vos applications et interfaces. D'autant plus que cela ne change pas grand-chose par rapport à ce que vous avez appris jusqu'ici.

Dans cette section, je vais vous montrer comment créer une application Vue 3 très simple en TypeScript avec Vue-Cli. Les fichiers se trouvent dans le dossier **cli\v3ts**.

Vous allez créer un nouveau projet. Rendez-vous dans le dossier où le projet doit être créé et tapez cette commande (ici, le projet a pour nom **v3ts**) :

```
vue create v3ts
```

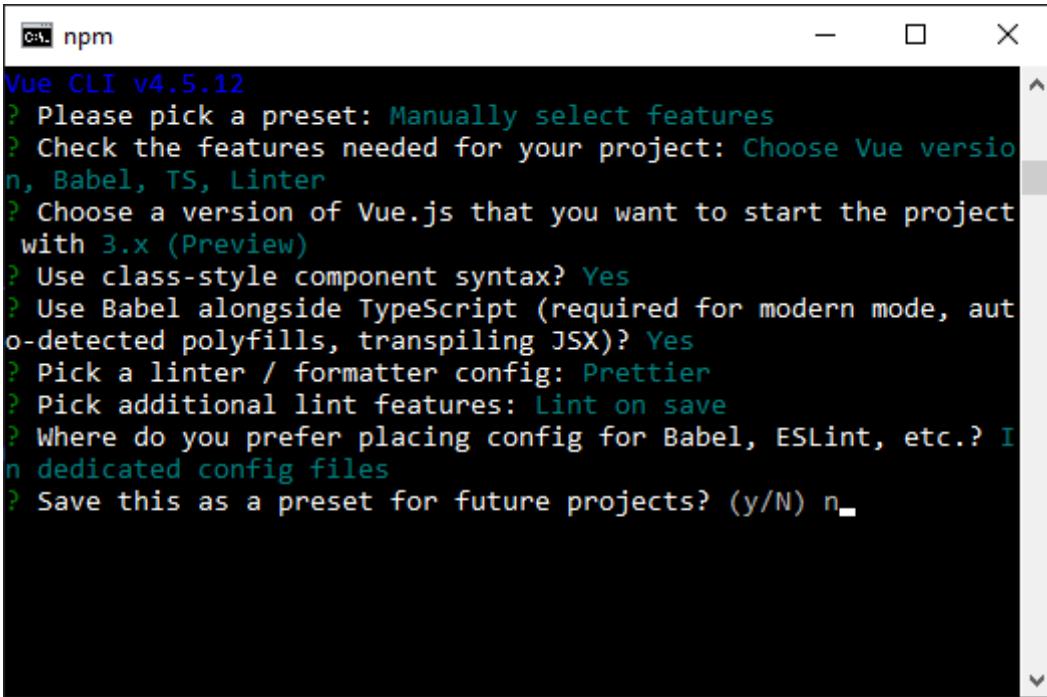
Quelques instants plus tard, on vous demande de choisir le type du projet à créer. Sélectionnez la dernière option et appuyez sur la touche *Entrée* :

```
Vue CLI v4.5.12
? Please pick a preset:
  Default ([Vue 2] babel, eslint)
  Default (Vue 3 Preview) ([Vue 3] babel, eslint)
> Manually select features
```

Sélectionnez les options **Choose Vue version**, **Babel**, **TypeScript** et **Linter/Formatter**, puis appuyez sur la touche *Entrée* :

```
Vue CLI v4.5.12
? Please pick a preset: Manually select features
? Check the features needed for your project:
  (*) Choose Vue version
  (*) Babel
>(*) TypeScript
  ( ) Progressive Web App (PWA) Support
  ( ) Router
  ( ) Vuex
  ( ) CSS Pre-processors
  (*) Linter / Formatter
  ( ) Unit Testing
  ( ) E2E Testing
```

Plusieurs questions vont ensuite vous être posées. Voici la configuration à utiliser :



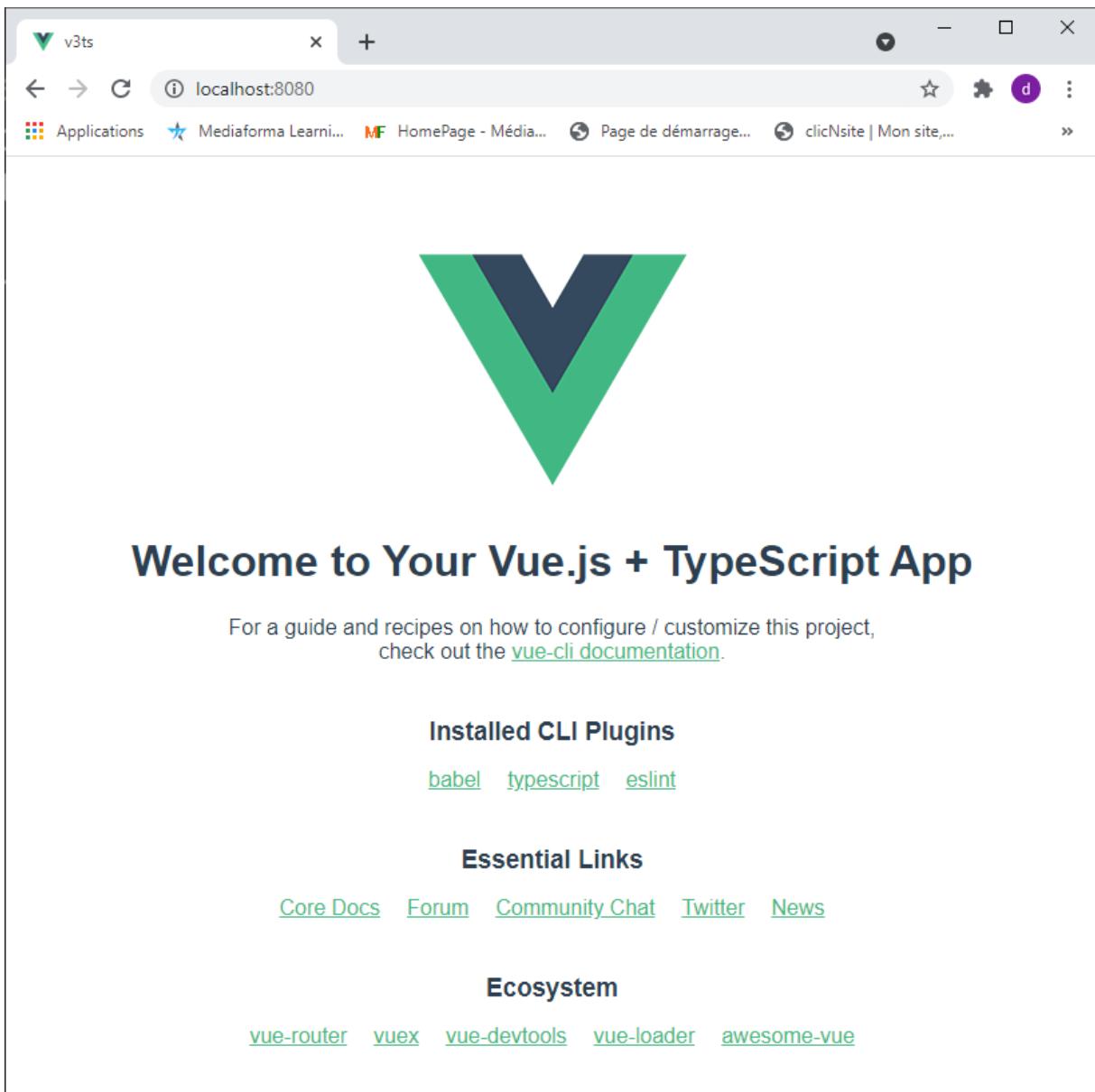
The screenshot shows a terminal window titled "npm" with the command "Vue CLI v4.5.12". The window displays a series of configuration prompts from the Vue CLI:

- ? Please pick a preset: Manually select features
- ? Check the features needed for your project: Choose Vue version, Babel, TS, Linter
- ? Choose a version of Vue.js that you want to start the project with 3.x (Preview)
- ? Use class-style component syntax? Yes
- ? Use Babel alongside TypeScript (required for modern mode, auto-detected polyfills, transpiling JSX)? Yes
- ? Pick a linter / formatter config: Prettier
- ? Pick additional lint features: Lint on save
- ? Where do you prefer placing config for Babel, ESLint, etc.? In dedicated config files
- ? Save this as a preset for future projects? (y/N) n

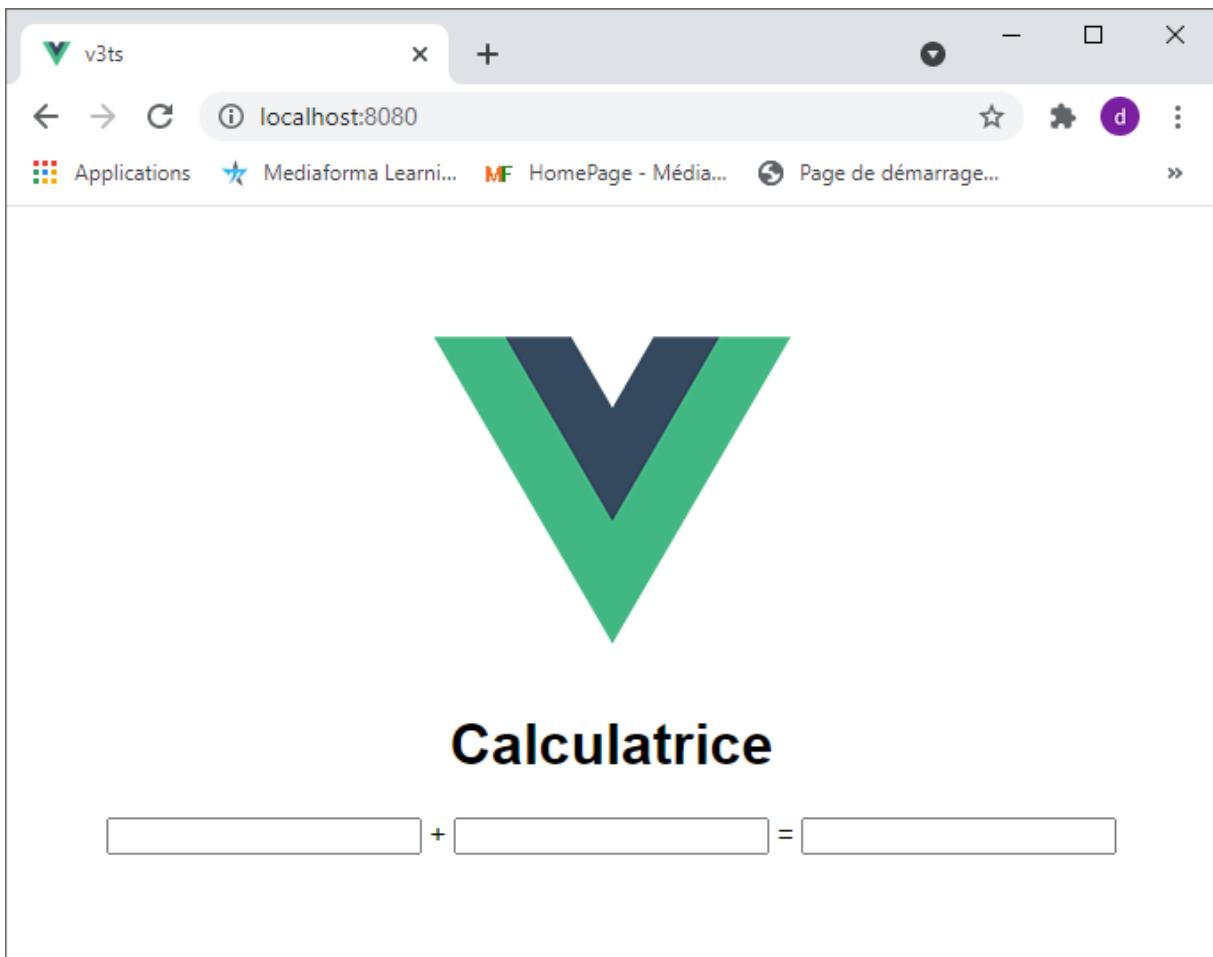
Déplacez-vous dans le dossier **v3ts** et exécutez l'application avec ces commandes :

```
cd v3ts  
npm run serve
```

Ouvrez votre navigateur et allez sur la page <http://localhost:8080/>. Voici ce que vous devez obtenir :



Vous allez maintenant modifier le code généré par Vue-Cli pour créer une mini calculatrice réactive :



Modifiez le fichier **App.vue** comme ceci :

```
<template>
  
  <h1>Calculatrice</h1>
  <Calculator></Calculator>
</template>
<script lang="ts">
  import { Options, Vue } from "vue-class-component";
  import Calculator from "./components/Calculator.vue";
  @Options({
    components : {
      Calculator,
    },
  },
```

```

        }

export default class App extends Vue { }

</script>

<style>

#app {
    font-family : Avenir, Helvetica, Arial, sans-serif;
    text-align : center;
    margin-top : 60px;
}

</style>

```

Rien de bien compliqué : l'application utilise le composant **Calculator** et non le composant **HelloWorld**.

Renomme le fichier **HelloWorld.vue** en Fichier **Calculator.vue**.

Le template contient trois balises **<input>** de type **text** : deux pour saisir les nombres à additionner et une pour afficher le résultat. Ces input sont liés aux propriétés **n1**, **n2** et **result** du ViewModel avec des directives **v-model**. À chaque frappe d'une touche du clavier, la fonction **calc()** est exécutée :

```

<template>

    <input type="text" v-model="n1" @keyup="calc">
    <span> + </span>
    <input type="text" v-model="n2" @keyup="calc">
    <span> = </span>
    <input type="text" v-model="result">
</template>

```

Nous allons utiliser l'API de composition et la fonction **ref()** pour créer des variables réactives. Ajoutez cette ligne au-dessus de l'instruction **export** :

```
import {ref} from 'vue';
```

Vous allez maintenant ajouter la méthode **setup()** dans la section **export**. C'est dans cette méthode que vous utiliserez la fonction **ref()**.

Les propriétés associées aux trois balises `<input>` sont réactives grâce à la fonction `ref()` et elles seront initialisées avec une chaîne vide :

```
export default {

  setup() {
    let n1 = ref('');
    let n2 = ref('');
    let result = ref('');

  }
}
```

Ajoutez la fonction `calc()` au `setup`. Cette fonction affecte la somme des deux premiers `<input>` au troisième. Ici, on utilise des instructions ternaires et la fonction `isNaN()` pour éviter d'afficher **Nan** dans le résultat :

```
function calc() : void {
  result.value = ((isNaN(parseInt(n1.value)) ? 0 : parseInt(n1.value))
    + (isNaN(parseInt(n2.value)) ? 0 : parseInt(n2.value))).toString();
}

  }
```

Retournez les propriétés et la fonction définie dans `setup()` :

```
return {
  n1,
  n2,
  result,
  calc
};
```

Voici le code complet du fichier **Calculator.vue** :

```
<template>

<input type="text" v-model="n1" @keyup="calc">
<span> + </span>
<input type="text" v-model="n2" @keyup="calc">
<span> = </span>
<input type="text" v-model="result">

</template>
```

```
<script lang="ts">

import { Options, Vue } from "vue-class-component";
import { ref } from 'vue';
export default {

  setup() {
    let n1 = ref('');
    let n2 = ref('');
    let result = ref('');

    function calc(): void {
      result.value = ((isNaN(parseInt(n1.value)) ? 0 : parseInt(n1.value))
        + (isNaN(parseInt(n2.value)) ? 0 : parseInt(n2.value))).toString();
    }

    return {
      n1,
      n2,
      result,
      calc
    };
  },
};

</script>
```

Le mot de la fin

Avant de vous quitter, je tiens à vous remercier pour l'achat de ce manuel. J'espère qu'il a entièrement répondu aux questions que vous vous posiez sur Vue.js 3.

J'aimerais vous demander une "petite faveur". Pourriez-vous prendre une ou deux minutes pour me dire ce que vous avez pensé de ce manuel ? Pour

cela, [il vous suffit de cliquer sur ce lien](#).

Je vous rappelle que vous pouvez compléter ce manuel par une formation vidéo de plus de 10 heures sur Vue.js 3. Contactez-moi [en cliquant sur ce lien](#). Je vous enverrai un bon de réduction.

La formation sera accessible à 9,99 € ~~au lieu de 109,99 €~~ sur la plateforme Udemy.

Je vous souhaite de beaux codes Vue.js 3 et beaucoup de plaisir à les développer.

Michel Martin