FORMATION-VUE.NETLIFY.APP

Construire une application reactive avec Vue

DE LA THÉORIE À L'APPLICATION BIEN FINIE

Michel EDIGHOFFER, 30-31 janvier 2023
Capgemini Engineering

Programme de la formation

- Développer avec Vue
  - Son histoire
  - Ses forces vives
  - Options API ou Composition API ?
- Guider ses développements et les mettre sous contrôle

Développer avec Vue

PARTIE 1

Son histoire

Créé en **2014 par Evan You** pour construire rapidement une application web sans prise de tête.

Devenu viral, il a grandi en gardant sa simplicité mais en permettant de monter en complexité progressivement.

> Il se considère **évolutif** pour sa capacité à s'adapter aux besoins techniques

Aujourd'hui, il culmine parmi les plus grands framework JS utilisés (avec React, Angular, Svelte et SolidJS).

Framework
ou
Librairie ?

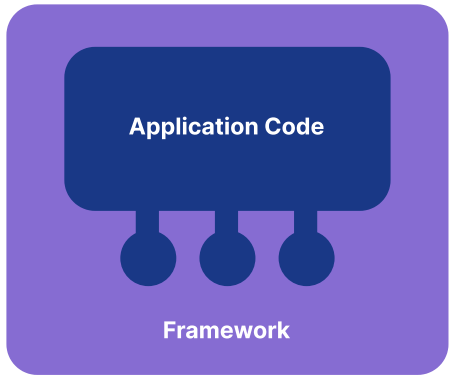
Framework
ou
Librairie ?

UNE PETITE EXPLICATION S'IMPOSE

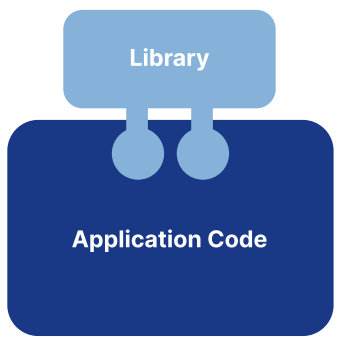
Framework / Librairie

`Framework` : Ensemble de règles imposées techniquement pour guider un travail à effectuer.

`Librairie` : Ensemble de solutions techniques prêtes à l'emploi



Malgré son apparence de librairie,
`Vue` **se positionne comme un framework grâce à son écosystème**.

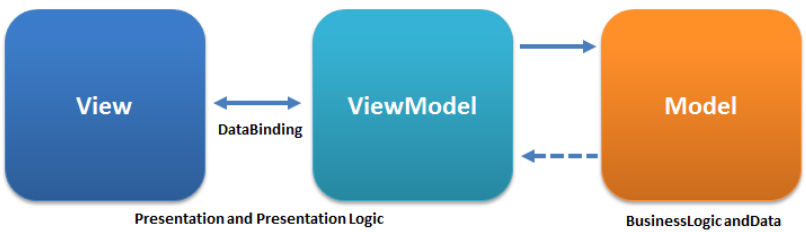


SES FORCES VIVES

POUR TE RENDRE PRODUCTIF ET EFFICACE

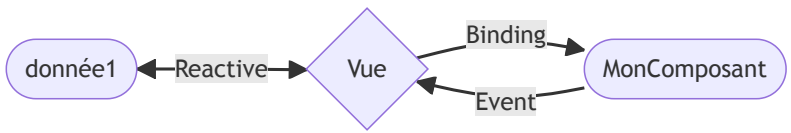
COMPOSANTS GUIDÉS PAR LES DONNÉES

L'un des principes fondamentaux de Vue certifie que les données soient toujours bien représentées dans la vue. Se reposant sur le principe `MVVC` (Model-View-ViewModel), Vue propose de simplifier cela via un système de réactivité.



COMPOSANTS GUIDÉS PAR LES DONNÉES

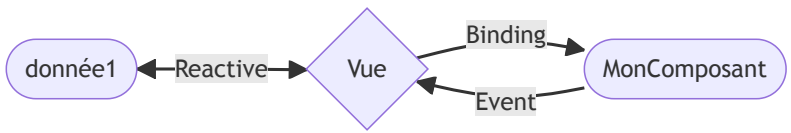
L'un des principes fondamentaux de Vue certifie que les données soient toujours bien représentées dans la vue. Se reposant sur le principe `MVVC` (Model-View-ViewModel), Vue propose de simplifier cela via un système de réactivité.



`Reactive` : Système de réactivité
`Binding` : Liaison utilisant le *Virtual DOM*
`Event` : Remontée d'événements

COMPOSANTS GUIDÉS PAR LES DONNÉES

L'un des principes fondamentaux de Vue certifie que les données soient toujours bien représentées dans la vue. Se reposant sur le principe `MVVC` (Model-View-ViewModel), Vue propose de simplifier cela via un système de réactivité.



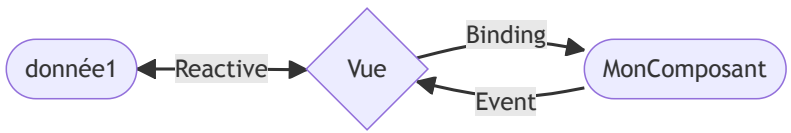
`Reactive` : Système de réactivité
`Binding` : Liaison utilisant le *Virtual DOM*
`Event` : Remontée d'événements

Ainsi, il est interdit de :

- Modifier directement le DOM
- Respecter la responsabilité des composants

COMPOSANTS GUIDÉS PAR LES DONNÉES

L'un des principes fondamentaux de Vue certifie que les données soient toujours bien représentées dans la vue. Se reposant sur le principe `MVVC` (Model-View-ViewModel), Vue propose de simplifier cela via un système de réactivité.

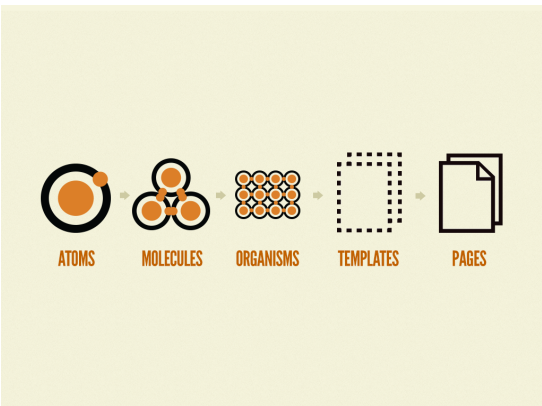


`Reactive` : Système de réactivité
`Binding` : Liaison utilisant le *Virtual DOM*
`Event` : Remontée d'événements

Ainsi, il est interdit de :

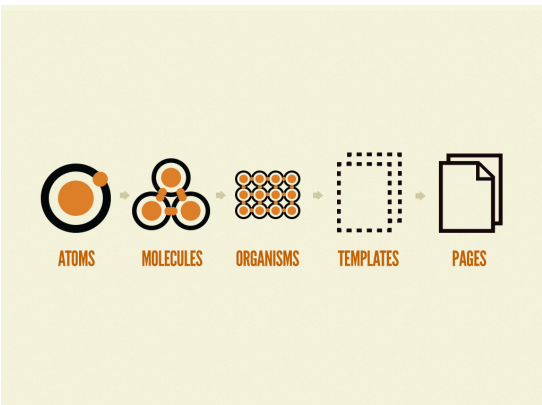
- Modifier directement le DOM
- Respecter la responsabilité des composants

> Ne pas modifier directement les props
> Ne pas modifier les props du parent



La composition dans une application consiste en l'utilisation de plusieurs composantes pour former un organisme muni de corps inter-dépendants.

L'Atomic Design suggère un découpage à l'image de la composition chimique en assurant la responsabilité de chacun.

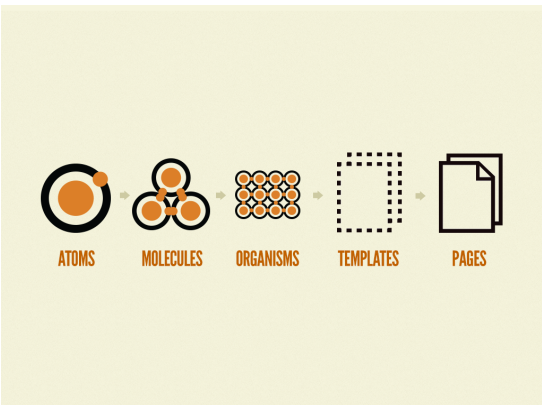


La composition dans une application consiste en l'utilisation de plusieurs composantes pour former un organisme muni de corps inter-dépendants.

L'Atomic Design suggère un découpage à l'image de la composition chimique en assurant la responsabilité de chacun.

- **Atome** : une entité simple ayant une sémantique unique et réutilisable

  bouton, champ texte, icône



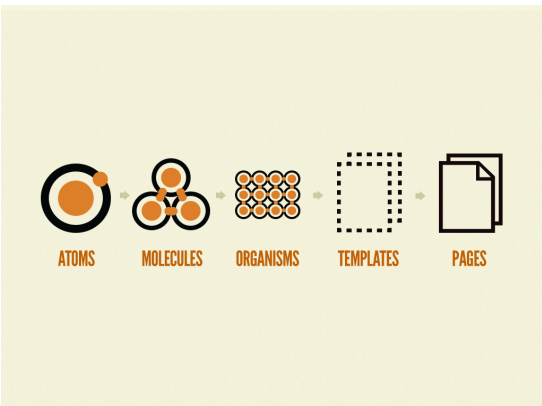
La composition dans une application consiste en l'utilisation de plusieurs composantes pour former un organisme muni de corps inter-dépendants.

L'Atomic Design suggère un découpage à l'image de la composition chimique en assurant la responsabilité de chacun.

- **Atome** : une entité simple ayant une sémantique unique et réutilisable

  bouton, champ texte, icône
- **Molécule** : atomes assemblés pour une sémantique flexible et réutilisable

  champ avec label, table avec champ de recherche



La composition dans une application consiste en l'utilisation de plusieurs composantes pour former un organisme muni de corps inter-dépendants.

L'Atomic Design suggère un découpage à l'image de la composition chimique en assurant la responsabilité de chacun.

- **Atome** : une entité simple ayant une sémantique unique et réutilisable

  bouton, champ texte, icône
- **Molécule** : atomes assemblés pour une sémantique flexible et réutilisable

  champ avec label, table avec champ de recherche
- **Organisme** : ensemble complexe d'atomes ou/et de molécules (voire d'autres organismes) de sémantiques différentes

  en-tête d'application, section d'application

TEMPLATING ET SES ASTUCES

Vue adopte un système de templating pour 4 grandes raisons :

pouvoir utiliser le DOM comme template
garder une séparation nette entre la vue et les données
proposer un sucre syntaxique facile à retenir
optimiser la manipulation du DOM

POUVOIR UTILISER LE DOM COMME TEMPLATE

Vue en mode **html**

```
<div id="root">
  <section>
    <h1>{{name}}</h1>
    <nav :class="{ hidden: list.length === 0 }">
      <ul>
        <li v-for="item in list">
          {{item}}
        </li>
      </ul>
    </nav>
  </section>
</div>
```

Ce template peut tout à fait être intégré tel quel dans du HTML, le navigateur peut l'interpréter sans erreur.

POUVOIR UTILISER LE DOM COMME TEMPLATE

Vue en mode **html**

```
<div id="root">
  <section>
    <h1>{{name}}</h1>
    <nav :class="{ hidden: list.length === 0 }">
      <ul>
        <li v-for="item in list">
          {{item}}
        </li>
      </ul>
    </nav>
  </section>
</div>
```

Ce template peut tout à fait être intégré tel quel dans du HTML, le navigateur peut l'interpréter sans erreur.

POUVOIR UTILISER LE DOM COMME TEMPLATE

Vue en mode **html**

```
<div id="root">
  <section>
    <h1>{{name}}</h1>
    <nav :class="{ hidden: list.length === 0 }">
      <ul>
        <li v-for="item in list">
          {{item}}
        </li>
      </ul>
    </nav>
  </section>
</div>
```

Ce template peut tout à fait être intégré tel quel dans du HTML, le navigateur peut l'interpréter sans erreur.

POUVOIR UTILISER LE DOM COMME TEMPLATE

Vue en mode **html**

```
<div id="root">
  <section>
    <h1>{{name}}</h1>
    <nav :class="{ hidden: list.length === 0 }">
      <ul>
        <li v-for="item in list">
          {{item}}
        </li>
      </ul>
    </nav>
  </section>
</div>
```

Ce template peut tout à fait être intégré tel quel dans du HTML, le navigateur peut l'interpréter sans erreur.

POUVOIR UTILISER LE DOM COMME TEMPLATE

Vue en mode **html**

```
<div id="root">
  <section>
    <h1>{{name}}</h1>
    <nav :class="{ hidden: list.length === 0 }">
      <ul>
        <li v-for="item in list">
          {{item}}
        </li>
      </ul>
    </nav>
  </section>
</div>
```

Ce template peut tout à fait être intégré tel quel dans du HTML, le navigateur peut l'interpréter sans erreur.

Vue peut se monter sur un template *inline* et y ajouter la réactivité et les interactions nécessaires.

```
const app = createApp()

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

POUVOIR UTILISER LE DOM COMME TEMPLATE

Vue en mode **application**

```
<!-- L'ancre de montage -->
<div id="root"></div>

<!-- menu.vue -->
<template>
  <section>
    <h1>{{name}}</h1>
    <nav :class="{ hidden: list.length === 0 }">
      <ul>
        <li v-for="item in list">{{item}}</li>
      </ul>
    </nav>
  </section>
</template>
```

POUVOIR UTILISER LE DOM COMME TEMPLATE

Vue en mode **application**

```
<!-- L'ancre de montage -->
<div id="root"></div>

<!-- menu.vue -->
<template>
  <section>
    <h1>{{name}}</h1>
    <nav :class="{ hidden: list.length === 0 }">
      <ul>
        <li v-for="item in list">{{item}}</li>
      </ul>
    </nav>
  </section>
</template>
```

Toute l'application App est montée sur le point de montage.

```
const app = createApp(App)

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

GARDER UNE SÉPARATION NETTE ENTRE LA VUE ET LES DONNÉES

Le web c'est l'HTML, le CSS et le Javascript.

L'HTML peut fonctionner seul, sans Javascript.
Le CSS aussi.

Respecter les responsabilités de chacun assure une bonne organisation, et permet une meilleure pérennité du code.

Les **composants monofichiers**,
ou les Single-File Component (SFC), apportent l'union des trois au sein d'un même fichier.

```
<script>
export default {
  data() {
    return {
      greeting: 'Hello World!'
    }
  }
}
</script>

<template>
  <p class="greeting">{{ greeting }}</p>
</template>

<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

GARDER UNE SÉPARATION NETTE ENTRE LA VUE ET LES DONNÉES

Le web c'est l'HTML, le CSS et le Javascript.

L'HTML peut fonctionner seul, sans Javascript.
Le CSS aussi.

Respecter les responsabilités de chacun assure une bonne organisation, et permet une meilleure pérennité du code.

Les **composants monofichiers**,
ou les Single-File Component (SFC), apportent l'union des trois au sein d'un même fichier.

```
<script>
export default {
  data() {
    return {
      greeting: 'Hello World!'
    }
  }
}
</script>

<template>
  <p class="greeting">{{ greeting }}</p>
</template>

<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

GARDER UNE SÉPARATION NETTE ENTRE LA VUE ET LES DONNÉES

Le web c'est l'HTML, le CSS et le Javascript.

L'HTML peut fonctionner seul, sans Javascript.
Le CSS aussi.

Respecter les responsabilités de chacun assure une bonne organisation, et permet une meilleure pérennité du code.

Les **composants monofichiers**,
ou les Single-File Component (SFC), apportent l'union des trois au sein d'un même fichier.

```
<script>
export default {
  data() {
    return {
      greeting: 'Hello World!'
    }
  }
}
</script>

<template>
  <p class="greeting">{{ greeting }}</p>
</template>

<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

GARDER UNE SÉPARATION NETTE ENTRE LA VUE ET LES DONNÉES

Le web c'est l'HTML, le CSS et le Javascript.

L'HTML peut fonctionner seul, sans Javascript.
Le CSS aussi.

Respecter les responsabilités de chacun assure une bonne organisation, et permet une meilleure pérennité du code.

Les **composants monofichiers**,
ou les Single-File Component (SFC), apportent l'union des trois au sein d'un même fichier.

```
<script>
export default {
  data() {
    return {
      greeting: 'Hello World!'
    }
  }
}
</script>

<template>
  <p class="greeting">{{ greeting }}</p>
</template>

<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

GARDER UNE SÉPARATION NETTE ENTRE LA VUE ET LES DONNÉES

Le web c'est l'HTML, le CSS et le Javascript.

L'HTML peut fonctionner seul, sans Javascript.
Le CSS aussi.

Respecter les responsabilités de chacun assure une bonne organisation, et permet une meilleure pérennité du code.

Les **composants monofichiers**,
ou les Single-File Component (SFC), apportent l'union des trois au sein d'un même fichier.

```
<script>
export default {
  data() {
    return {
      greeting: 'Hello World!'
    }
  }
}
</script>

<template>
  <p class="greeting">{{ greeting }}</p>
</template>

<style>
.greeting {
  color: red;
  font-weight: bold;
}
</style>
```

PROPOSER UN SUCRE SYNTAXIQUE FACILE À RETENIR

À la différence du JavaScript, le templating est extensible
et propose l'abstraction des fonctionnalités du framework tout en se reposant sur l'HTML.

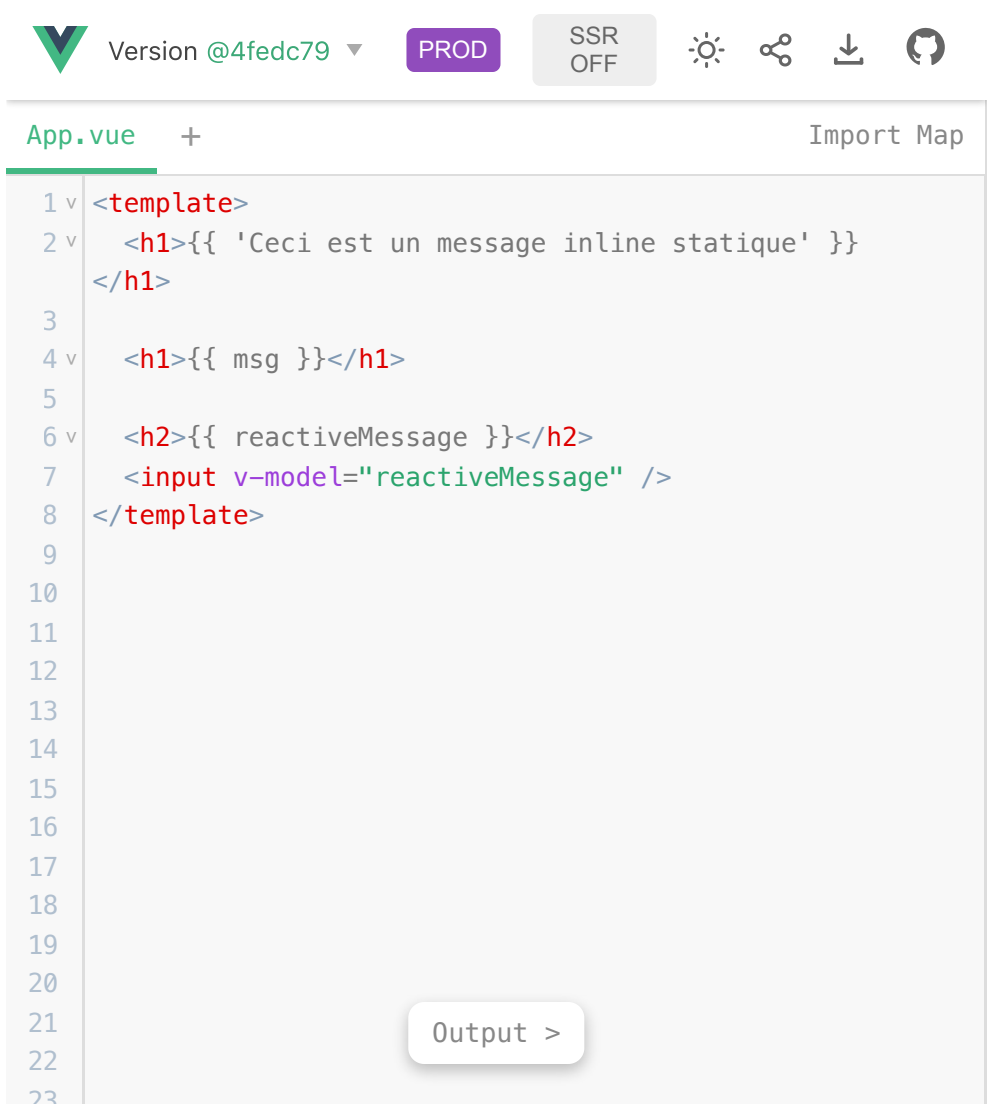
Voici une liste des fonctions les plus pratiques utilisable via le template :

Interpolation de texte
Les directives
Liaison de données réactives
Rendu conditionnel
Rendu de boucles

INTERPOLATION DE TEXTE

Pour rendre une variable de type texte dans le template via les moustaches `{{ ... }}`.

Le contenu doit être une expression javascript.



LES DIRECTIVES

Dans le template, Vue propose de rendre certains attributs prédéfinis ou personnalisés pour manipuler les éléments.

Préfixés par `v-`, la valeur de l'attribut doit être une expression javascript.

> Expression javascript, c'est-à-dire quelque chose que l'on peut mettre après un `return` dans une fonction.

Les directives acceptent des arguments (via `:argument`) et des modificateurs (via `.modifier`).

Exemple d'une directive faite-maison :

```
<div v-detective:class.uppercase="maValeur"></div>
<!--
  Utilisation de la directive `detective` avec
  pour argument `class`,
  pour modificateur `uppercase` et
  comme valeur d'attribut de `maValeur`
-->
```

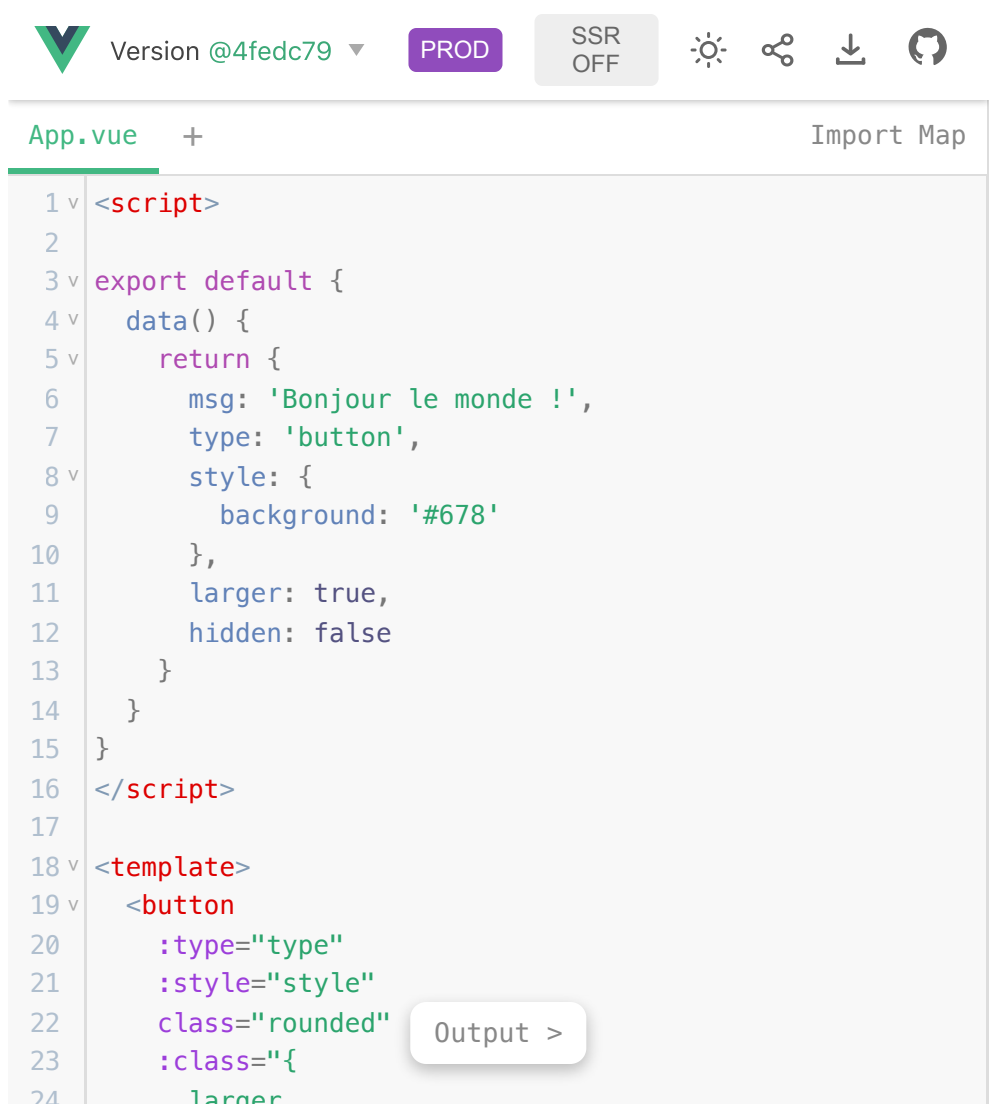
Vue en propose dans son language de template des directives prêtes à l'emploi. →

LIAISON DE DONNÉES RÉACTIVES

Comme déjà mentionné, Vue utilise le binding via un un système de réactivité pour lier une donnée à la vue.

La réactivité se repose sur les Proxys.

Le modifier `v-bind` permet de lier une donnée à un attribut. `:` étant son raccourci



Réactivité
ou
Immutabilité ?

Réactivité ou Immutabilité ?

La réactivité est une manière de gérer les données, et c'est d'ailleurs la plus naturelle de le faire.

Il s'agit de modifier la valeur directement, quelle soit constante ou non.

```
let texte = 'Bonjour'
const article = { nom: 'Sac' }

texte += ' !'
article.prix = 0.85
```

Vue (ainsi que Angular, Svelte et Solid) le pratique pour aligner les changements côté client automatiquement.

Un inconvénient existe :
pour des développeurs non avertis, il peut vite arriver des surprises avec les référence.

Réactivité ou Immutabilité ?

La réactivité est une manière de gérer les données, et c'est d'ailleurs la plus naturelle de le faire.

Il s'agit de modifier la valeur directement, quelle soit constante ou non.

```
let texte = 'Bonjour'
const article = { nom: 'Sac' }

texte += ' !'
article.prix = 0.85
```

Vue (ainsi que Angular, Svelte et Solid) le pratique pour aligner les changements côté client automatiquement.

Un inconvénient existe :
pour des développeurs non avertis, il peut vite arriver des surprises avec les référence.

L'immutabilité est inspirée du monde fonctionnel où pour modifier une donnée, il faut en retourner une nouvelle.

```
const article = { nom: 'Sac' }
function ajouterPrix (prix) {
  return {
    ...article,
    prix
  }
}
const newArticle = ajouterPrix(0.85);
```

Une sécurité dans la manipulation de la donnée au détriment de la simplicité.

Un inconvénient existe :
performance au runtime pouvant être moindre

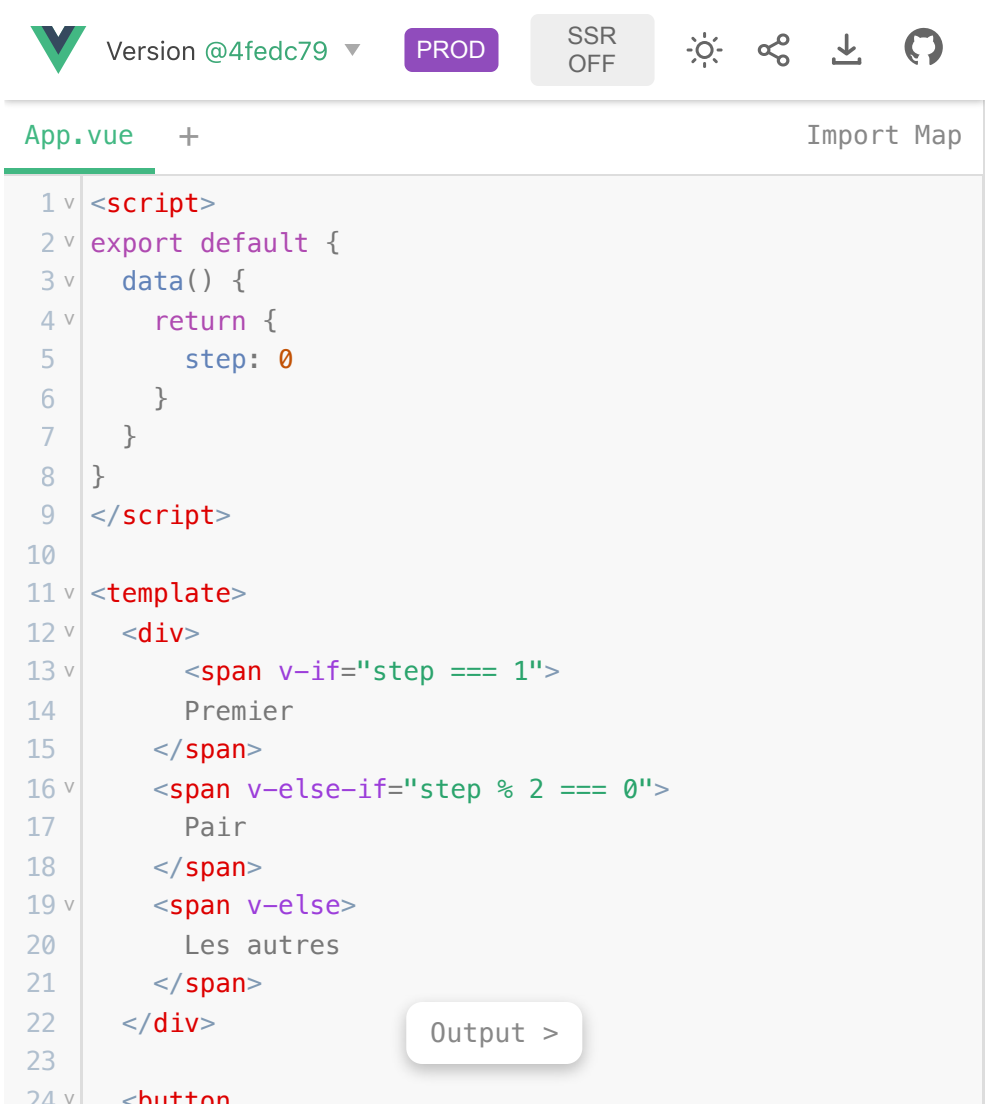
RENDU CONDITIONNEL

Vue propose une directive pour rendre un élément de manière conditionnelle.

`v-if` prend comme valeur une condition boolean,
si vrai, va rendre l'élément,
si faux, ne va pas le rendre (et le retirer).

À la manière des conditions JS, on peut coupler des blocs via `v-else-if` et/ou `v-else`.

L'autre alternative pour cacher sans supprimer : `v-show` qui va simplement appliquer un style pour faire disparaitre de l'écran.

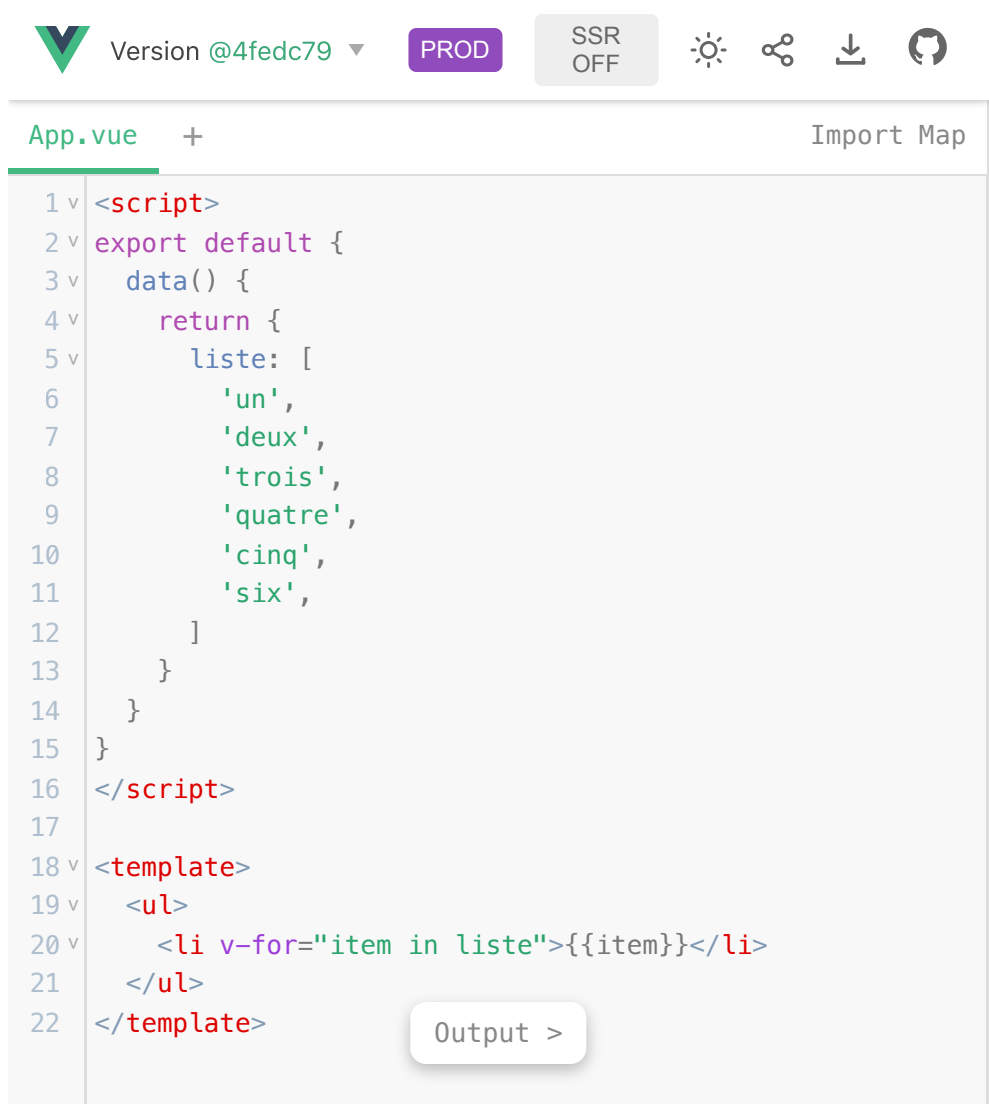


RENDU DE BOUCLES

Pour transformer une liste de données en liste d'éléments, la directive `v-for` est de rigeur.

Il permet de répéter l'élément auquel il est utilisé et donner un contexte pour afficher les données.

```
<section v-for="information in informations">
  <datetime>{{ information.date }}</datetime>
</section>
```



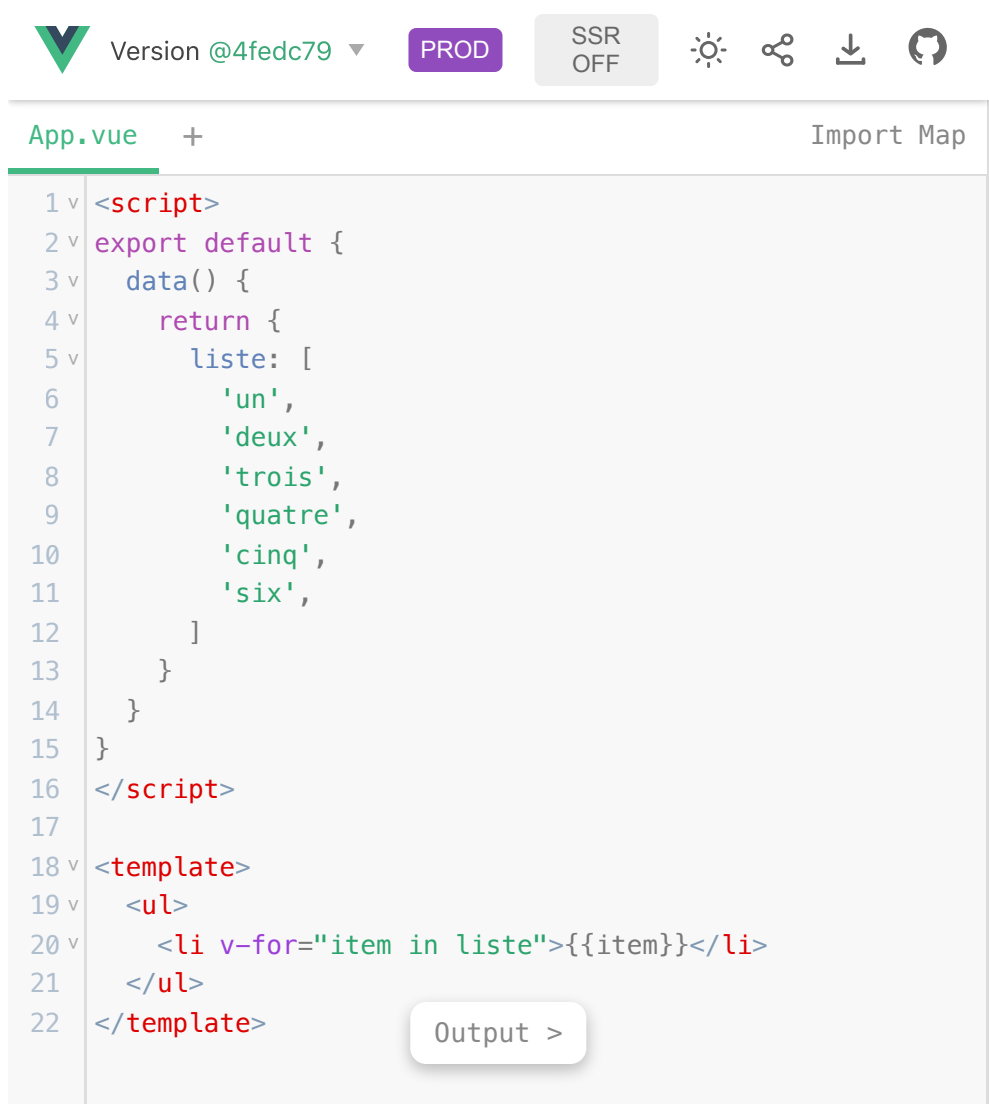
RENDU DE BOUCLES

Pour transformer une liste de données en liste d'éléments, la directive `v-for` est de rigeur.

Il permet de répéter l'élément auquel il est utilisé et donner un contexte pour afficher les données.

On peut destructurer l'information et extraire l'information comme en JS

```
<section v-for="{ date } in informations">
  <datetime>{{ date }}</datetime>
</section>
```

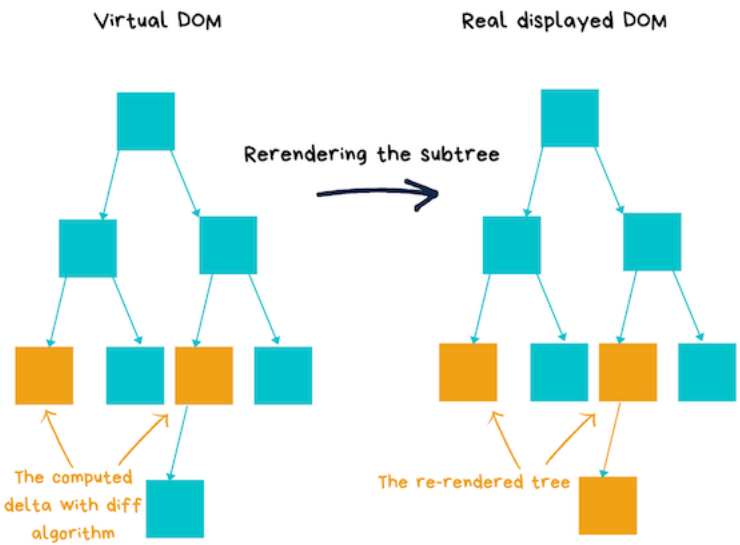


OPTIMISER LA MANIPULATION DU DOM

L’usage du Virtual DOM est déjà une optimisation de la manipulation DOM (très couteuse), présente depuis les débuts de Vue.

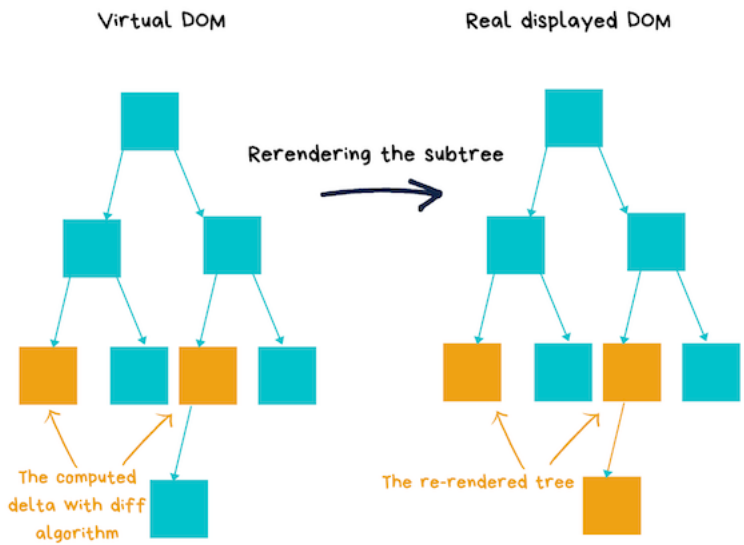
OPTIMISER LA MANIPULATION DU DOM

L'usage du Virtual DOM est déjà une optimisation de la manipulation DOM (très couteuse), présente depuis les débuts de Vue.



OPTIMISER LA MANIPULATION DU DOM

L'usage du Virtual DOM est déjà une optimisation de la manipulation DOM (très couteuse), présente depuis les débuts de Vue.



Info: Vue travaille sur un mode appelé "**Vapor**" qui optimise l'usage du DOM sans Virtual DOM (à la sauce Solid.js)

OPTIMISER LA MANIPULATION DU DOM

React utilise également le Virtual DOM mais un point d'optimisation lui manque : **la capacité changer son comportement de rendu sur les éléments statiques (sans contenu dynamique).**

Lors de la compilation, Vue va travailler en amont pour différentier les éléments :

- hisser les éléments staiques
- marquer les attributs dynamiques
- identifier les blocs d'éléments selon certains critères

> À l'exécution, Vue va éviter plusieurs opérations inutiles au DOM pour ne concentrer que celles qui soient absolument nécessaires.

ÉVÈNEMENTS ET INTERACTIONS

Écouter les événements

Émettre des événements

Transition entre état et style

ÉCOUTER LES ÉVÉNEMENTS

La directive `v-on` ou son raccourci `@` permet d'ajouter une écoute d'évènements sur un élément.

Des modificateurs sont également applicables à cette directive:

- `.prevent`: permet d'annuler le comportement par défaut (`event.preventDefault()`)
- `.stop`: permet de stoper la propagation de l'évènement (`event.stopPropagation()`)
- `.self`: restreind l'écoute d'événement à l'élément seul (en excluant ses enfants)

  > TIP
  > L'ordre est important lors de l'utilisation de modificateurs, car le code correspondant est généré dans le même ordre. Par conséquent, l'utilisation de `@click.prevent.self` empêchera l'action par défaut du clic sur l'élément lui-même et ses enfants, tandis que `@click.self.prevent` empêchera uniquement l'action par défaut du clic sur l'élément lui-même.

- `.enter, .tab, .delete, .esc, .space, .up, .down, .left, .right` pour le clavier
- `.ctrl, .alt, .shift, .meta` pour le clavier et souris
- `.left, .right, .middle` pour la souris
- `.exact`: à utiliser avec d'autres modificateurs, va uniquement écouter l'événément correspondant exactement aux modificateurs

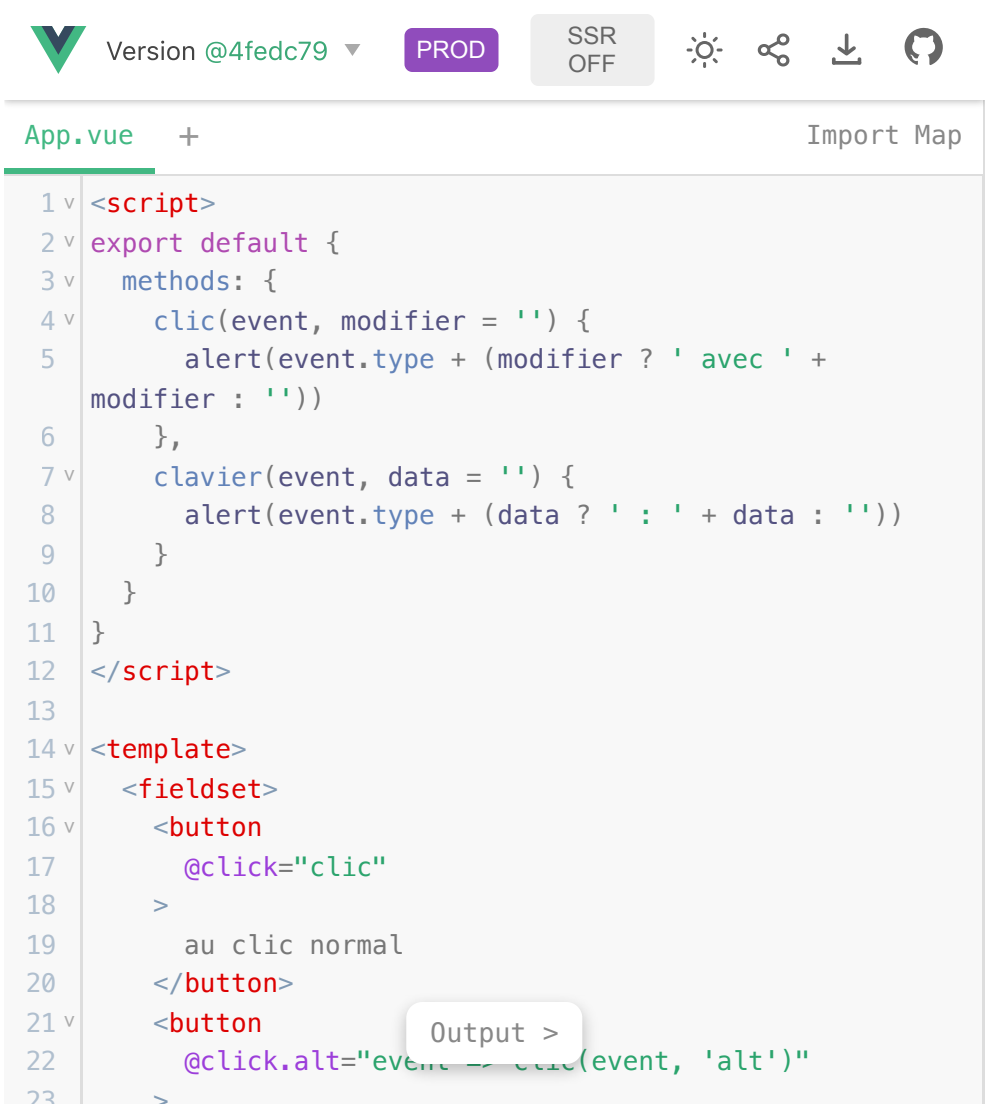
ÉCOUTER LES ÉVÉNEMENTS

La valeur passée peut être le corps d'une fonction ou directement une fonction.

Il est préférable d'utiliser la fonction pour avoir accès à l'évènement plus simplement et éviter l'usage du `$event` pouvant être mal typé.

On peut appeler une fonction du composant, ou directement effectuer des petits changements d'état. Les variables dynamiques sont modifiables, mais à éviter pour des données critiques, il est préférable de laisser les fonctions s'en charger.

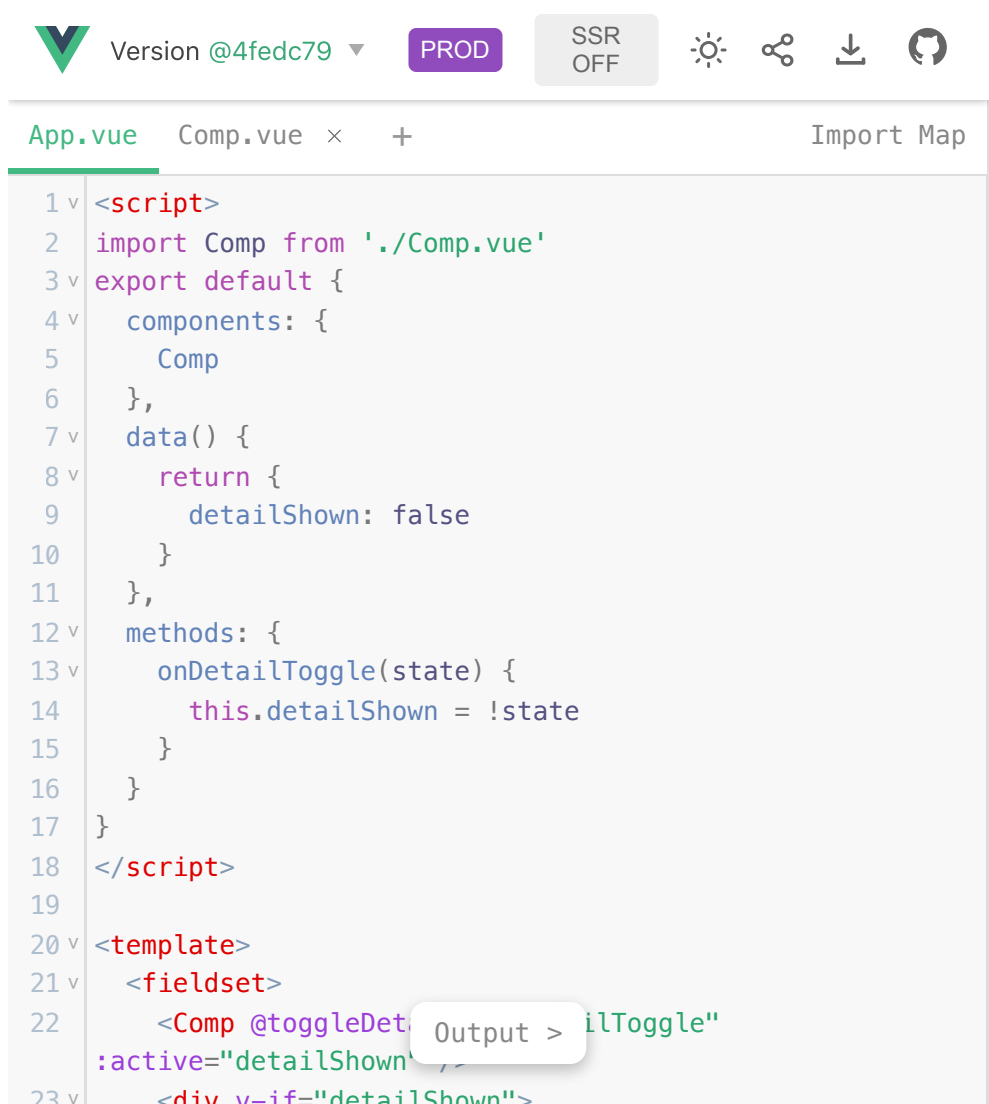
> L'écoute d'évènements ne se limite pas aux évènements natifs, mais également à ceux émis par les composants Vue.



ÉMETTRE DES ÉVÉNEMENTS

Un composant peut transmettre des informations à son parent via l'émission d'évènements. Depuis le template, l'usage de `$emit` permet de le faire en fournissant en paramètres :

- le nom de l'évènement
- les éventuelles données à transmettre



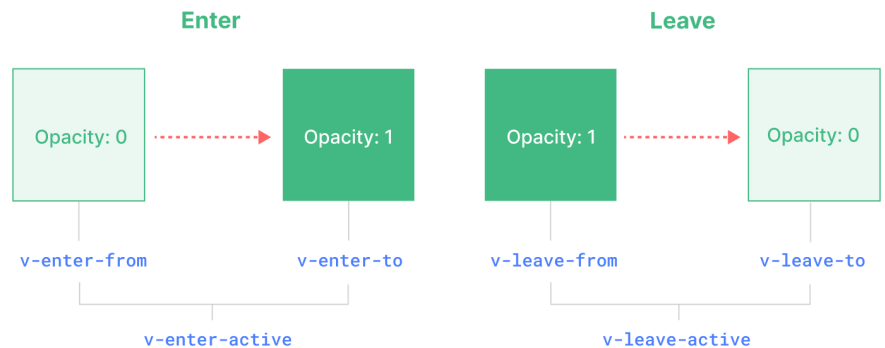
TRANSITION ENTRE ÉTAT ET STYLE

Après certaines interactions utilisateur, il peut y avoir besoin d'afficher ou cacher des éléments de manière fluide:

Pour cela, les `Transition` peuvent avoir leur rôle à jouer.

Les transitions jouent sur les changements d'état et synchronise des classes d'entrée et de sortie de manière à pouvoir styliser les éléments et utiliser des transitions CSS.

Il est possible de modifier le comportement du modification du DOM via le mode : 'out-in' qui permet d'attendre la fin de l'animation de sortie pour ajouter l'élément suivant à afficher.



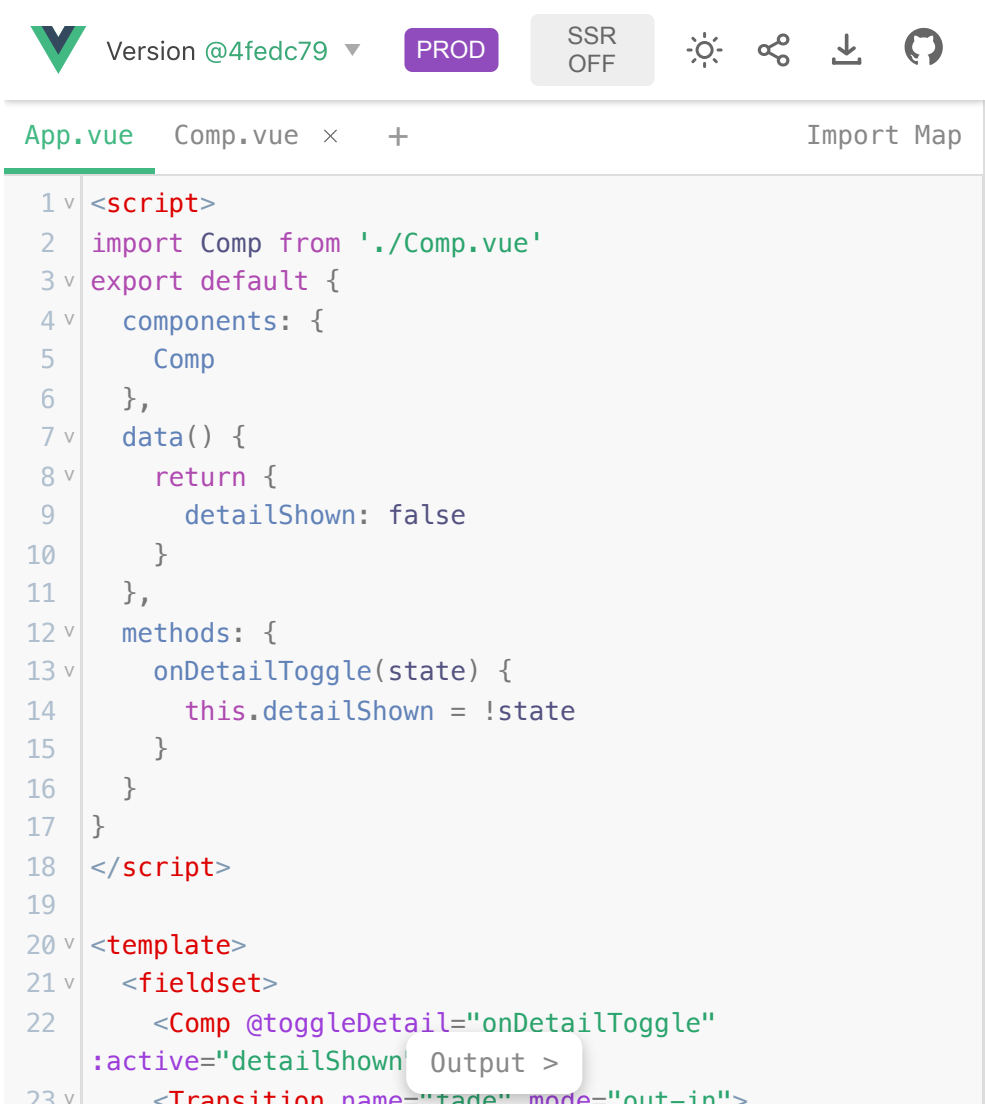
TRANSITION ENTRE ÉTAT ET STYLE

Après certaines interactions utilisateur, il peut y avoir besoin d'afficher ou cacher des éléments de manière fluide:

Pour cela, les `Transition` peuvent avoir leur rôle à jouer.

Les transitions jouent sur les changements d'état et synchronise des classes d'entrée et de sortie de manière à pouvoir styliser les éléments et utiliser des transitions CSS.

Il est possible de modifier le comportement du modification du DOM via le mode : 'out-in' qui permet d'attendre la fin de l'animation de sortie pour ajouter l'élément suivant à afficher.



Options API

ou

Composition API ?

De la simplicité à la complétude

Options API ou Composition API ?

Vue s'est fait connaître pour sa simplicité et sa facile prise en main. Ceci est surtout du à son API utilisant des options sous la forme d'un objet.

```
// Options API

export default {
  props: {
    userId: Number,
  },
  data() {
    return {
      activeUser: null
    };
  },
  mounted() {
    this.fetchActiveUser();
  },
  watch: {
    userId() {
      this.fetchActiveUser();
    }
  },
  methods: {
    fetchActiveUser() {
      fetch(`/user/${this.userId}`).then((user) => this.activeUser = user)
    }
  },
}
```

Options API ou Composition API ?

Vue s'est fait connaître pour sa simplicité et sa facile prise en main. Ceci est surtout du à son API utilisant des options sous la forme d'un objet.

```
// Options API

export default {
  props: {
    userId: Number,
  },
  data() {
    return {
      activeUser: null
    };
  },
  mounted() {
    this.fetchActiveUser();
  },
  watch: {
    userId() {
      this.fetchActiveUser();
    }
  },
  methods: {
    fetchActiveUser() {
      fetch(`/user/${this.userId}`).then((user) => this.activeUser = user)
    }
  },
}
```

Options API ou Composition API ?

Vue s'est fait connaître pour sa simplicité et sa facile prise en main. Ceci est surtout du à son API utilisant des options sous la forme d'un objet.

```
// Options API

export default {
  props: {
    userId: Number,
  },
  data() {
    return {
      activeUser: null
    };
  },
  mounted() {
    this.fetchActiveUser();
  },
  watch: {
    userId() {
      this.fetchActiveUser();
    }
  },
  methods: {
    fetchActiveUser() {
      fetch(`/user/${this.userId}`).then((user) => this.activeUser = user)
    }
  },
}
```

```
// Composition API

export default {
  setup() {
    const props = defineProps({ userId: Number });

    const activeUser = ref(null);

    const fetchActiveUser = fetch(`/user/${props.userId}`)
          .then(user => activeUser.value = user);

    watchEffect(() => {
      if (props.userId)
        fetchActiveUser()
    });

    return { activeUser }
  }
}
```

Il y a près de 3 ans est sortie la nouvelle Composition API :
nouveau paradigme,
nouvelle vision,
très proche des hooks de React 📖,
reprenant toutes les fonctionnalités de l'Option API.

LES FORCES

OPTIONS API

- Simple d’utilisation
- Organisation par sémantique

COMPOSITION API

- Du javascript sans notion de contexte
- Organisation plus flexible
- Permet un découplage plus précis

LES FAIBLESSES

OPTIONS API

- Support TypeScript limité
- Découpage limité (via mixins)

COMPOSITION API

- Complexité plus forte et un certain niveau de JavaScript à avoir
- Les notions de ref compliqué à comprendre

À LA DÉCOUVERTE DE LA COMPOSITION API

On part du principe que l'Options API (OAPI) est acquise et on va donc à la découverte de la Composition API (CAPI).

Elle propose une évolution nette qui peut sembler déroutante de premier abord en offrant une multitude de possibilités pour manipuler les données et les éléments de manière réactive.

> Pour information, la CAPI est accessible à la fois pour la version 2 (à partir de la version 2.7) et la version 3 de Vue.

À LA DÉCOUVERTE DE LA COMPOSITION API

On part du principe que l'Options API (OAPI) est acquise et on va donc à la découverte de la Composition API (CAPI).

Elle propose une évolution nette qui peut sembler déroutante de premier abord en offrant une multitude de possibilités pour manipuler les données et les éléments de manière réactive.

> Pour information, la CAPI est accessible à la fois pour la version 2 (à partir de la version 2.7) et la version 3 de Vue.

C'est parti !

Tout commence avec la nouvelle méthode `setup` qui permet d'enregistrer en une *seule exécution* tout les principes de réactivité **d'un composant**.

DÉCLARER DES VARIABLES RÉACTIVES

Pour rappel, une variable réactive est par convention une variable que l'on modifie directement, et qui suite au changement, peut entraîner des effets de bord. La réactivité utilise la fonctionnalité Proxy de JavaScript.

Pour identifier une variable réactive, il existe la première solution pour toute variables dites primitives (string, number, boolean) et tableaux.

- `ref()`: retourne une référence vers une donnée dynamique, prenant en argument sa valeur par défaut.

```
import { ref } from 'vue'
setup() {
  const nom = ref('André');
  nom.value = 'Jean'
}
```

La seconde solution sera moins courante mais intéressante pour regrouper plusieurs ref, à utiliser pour des objets.

- `reactive()`: prévu pour créer un objet réactif, s'apparente à l'ancien `data()` de l'OAPI, prenant en argument sa valeur par défaut.

```
import { ref } from 'vue'
setup() {
  const data = reactive({ nom: 'André' })
  data.nom = 'André'
}
```

DÉCLARER DES VARIABLES RÉACTIVES

> Pour rappel, une variable réactive est par convention une variable que l'on modifie directement, et qui suite au changement, peut entraîner des effets de bord. La réactivité utilise la fonctionnalité Proxy de JavaScript.

Pour identifier une variable réactive, il existe la première solution pour toute variables dites primitives (string, number, boolean) et tableaux.

- `ref()`: retourne une référence vers une donnée dynamique, prenant en argument sa valeur par défaut.

```
import { ref } from 'vue'
setup() {
  const nom = ref('André');
  nom.value = 'Jean'
}
```

La seconde solution sera moins courante mais intéressante pour regrouper plusieurs ref, à utiliser pour des objets.

- `reactive()`: prévu pour créer un objet réactif, s'apparente à l'ancien `data()` de l'OAPI, prenant en argument sa valeur par défaut.

```
import { ref } from 'vue'
setup() {
  const data = reactive({ nom: 'André' })
  data.nom = 'André'
}
```

DÉCLARER DES VARIABLES RÉACTIVES

> Pour rappel, une variable réactive est par convention une variable que l'on modifie directement, et qui suite au changement, peut entraîner des effets de bord. La réactivité utilise la fonctionnalité Proxy de JavaScript.

Pour identifier une variable réactive, il existe la première solution pour toute variables dites primitives (string, number, boolean) et tableaux.

- `ref()`: retourne une référence vers une donnée dynamique, prenant en argument sa valeur par défaut.

```
import { ref } from 'vue'
setup() {
  const nom = ref('André');
  nom.value = 'Jean'
}
```

La seconde solution sera moins courante mais intéressante pour regrouper plusieurs ref, à utiliser pour des objets.

- `reactive()`: prévu pour créer un objet réactif, s'apparente à l'ancien `data()` de l'OAPI, prenant en argument sa valeur par défaut.

```
import { ref } from 'vue'
setup() {
  const data = reactive({ nom: 'André' })
  data.nom = 'André'
}
```

DÉCLARER DES VARIABLES RÉACTIVES

> Pour rappel, une variable réactive est par convention une variable que l'on modifie directement, et qui suite au changement, peut entraîner des effets de bord. La réactivité utilise la fonctionnalité Proxy de JavaScript.

Pour identifier une variable réactive, il existe la première solution pour toute variables dites primitives (string, number, boolean) et tableaux.

- `ref()`: retourne une référence vers une donnée dynamique, prenant en argument sa valeur par défaut.

```
import { ref } from 'vue'
setup() {
  const nom = ref('André');
  nom.value = 'Jean'
}
```

La seconde solution sera moins courante mais intéressante pour regrouper plusieurs ref, à utiliser pour des objets.

- `reactive()`: prévu pour créer un objet réactif, s'apparente à l'ancien `data()` de l'OAPI, prenant en argument sa valeur par défaut.

```
import { ref } from 'vue'
setup() {
  const data = reactive({ nom: 'André' })
  data.nom = 'André'
}
```

`ref` est à privilégier pour la plupart des cas d'usage. `reactive` est à voir comme un objet const qui ne peut être réassigné mais ses membres si. `reactive` a un intérêt fort pour passer un ensemble de `ref` entre fonctions. **Ces règles sont imposées pour préserver la réactivité**

TRANSMETTRE LES DONNÉES À LA VUE

Via la fonction `setup()` au sein du composant, toutes les données (réactives ou non) retournées par la fonction sera accessible au template.

Ou, avec l'option `setup` sur `<script>`, les variables accessibles dans le block le seront dans le template. On s'attardera plus tard dessus.

```
<script>
  import VERSION from 'constant.js';
  import { ref } from 'vue'
  export default {
    setup() {
      const user = ref('');
      return {
        user,
        version: VERSION
      }
    }
  }
</script>

<script setup>
  import { ref } from 'vue'
  import { VERSION as version } from 'constant.js';
  const user = ref('');
</script>

<template>
  {{ user }} {{ version }}
</template>
```

TRANSMETTRE LES DONNÉES À LA VUE

Via la fonction `setup()` au sein du composant, toutes les données (réactives ou non) retournées par la fonction sera accessible au template.

Ou, avec l'option `setup` sur `<script>`, les variables accessibles dans le block le seront dans le template. On s'attardera plus tard dessus.

```
<script>
  import VERSION from 'constant.js';
  import { ref } from 'vue'
  export default {
    setup() {
      const user = ref('');
      return {
        user,
        version: VERSION
      }
    }
  }
</script>

<script setup>
  import { ref } from 'vue'
  import { VERSION as version } from 'constant.js';
  const user = ref('');
</script>

<template>
  {{ user }} {{ version }}
</template>
```

TRANSMETTRE LES DONNÉES À LA VUE

Via la fonction `setup()` au sein du composant, toutes les données (réactives ou non) retournées par la fonction sera accessible au template.

Ou, avec l'option `setup` sur `<script>`, les variables accessibles dans le block le seront dans le template. On s'attardera plus tard dessus.

```
<script>
  import VERSION from 'constant.js';
  import { ref } from 'vue'
  export default {
    setup() {
      const user = ref('');
      return {
        user,
        version: VERSION
      }
    }
  }
</script>

<script setup>
  import { ref } from 'vue'
  import { VERSION as version } from 'constant.js';
  const user = ref('');
</script>

<template>
  {{ user }} {{ version }}
</template>
```

LES DONNÉES DÉRIVÉES

Dériver une donnée se fait comme avant, via `computed`.

```
import { ref, computed } from 'vue'
const user = ref('  roland ');
const userName = computed(() => {
  const name = user.value.trim(' ');
  return name.at(0).toUpperCase() + name.substr(1)
}
```

> `computed` retourne une `ref`.
> Dès lors qu'une ref change dans la fonction, elle sera recalculée automatiquement.

Version @4fedc79 PROD SSR OFF

App.vue + Import Map

```
<script setup>
import { ref, computed } from 'vue'

const msg = ref('Hello World!')

const uppercaseMsg = computed(() =>
msg.value.toUpperCase())
</script>

<template>
  <input v-model="msg">
  <h1>{{ uppercaseMsg }}</h1>
</template>
```

Output >

LES COMPOSABLES

Pour extraire de la logique et pouvoir la réutiliser, il suffit de créer un function, avec ou sans paramètre. Celle-ci peut retourner au besoin des données réactives.

> Les composables par conventions commencent par `use`, par exemple, un composable pour récupérer l'utilisateur actif pour s'appeler `useActiveUser`.

```
import { ref } from 'vue'

export const useActiveUser = () => {
  const user = ref(null);

  const checkAuthentication = () => {
    // ...
  }

  return {
    user,
    checkAuthentication,
  };
}
```

LES CYCLES DE VIE

Pour effectuer des actions pendant le cycle de vie du composant ou d'un composable, voici quelques hooks disponibles :

- `onMounted`: exécuté lorsque le composant est monté (càd ajouté au DOM)
- `onUpdated`: exécuté lorsque le composant a subi une mise à jour
- `onUnmounted`: exécuté lorsque le composant est démonté (càd retiré du DOM)
- `onErrorCaptured`: exécuté lorsque le composant ou un enfant a émit une erreur

```
import { ref, onMounted, onUnmounted } from 'vue'

export const useActiveUser = () => {
  const user = ref(null);

  const checkAuthentication = () => {
    // ...
  }

  onMounted(() => {
    checkAuthentication();
  });

  onUnmounted(() => {
    disconnect();
  });

  return {
    user,
  };
}
```

LES CYCLES DE VIE

Pour effectuer des actions pendant le cycle de vie du composant ou d'un composable, voici quelques hooks disponibles :

- `onMounted`: exécuté lorsque le composant est monté (càd ajouté au DOM)
- `onUpdated`: exécuté lorsque le composant a subi une mise à jour
- `onUnmounted`: exécuté lorsque le composant est démonté (càd retiré du DOM)
- `onErrorCaptured`: exécuté lorsque le composant ou un enfant a émit une erreur

```
import { ref, onMounted, onUnmounted } from 'vue'

export const useActiveUser = () => {
  const user = ref(null);

  const checkAuthentication = () => {
    // ...
  }

  onMounted(() => {
    checkAuthentication();
  });

  onUnmounted(() => {
    disconnect();
  });

  return {
    user,
  };
}
```

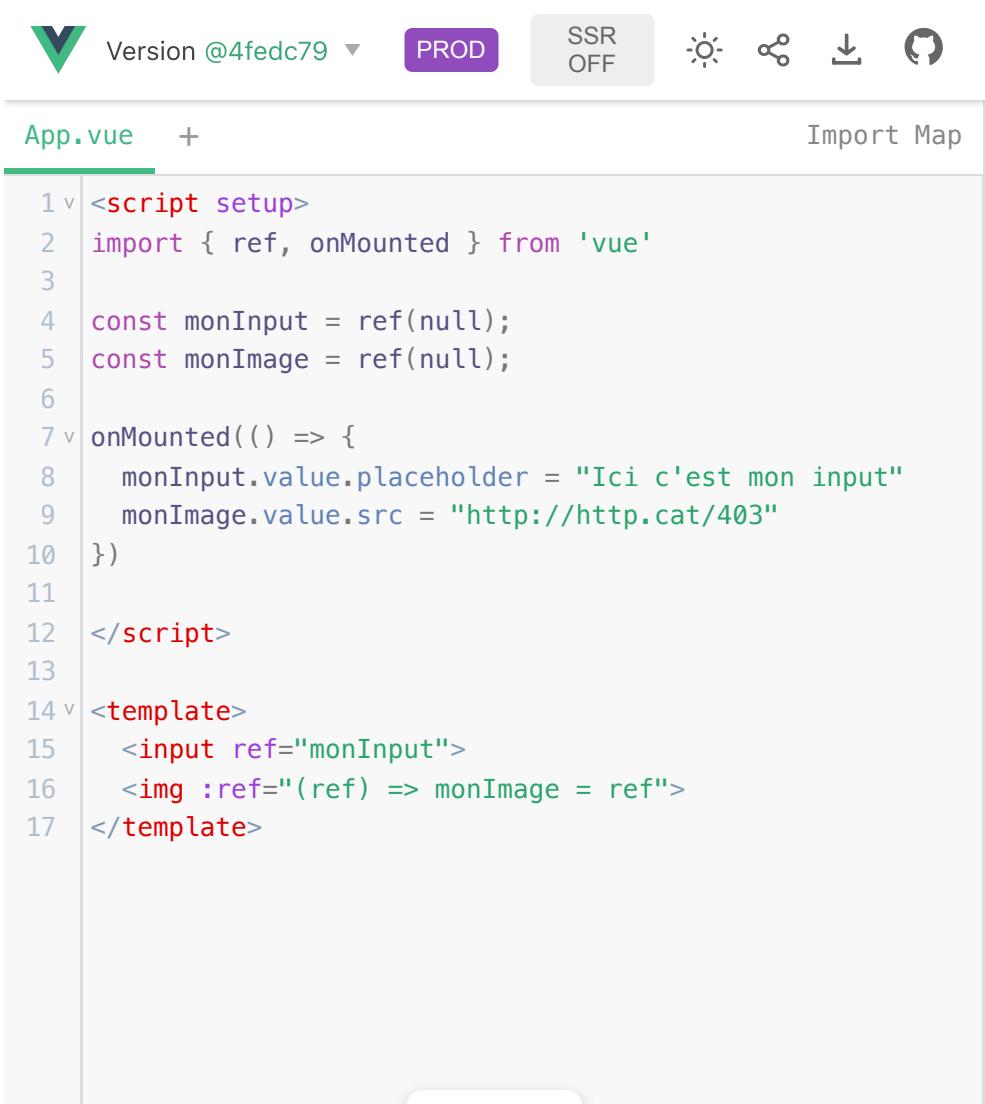
MANIPULER LE DOM AVEC LES TEMPLATE REFS

Avec l'usage de composants tiers ou autre, récupérer l'instance d'un élément DOM est nécessaire.

Pour ce faire,
utiliser la même syntaxe qu'auparavant pour le template c'est-à-dire `ref="monInput"`
et créer la ref correspondante dans `setup`
ainsi: `const monInput = ref(null)`.

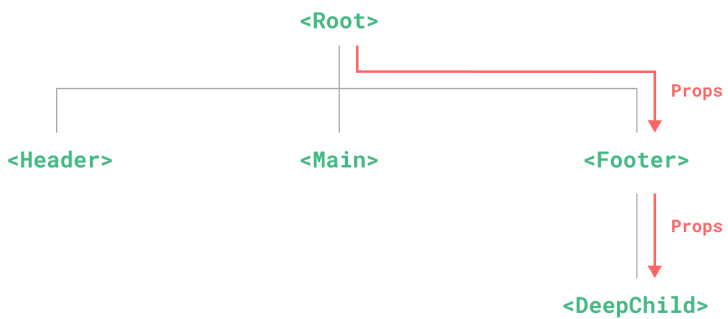
La valeur de la ref après que le composant soit monté est accessible pour y faire ce qu'on y souhaite.

On peut récupérer la ref d'un composant Vue.

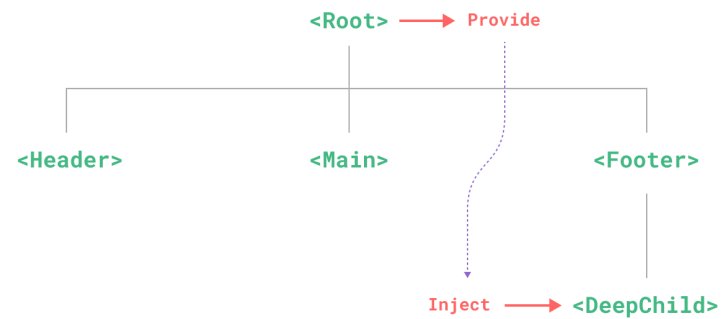


INJECTION DE DÉPENDANCES

Déjà accessible en version 2 (mais usage réservé pour les librairies), `provide` et `inject` permettent l'injection de dépendances pour faire passer des données réactives (ou non) à travers toute l'application.



Props drilling



Provide / Inject

DÉFINIR LES PROPS ET LES ÉVÈNEMENTS ÉMIS

Pour la définitions des props et évènements, on peut utiliser les options `props` et `emits` traditionnels avec l'usage de l'option `setup()`.

Avec `<script setup>`, c'est différent, et ça bouge grâce aux…

MACROS

> Fonctions globales disponibles sans import pour simplifier certaines configurations et permettre un meilleur support TypeScript

- defineProps & defineEmits
- defineExpose

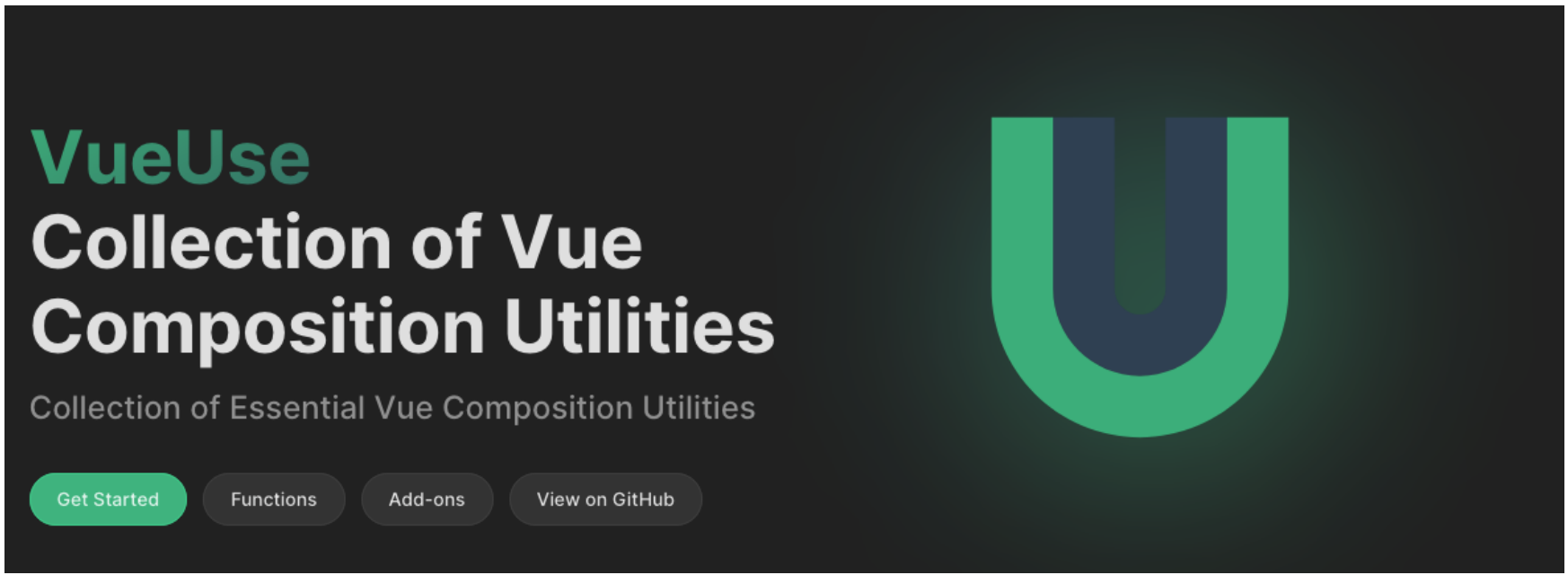
ENCORE PLUS LOIN!

Il y a encore de nombreuses choses à explorer dont:

- la gestion des formulaires et la liaison bidirectionnelle avec `v-model`
- watchEffect et gestion d'effets de bord
- les slots
- Teleport et Suspense

OUTILS À UTILISER POUR NE PAS RECRÉER LA ROUE

Vue Use propose une multitude de fonctionnalités utiles au quotidien fonctionnant la plupart avec les deux versions de Vue.



BONNES PRATIQUES

Vue suggère une multitude de bonnes pratiques à suivre pour éviter les pièges au fur et à mesure du développement d'applications. En voici quelques unes :

- Définir les types des props correctement

> Avec des types bien définis, les erreurs bêtes pourraient être évitées

- Utiliser des clés sur les boucles via `v-for`

> Pour donner du contexte à Vue pour la modications de la liste

- Eviter d'imbriquer `v-for` avec un `v-if`

> `v-if` ayant une priorité sur `v-for`, il peut y avoir des surprises: utiliser un `<template>` ou filtrer la liste en amont

- Utiliser du style cloisoné niveau composant

> À l'exception des librairies, l'usage du `scoped style` évite que du style ne s'applique sans le vouloir en dehors du composant

BONNES PRATIQUES

Vue suggère une multitude de bonnes pratiques à suivre pour éviter les pièges au fur et à mesure du développement d'applications. En voici quelques unes :

- Définir les types des props correctement

> Avec des types bien définis, les erreurs bêtes pourraient être évitées

- Utiliser des clés sur les boucles via `v-for`

> Pour donner du contexte à Vue pour la modications de la liste

- Eviter d'imbriquer `v-for` avec un `v-if`

> `v-if` ayant une priorité sur `v-for`, il peut y avoir des surprises: utiliser un `<template>` ou filtrer la liste en amont

- Utiliser du style cloisoné niveau composant

> À l'exception des librairies, l'usage du `scoped style` évite que du style ne s'applique sans le vouloir en dehors du composant

BONNES PRATIQUES

Vue suggère une multitude de bonnes pratiques à suivre pour éviter les pièges au fur et à mesure du développement d'applications. En voici quelques unes :

- Définir les types des props correctement

> Avec des types bien définis, les erreurs bêtes pourraient être évitées

- Utiliser des clés sur les boucles via `v-for`

> Pour donner du contexte à Vue pour la modications de la liste

- Eviter d'imbriquer `v-for` avec un `v-if`

> `v-if` ayant une priorité sur `v-for`, il peut y avoir des surprises: utiliser un `<template>` ou filtrer la liste en amont

- Utiliser du style cloisoné niveau composant

> À l'exception des librairies, l'usage du `scoped style` évite que du style ne s'applique sans le vouloir en dehors du composant

BONNES PRATIQUES

Vue suggère une multitude de bonnes pratiques à suivre pour éviter les pièges au fur et à mesure du développement d'applications. En voici quelques unes :

- Définir les types des props correctement

> Avec des types bien définis, les erreurs bêtes pourraient être évitées

- Utiliser des clés sur les boucles via `v-for`

> Pour donner du contexte à Vue pour la modications de la liste

- Eviter d'imbriquer `v-for` avec un `v-if`

> `v-if` ayant une priorité sur `v-for`, il peut y avoir des surprises: utiliser un `<template>` ou filtrer la liste en amont

- Utiliser du style cloisoné niveau composant

> À l'exception des librairies, l'usage du `scoped style` évite que du style ne s'applique sans le vouloir en dehors du composant

BONNES PRATIQUES

Vue suggère une multitude de bonnes pratiques à suivre pour éviter les pièges au fur et à mesure du développement d'applications. En voici quelques unes :

- Définir les types des props correctement

> Avec des types bien définis, les erreurs bêtes pourraient être évitées

- Utiliser des clés sur les boucles via `v-for`

> Pour donner du contexte à Vue pour la modications de la liste

- Eviter d'imbriquer `v-for` avec un `v-if`

> `v-if` ayant une priorité sur `v-for`, il peut y avoir des surprises: utiliser un `<template>` ou filtrer la liste en amont

- Utiliser du style cloisoné niveau composant

> À l'exception des librairies, l'usage du `scoped style` évite que du style ne s'applique sans le vouloir en dehors du composant

BONNES PRATIQUES

- Les raccourcis de directives

> Les raccourcis de directives (`:` pour `v-bind:`, `@` pour `v-on:` et `#` pour `v-slot`) doivent toujours être utilisés ou ne jamais l'être.

- Expressions simples dans les templates

> Les templates de composants ne doivent inclure que des expressions simples, avec des expressions plus complexes refactorisées en propriétés calculées ou en méthodes.

- Propriétés calculées simples

> Les propriétés calculées complexes doivent être divisées en propriétés plus simples.

D'autres règles à suivre sur la page des bonnes pratiques en Vue

BONNES PRATIQUES

- Les raccourcis de directives

> Les raccourcis de directives (`:` pour `v-bind:`, `@` pour `v-on:` et `#` pour `v-slot`) doivent toujours être utilisés ou ne jamais l'être.

- Expressions simples dans les templates

> Les templates de composants ne doivent inclure que des expressions simples, avec des expressions plus complexes refactorisées en propriétés calculées ou en méthodes.

- Propriétés calculées simples

> Les propriétés calculées complexes doivent être divisées en propriétés plus simples.

D'autres règles à suivre sur la page des bonnes pratiques en Vue

BONNES PRATIQUES

- Les raccourcis de directives

> Les raccourcis de directives (`:` pour `v-bind:`, `@` pour `v-on:` et `#` pour `v-slot`) doivent toujours être utilisés ou ne jamais l'être.

- Expressions simples dans les templates

> Les templates de composants ne doivent inclure que des expressions simples, avec des expressions plus complexes refactorisées en propriétés calculées ou en méthodes.

- Propriétés calculées simples

> Les propriétés calculées complexes doivent être divisées en propriétés plus simples.

D'autres règles à suivre sur la page des bonnes pratiques en Vue

BONNES PRATIQUES

- Les raccourcis de directives

> Les raccourcis de directives (`:` pour `v-bind:`, `@` pour `v-on:` et `#` pour `v-slot`) doivent toujours être utilisés ou ne jamais l'être.

- Expressions simples dans les templates

> Les templates de composants ne doivent inclure que des expressions simples, avec des expressions plus complexes refactorisées en propriétés calculées ou en méthodes.

- Propriétés calculées simples

> Les propriétés calculées complexes doivent être divisées en propriétés plus simples.

D'autres règles à suivre sur la page des bonnes pratiques en Vue

BONNES PRATIQUES

- Les raccourcis de directives

> Les raccourcis de directives (`:` pour `v-bind:`, `@` pour `v-on:` et `#` pour `v-slot`) doivent toujours être utilisés ou ne jamais l'être.

- Expressions simples dans les templates

> Les templates de composants ne doivent inclure que des expressions simples, avec des expressions plus complexes refactorisées en propriétés calculées ou en méthodes.

- Propriétés calculées simples

> Les propriétés calculées complexes doivent être divisées en propriétés plus simples.

D'autres règles à suivre sur la page des bonnes pratiques en Vue

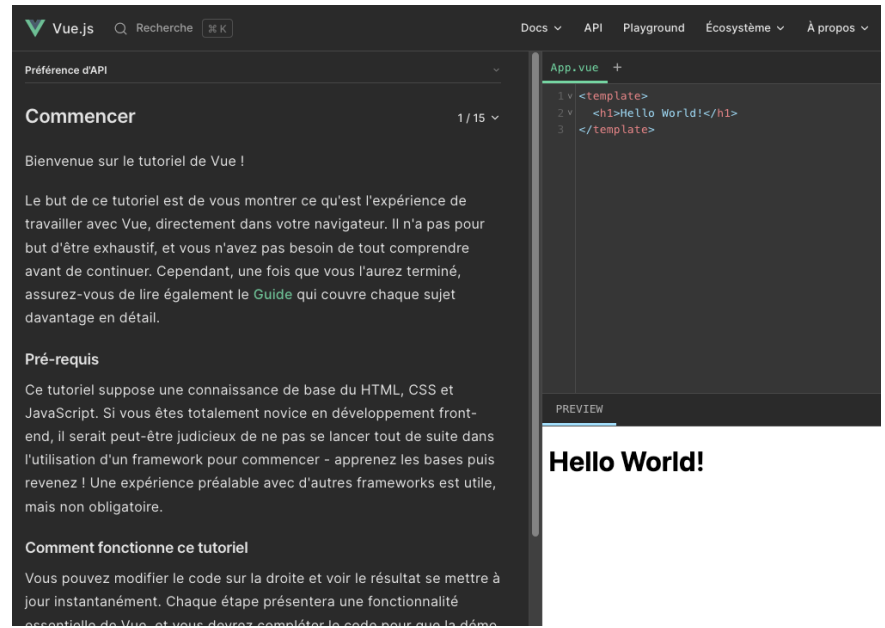
Mise en pratique

Après la théorie, passons à la pratique.

La documentation Vue propose depuis sa nouvelle version un tutoriel ainsi que des exemples.

Rendez-vous sur la page Tutoriel de Vue

RÉSOUDRE LES 15 ÉTAPES



Mise en pratique (suite)

Créons ensemble une mini-application.

Créer un nouveau projet avec la commande `npm create vite@latest` ou via vite.new/vue-ts (en ligne).

> Afficher une liste d'utilisateurs avec leur nom et pouvoir afficher les détails en cliquant dessus.
> Les détails doivent être affichés de manière bien distincte (faites appel à votre imagination)
> Pensez à créer autant de composants que nécessaire pour bien découper les parties de l'application

Utiliser la source de données formation-vue.free.beeceptor.com

> Liste des utilisateurs: `/users`
> Détails d'un utilisateur: `/users/<id>`

Mise en pratique (suite de la suite)

- Ajouter une croix pour fermer les détails
- Exporter les logiques dans les composants en dehors (vers des composables)
- Utiliser les types TypeScript
- Tester les composants

Guider ses développements et les mettre sous contrôle

avec Vitest & Cypress

PARTIE 2

Guider ses développements et les mettre sous contrôle

PRINCIPES

TDD: des tests? Non, des intentions.

Si un test n’est pas le résultat d’une direction que l’on veut donner, il existe un risque qu’il ne valide rien d’intéressant.

Tester, c’est assurer qu’un comportement (action ou input) fonctionne comme attendu (output).

L’écriture d’un test pendant l’écriture initiale d’un code permet d’avoir des directives et de sécuriser son travail en minimisant la perte de temps.

PRINCIPES

TDD: des tests? Non, des intentions.

Si un test n’est pas le résultat d’une direction que l’on veut donner, il existe un risque qu’il ne valide rien d’intéressant.

Tester, c’est assurer qu’un comportement (action ou input) fonctionne comme attendu (output).

L’écriture d’un test pendant l’écriture initiale d’un code permet d’avoir des directives et de sécuriser son travail en minimisant la perte de temps.

PRINCIPES

TDD: des tests? Non, des intentions.

Si un test n’est pas le résultat d’une direction que l’on veut donner, il existe un risque qu’il ne valide rien d’intéressant.

Tester, c’est assurer qu’un comportement (action ou input) fonctionne comme attendu (output).

L’écriture d’un test pendant l’écriture initiale d’un code permet d’avoir des directives et de sécuriser son travail en minimisant la perte de temps.

PRINCIPES

TDD: des tests? Non, des intentions.

Si un test n’est pas le résultat d’une direction que l’on veut donner, il existe un risque qu’il ne valide rien d’intéressant.

Tester, c’est assurer qu’un comportement (action ou input) fonctionne comme attendu (output).

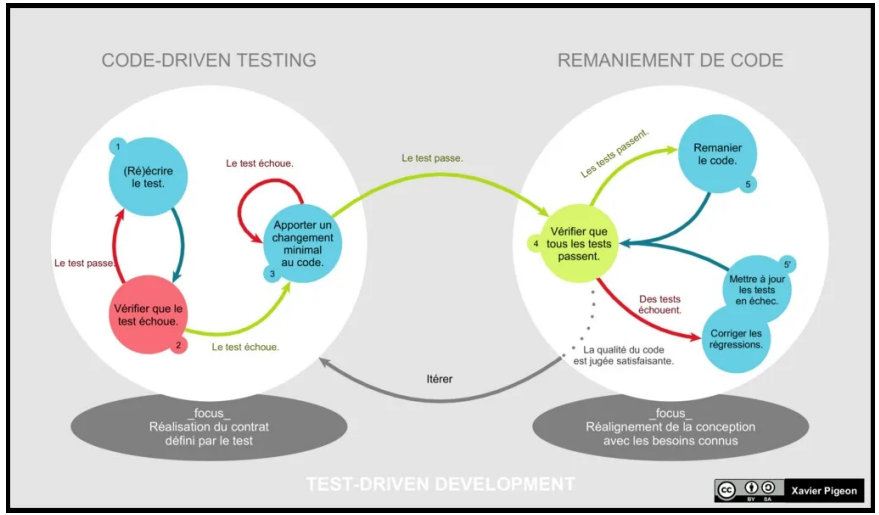
L’écriture d’un test pendant l’écriture initiale d’un code permet d’avoir des directives et de sécuriser son travail en minimisant la perte de temps.

MÉTHODOLOGIE

Avant toute chose,
bien comprendre la fonctionnalité cible à atteindre, elle doit être bien spécifiée.

Identifier les petites étapes pour y parvenir et commencer par la première :

- Ecrire un test qui couvre les attentes de l'étape
- Développer la fonctionnalité le plus rapidement jusqu'à que le test passe au vert
- Nettoyer le code de manière à le rendre conforme au projet tout en assurant que les tests restent au vert
- Passer à l'étape suivante



VITEST

Outils de tests pour exécuter les tests qui permet de réutiliser la configuration Vite.

```
// transform.spec.ts
describe('transform raw data to specilized data', () => {

    it('should return string from object', () => {
        const transformedData = transformData({
            key: 'value'
        });
        expect(transformedData).toBeTypeOf('string');
    })

    it('should render correctly the object', () => {
        const transformedData = transformData({
            bar: 'foo'
        });
        expect(transformedData).toContain('foo');
        expect(transformedData).toContain('bar');
    })

})
```

Lancer le test avec :

```
npx vitest
```

TESTER LES COMPOSANTS VUE

EXEMPLE LIVE

Via le plugin Vue de Vite, Vitest par défaut peut interpréter vos composants. (exemple de config avec Vue)

Il est néanmoins nécessaire d'ajouter une librarie pour pouvoir monter virtuellement vos composants et pourvoir y faire des assertions: `@vue/test-utils`

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

describe('mount component', async () => {

    it('should be truthy', () => {
        expect(Hello).toBeTruthy()
    })

    it('should increment on click and calculate accordingly', () => {

    const wrapper = mount(Hello, {
        props: {
        count: 4,
        },
    });

    expect(wrapper.text()).toContain('4 x 2 = 8')

    await wrapper.get('button').trigger('click')

    expect(wrapper.text()).toContain('4 x 3 = 12')

    await wrapper.get('button').trigger('click')

    expect(wrapper.text()).toContain('4 x 4 = 16')
    })
})
```

MISE EN PRATIQUE

Exporter la logique de récupération des données utilisateur dans un composable (existant ou un nouveau selon votre choix).

> Suivez la méthodologie de test pour être guidé par les tests.