Web Client - **#3**

[Vue.js - components]

Cyril Rouyer

https://vuejs.org

Sommaire

- Définition / Utilisation
- Props
- Événements personnalisés
- Slots (transcluding)

Définition / Utilisation

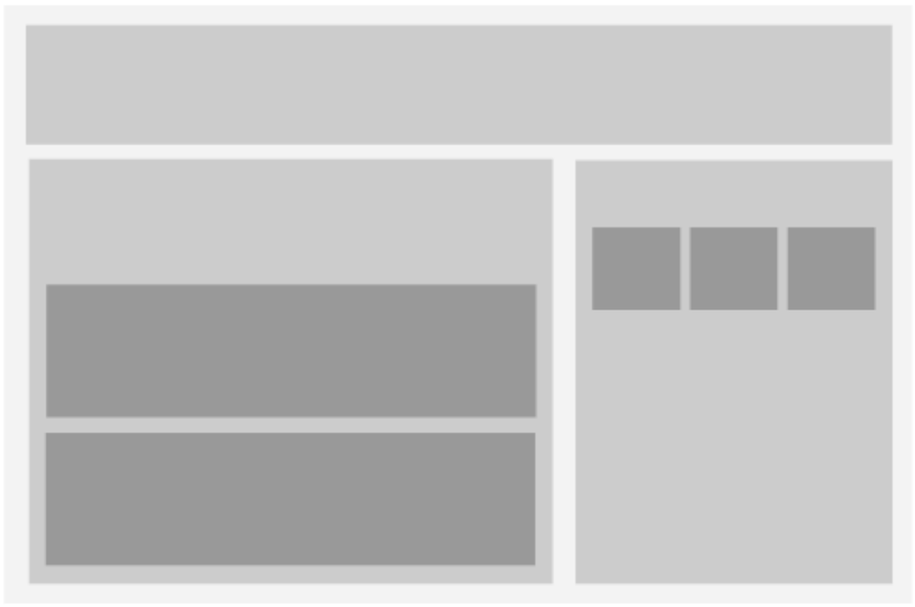
Définition

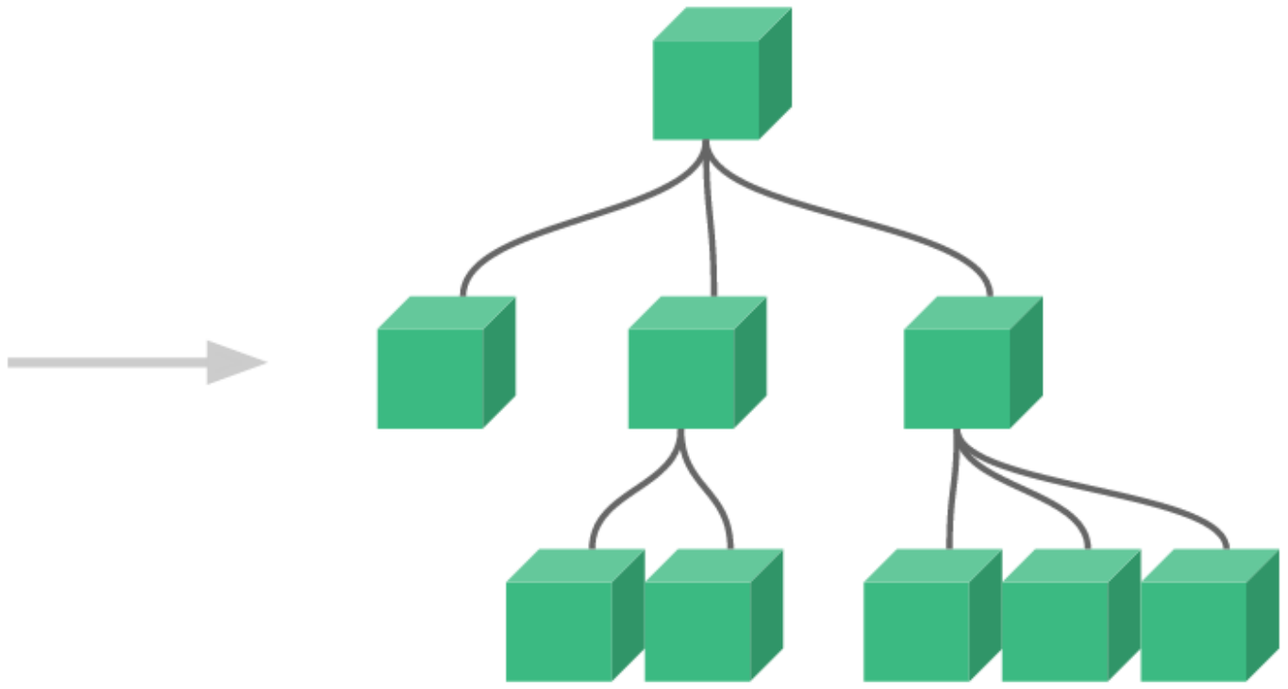
Les composants

- Permettent d'étendre le code HTML de base avec des balises custom
  - <todolist><todo v-for="t in todos" :obj="t"></todo></todolist>
- Encapsulent du code réutilisable
- Sont quasi identiques à l'instance racine de la Vue (même propriétés, mêmes hooks, …)

Définition

Les composants





Utilisation 1

Inscription globale

- On peut déclarer l'utilisation d'un composant de façon globale

- Pour cela on utilise **Vue.component**('my-component', { … })

- À partir de là, le composant peut-être utilisé partout dans l'application

Utilisation 2

Inscription globale

```
let app = Vue.createApp(…)

app.component(‘my-component', {
  template: '<div>Un composant
personnalisé !</div>'
})

app.mount(‘#app’)
```

```
<div id="example">
  <my-component></my-
component>
</div>
```

Utilisation 3

Inscription globale

- L’inscription globale oblige à pré-déclarer les composants avant de démarrer l’instance racine de votre Vue

- Le composant déclaré de façon globale sera visible dans tout le scope de votre application

- C’est plutôt utilisé dans le contexte de l’import de librairies externes

Utilisation 4

Inscription locale

- L’inscription locale permet de déclarer les composants dont on a besoin au moment où on en a besoin

- Dans ce cas, le composant ne sera disponible que dans le scope du composant parent où il est appelé

Utilisation 5

Inscription locale

```
const Child = {
  template: '<div>Un composant personnalisé !</div>'
}
```

```
Vue.createApp({
  // ...
  components: {
    // <my-component> ne sera disponible que dans le template parent
    'my-component': Child
  }
}).mount(“#app”)
```

Utilisation 6

Restrictions HTML

- Le moteur de rendu de Vue passe après le parsing natif du browser

- Or certaines balises (ex : select, table, ul, ol) attendent des éléments spécifiques directement sous elles (ex : select -> option ; table -> tr)

- Donc si votre composant a pour vocation a être directement attaché sous une de ces balises, ce sera évalué comme “invalide”.

```
<table>
  <my-row>...</my-row>
</table>
```

Utilisation 7

Restrictions HTML

- Vue.js apporte une solution pour contourner le souci : is
- is est une directive qui permet d'indiquer que la balise en question doit être interprétée comme un composant Vue, tout en conservant la compatibilité HTML :

```
<table>
  <tr is="vue:my-row"></tr>
</table>
```

Utilisation 8

Restrictions HTML

- Cette restriction ne s'applique plus à partir du moment où vos templates sont obtenus depuis une des sources suivantes :

- Les balises <script type="text/x-template">

- Les templates de chaine de caractères littérales en Javascript (`texte ${expression} text`)

- Les composants .vue (format possible avec ES2015 & vue-cli)

Utilisation 9

Data

- Petit rappel : l'instance de votre vue peut prendre des propriétés réactives, présentes dans la clef "data" :

```
const vm = Vue.createApp({
    data() {
        return {
            message: "Hello World"
        }
    }
})
```

- Les composants ont eux-aussi leurs propres data

Utilisation 10

Composition

- On peut composer notre vue en utilisant des composants dans des composants …

- Notion de parent > enfant

- Forcément à un moment donné, les parents doivent donner des infos aux enfants, et les enfants doivent indiquer aux parents que quelque chose s’est produit

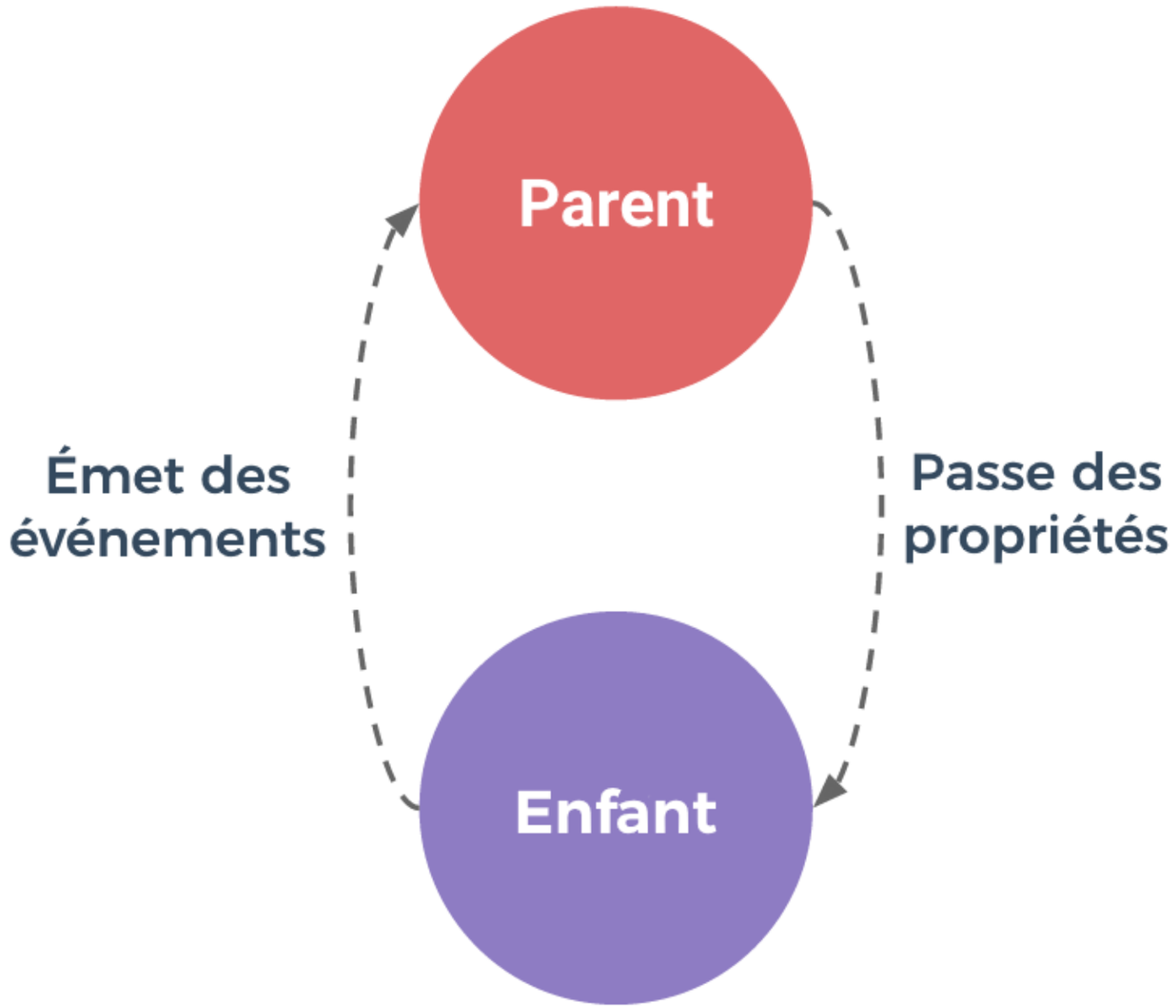
Utilisation 11

Composition

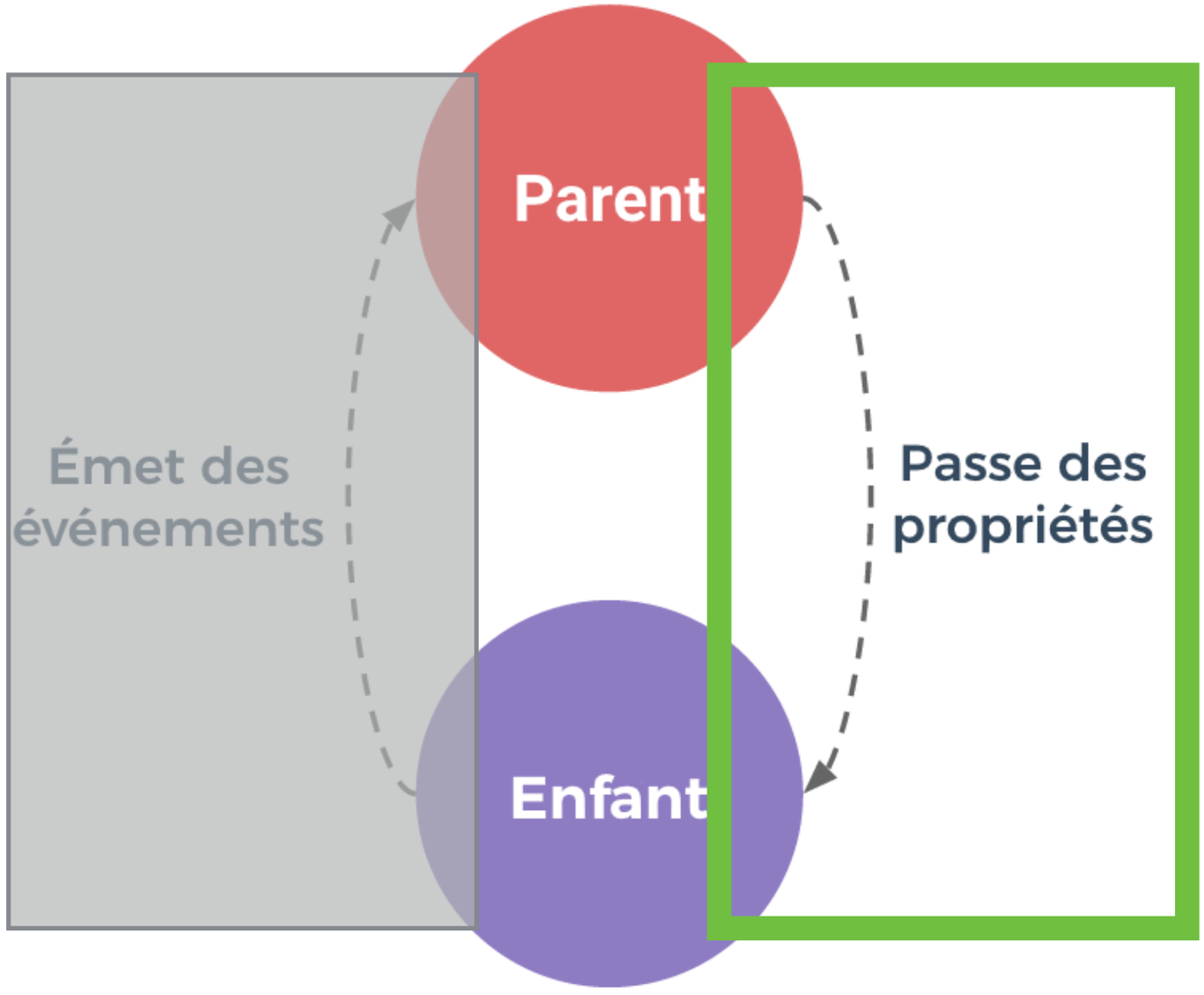
- Il faut garder à l'esprit que les parents et enfants doivent toujours être découplés de façon à faciliter la maintenance de leur code, et surtout leur réutilisabilité
- L'enfant ne doit **jamais** toucher aux données du père
- Il peut seulement remonter des événements

Utilisation 12

Composition



Props



Props 1

Passage de données Parent -> Enfant

- Chaque instance d’un composant possède son propre scope
- L’enfant ne peut pas accéder directement aux données du parent
- Les props permettent au parent de donner les infos dont l’enfant a besoin
- Les props possibles sont déclarées par l’enfant

Props 2

Passage de données Parent -> Enfant

```
app.component('Child', {
  // déclarer les props
  props: ['message'],
  // tout comme un élément de `data`, une prop peut être utilisée à
l'intérieur de templates
  // et est également disponible dans l'instance via `this.message`
  template: '<span>{{ message }}</span>'
})
```

```
<child message="bonjour !"></child>
```

Props 3

camelCase vs kebab-case

- Convention :
  - Le nom de la propriété est définie en camelCase
  - L’attribut HTML est défini en kebab-case
- C’est une nécessité pour les templates qui ne sont pas des chaînes de caractères (!= .vue)

```
app.component(‘child', {
  // camelCase en JavaScript
  props: ['myMessage'],
  template: '<span>{{ myMessage }}</span>'
})
```

```
<!-- kebab-case en HTML -->
<child my-message="bonjour !"></child>
```

Props 4

Props dynamiques

- Pour binder une propriété dynamique, on peut utiliser v-bind
- Ou ‘:’ (sa version abrégée)
- Mais c’est un flux unidirectionnel (si la valeur change dans le parent, ça redescendra dans l’enfant, mais pas l’inverse)
- Vue enverra un warning dans la console si un enfant tente de muter une prop parente

```
<child :my-message="parentMsg"></child>
```

Props 5

Validation

- Les composants peuvent définir des conditions qui permettent de valider les données passées par le parent à travers les props
- Pour cela les propriétés du composant doivent être déclarées comme un objet d’objets
- Chaque objet correspond à une propriété et ses contraintes

Props 6

Validation

```
app.component('Example', {
  props: {
    propA: Number,
    propB: [String, Number],
    propC: {
      type: String,
      required: true
    },
    propD: {
      type: Number,
      default: 100
    },
    propE: {
      type: Object,
      default: function () {
        return { message: 'hello' }
      }
    },
    propF: {
      validator: function (value) {
        return value > 10
      }
    }
  }
})
```

Props 7

Attributs non-props

- Que se passe-t-il si le développeur passe un attribut sur un composant qui n’est pas une prop déclarée dans le composant ?

- La valeur de l’attribut ne sera pas accessible dans le composant

- Par contre sur l’élément racine du composant, dans le DOM généré, on retrouvera cet attribut, tel quel, avec sa valeur

Props 8

Attributs non-props

```
const plop = {
  template: `<div>PLOP :
{{message}}</div>`,
  props: ['message']
}

Vue.createApp({
  components: {
    plop
  },
  data() {
    return {message: “Hello World”}
  }
  ...
})
```

```
<body>
  <div id="app">
      <plop :message="message"
unknown-attr=“salut”></plop>
  </div>
</body>
```

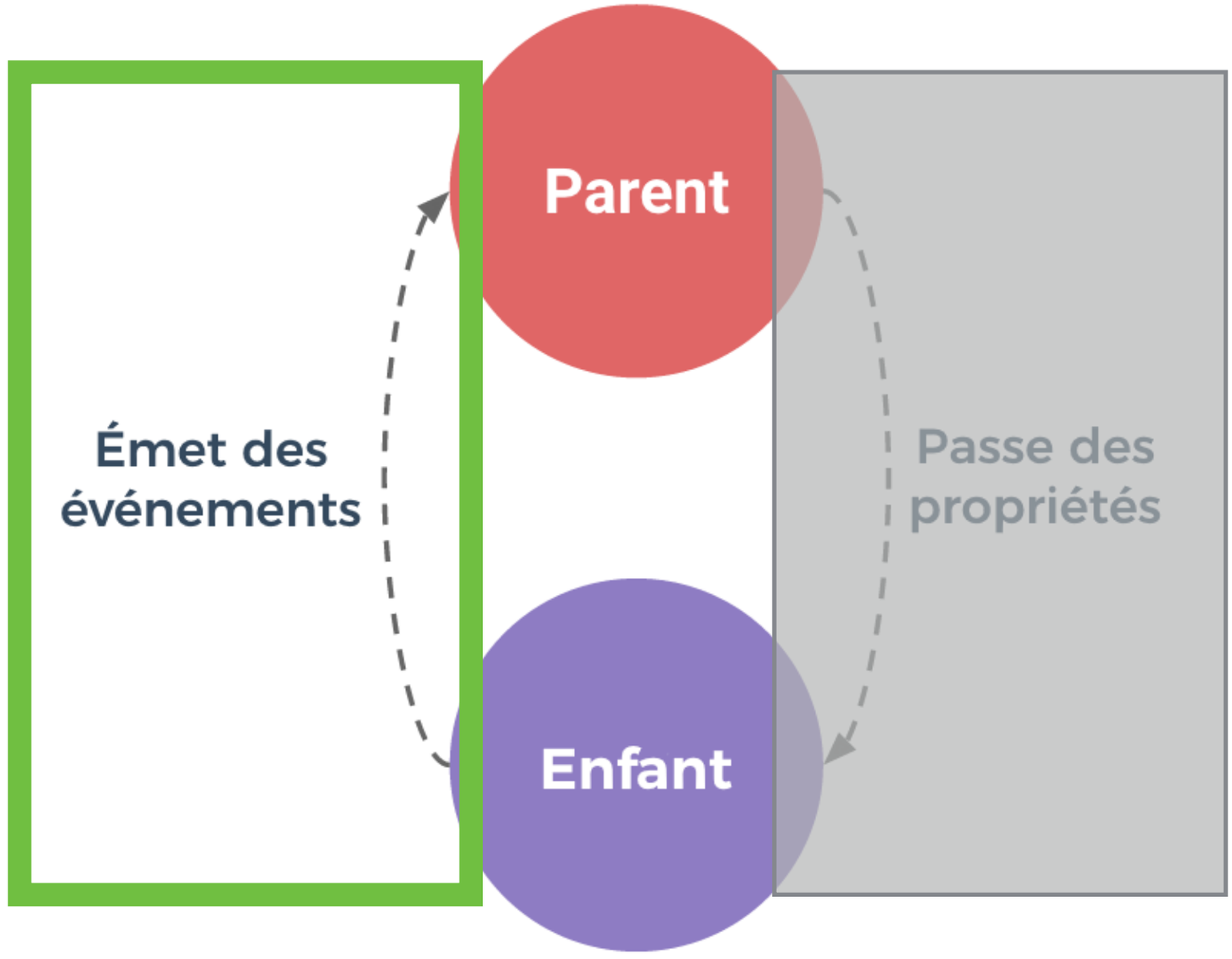
```
<!DOCTYPE html>
▼ <html>
  ▶ <head>…</head>
  ▼ <body>
    ▼ <div id="app">
        <div unknown-attr="salut">PLOP : Hello World !</div> = $0
```

Props 9

Remplacement et merge des attributs existants

- Que se passe-t-il si le développeur passe un attribut sur un composant alors que l'élément racine du composant possède déjà cet attribut ?
- Par défaut c'est la valeur passée par le père qui gagne
- Mais il y a deux exceptions : class & style
  - Pour ces deux attributs, Vue.js fera un merge des valeurs

Event



Event 1

$on / $emit

- On a vu que les parents passent des propriétés aux enfants
- On sait que les enfants ne peuvent pas modifier les propriétés des parents pour rester “neutres” vis-à-vis du contexte
- Problème : comment les enfants peuvent-ils interagir ?
- Solution : en émettant des événements

Event 2

$on / $emit

- Chaque instance de Vue implémente une interface d’évènements :
  - Permet d’écouter un événement : $on
  - Permet d’émettre un événement : $emit
- Ce ne sont pas des alias pour addEventListener et dispatchEvent, c’est une mécanique propre à Vue
- On utilise rarement directement $on, on passe par le template avec v-on (ou @ en version abrégée)

Event 3

```
app.component('ButtonCounter', {
  template: '<button v-on:click="incrementCounter">{{ counter }}</button>',
  data: function () {
    return {
      counter: 0
    }
  },
  methods: {
    incrementCounter: function () {
      this.counter += 1
      this.$emit('increment')
    }
  },
})

Vue.createApp({
  data() {
    return {total: 0}
  },
  methods: {
    incrementTotal: function () {
      this.total += 1
    }
  }
}).mount('#counter-event-example')
```

```
<div id="counter-event-example">
  <p>{{ total }}</p>
  <button-counter v-
on:increment="incrementTotal"></button-
counter>
  <button-counter v-
on:increment="incrementTotal"></button-
counter>
</div>
```

3

3 0

Event 4

- $emit peut prendre un second argument : une valeur que le parent pourra récupérer

- this.$emit(‘name-of-the-event’, **myObject**)

Event 5

Composant comme champ de formulaire

- On a vu que **v-model** permet d'attacher une data de façon réactive sur un input, textarea ou select

- On peut aussi l'utiliser sur un composant custom à condition de respecter deux règles

  - Le composant doit attendre une propriété **modelValue**

  - Le composant doit émettre un événement **update:modelValue** au @input avec la nouvelle valeur pour affecter la valeur au modèle du père.

Event 6

Composant comme champ de formulaire

Fils

```
app.component('LabeledTextField', {
  template: `<label>
    <input type="text"
           :value="modelValue"
           @input="$emit('update:modelValue',
           $event.target.value)" />
    {{text}}
    </label>`,
  props: ['text', 'modelValue']
})
```

Père

```
...
<labeled-text-field :text="Nom de famille" v-
model="lastname"></labeled-text-field>

...
```

Slots

Slots 1

Transcluding

```
<app>
  <app-header></app-header>
  <app-footer></app-footer>
</app>
```

- Le transcluding (transclusion en français) correspond au principe de la distribution de contenu

- Permet à un composant custom de rendre du contenu dynamique issu du parent, sans connaître sa nature à l’avance

Slots 2

Transcluding & Portée

```
<child-component>
  {{ message }}
</child-component>
```

- La portée de compilation des éléments présents dans un template est toujours celle du composant auquel appartient le template
- Dans l’exemple, message vient du scope du parent
- Il n’y a pas d’exception à cette règle

Slots 3

Élément <slot>

- L'élément **<slot>** utilisé dans le template du composant permet d'indiquer la position à laquelle sera distribué l'éventuel contenu provenant du père

- Il peut lui-même prendre du contenu, dit "par défaut", qui sera affiché si le père ne précise pas de contenu

Slots 4

Élément <slot>

Enfant ‘my-component’

```
<div>
  <h2>Je suis le titre de l'enfant</
h2>
  <slot>
    Ceci ne sera affiché que s'il n'y a
pas de contenu à distribuer.
  </slot>
</div>
```

Parent

```
<div>
  <h1>Je suis le titre du parent</h1>
  <my-component>
    <p>Ceci est le contenu original</p>
    <p>Ceci est encore du contenu original</p>
  </my-component>
</div>
```

Résultat

```
<div>
  <h1>Je suis le titre du parent</h1>
  <div>
    <h2>Je suis le titre de l'enfant</h2>
    <p>Ceci est le contenu original</p>
    <p>Ceci est encore du contenu original</p>
  </div>
</div>
```

Slots 5

Élément <slot name="xxx">

- Un composant peut avoir plusieurs balises <slot>
- Dans ce cas elles sont nommées et permettent de cibler une zone précise du template **<slot name="header">**
- Il peut rester une balise <slot> sans nom, ce sera la zone par défaut
- Le parent cible le slot désiré via une balise template et la directive v-slot (ex : **v-slot:header**)
- v-slot: a sa propre abréviation : **#**

Slots 6

Élément <slot name=“xxx”>

Enfant ‘app-layout’

```
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

Parent

```
<app-layout>
  <template #header>
    <h1>Voici un titre de page</h1>
  </template>
  <p>Un paragraphe pour le contenu
principal.</p>
  <p>Et un autre.</p>
  <template #footer>
    <p>Ici plusieurs informations de contact</p>
  </template>
</app-layout>
```

À noter : le slot par défaut est implicitement nommé #default

Slots 7

Élément <slot name=“xxx”>

Résultat

```
<div class="container">
  <header>
    <h1>Voici un titre de page</h1>
  </header>
  <main>
    <p>Un paragraphe pour le contenu principal.</p>
    <p>Et un autre.</p>
  </main>
  <footer>
    <p>Ici plusieurs informations de contact</p>
  </footer>
</div>
```

Slots 8

Élément <slot name="xxx">

- Il est possible de cibler un slot nommé de manière dynamique
- dynamique = le nom du slot provient d'une variable
- <template #[dynamicSlotName]>…</template>

Slots 9

Slot avec portée

- Il est possible d'utiliser les slots pour laisser la main au développeur qui utilise le composant afin de customiser le contenu

- On utilise pour ça les slots avec portée, qui permettent à l'enfant de passer au père des **props** à afficher (on inverse les rôles)

- Tous les attributs bindés sur le slot deviennent alors disponibles à travers un objet côté parent, que l'on nomme comme on veut dans un **v-slot**

Slots 10

Slot avec portée

Enfant ‘MyComponent2’

```
<div>
  <slot :text="greetingMessage" :count="1"
  ></slot>
</div>
```

Parent

```
<MyComponent2 v-slot="slotProps">
  {{ slotProps.text }}
  {{ slotProps.count }}
</MyComponent2>
```

Slots 11

Slot avec portée

- On peut faire la même chose avec un slot nommé
- <template v-slot:header="headerProps">…</template>

Slots 12

Scoped slots avec v-for

- On peut personnaliser une liste avec les scoped slots
- Le fils donne les propriétés de chaque item de la liste et se charge de la structure générale
- Le père peut définir comme il l'entend l'affichage des items de la liste grâce aux props

Slots 13

Scoped slots avec v-for

Enfant

```
<ul>
  <slot
    v-for="item in items”
    name="item"
    v-bind="item"></slot>
</ul>
```

Parent

```
<FancyList :api-url="url" :per-page="10">
 <template #item="{ body, username, likes }">
  <li>
    <div class="item">
      <p>{{ body }}</p>
      <p>by {{ username }} | {{ likes }} likes</p>
    </div>
  </li>
 </template>
</FancyList>
```

À noter : on peut passer un objet complet en **props** avec **v-bind**