

TP2 (suite): Vue.js

Ibrahim ALAME

4 décembre 2025

1 Chargement de TP2

1. Dans un terminal taper la commande :

```
git clone https://github.com/ialame/vuejs-boutique-tp2-1.git
```

Git doit être installé sur votre ordinateur sinon cliquer [ici](#).

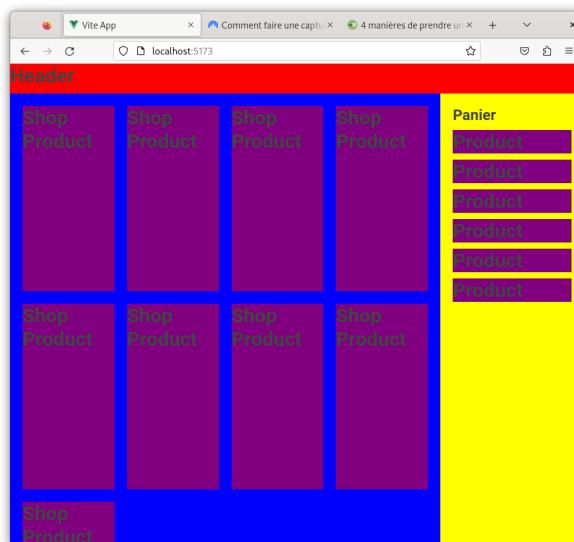
2. renommer le dossier vuejs-boutique-tp2-1 en vuejs-boutique-tp2-2 :

```
mv vuejs-boutique-tp2-1 vuejs-boutique-tp2-2
```

3. Exécuter dans le terminal les deux commandes :

```
cd vuejs-boutique-tp2-2
```

```
npm install
```



2 Mise en page du header et du footer

2.1 Modification de `assets/base.scss`

Ajoutez les classes utilitaires dont nous aurons besoin :

```

.px-20 {
  padding-left: 20px;
  padding-right: 20px;
}
.p-30 { padding: 30px; }
// margin
.m-10 { margin: 10px; }
.mb-10 { margin-bottom: 10px; }
.mr-10 { margin-right: 10px; }
.mr-20 { margin-right: 20px; }

```

2.2 Modification de Header.vue

Nous mettons en place notre header de marges 20px à gauche et à droite (px-20) flexible (d-flex) horizontal (flex-row) dont les éléments sont centrés (align-items-center) contenant :

1. un ancrage `a` sans cible `href="#"` flexible (d-flex) horizontal (flex-row) dont les éléments sont centrés (align-items-center) et de marge droite 20px (mr-20) contenant :
 - l'image `logo.svg`.
 - un span de classe `logo` contenant le nom de la société PCA.
2. une liste `ul` flexible (d-flex) horizontal (flex-row) avec un ressort (flex-fill) contenant deux éléments :
 - le premier `li` est de marge droite 10px (mr-10) contenant un ancrage `a` sans cible `href="#"` dont le mot sensible est **Boutique**.
 - le deuxième `li` contient un ancrage `a` sans cible `href="#"` dont le mot sensible est **Admin**.
3. une liste `ul` flexible (d-flex) horizontal (flex-row) contenant deux éléments :
 - le premier `li` est de marge droite 10px (mr-10) contenant un ancrage `a` sans cible `href="#"` dont le mot sensible est **Inscription**.
 - le deuxième `li` contient un ancrage `a` sans cible `href="#"` dont le mot sensible est **Connexion**.

Le style de header est le code sass suivant :

```

header {
  background-color: var(--primary-1);
  a {
    color: var(--text-primary-color);
    img {
      width: 20px;
      margin-right: 5px;
    }
    .logo {
      font-weight: 700;
      font-size: 20px;
    }
  }
}

```

2.3 Modification de Footer.vue

Nous mettons en place notre footer :

```

<template>
  <footer class="d-flex flex-row justify-content-center align-items-center">
    <p>Copyright © 2014-2022 PCA</p>
  </footer>
</template>

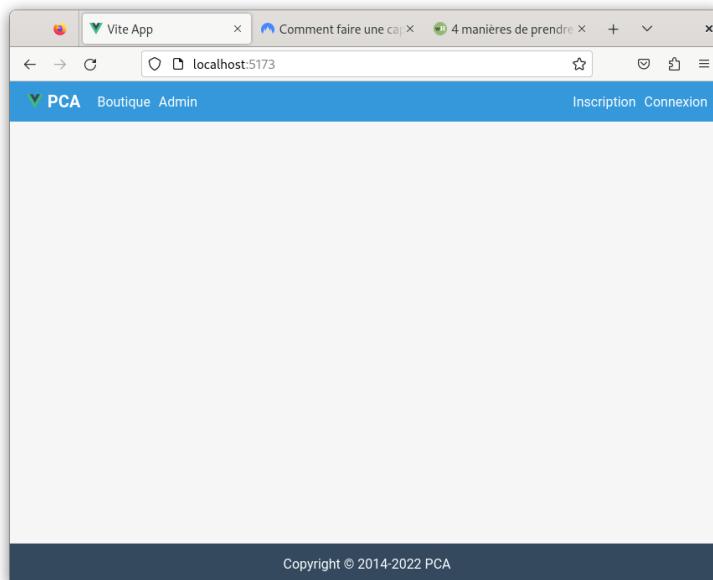
```

Dont le style de `footer` est défini par :

```

footer {
  background-color: var(--gray-3);
  color: var(--text-primary-color);
}

```



3 Mise en page de la partie Shop

3.1 Modification de `ShopProduct.vue`

Nous allons utiliser le style `sass` suivant :

```

.product {
  background-color: #ffffff;
  border: var(--border);
  border-radius: var(--border-radius);
  &::before {
    border-top-right-radius: var(--border-radius);
    border-top-left-radius: var(--border-radius);
    background-image: url('/src/assets/images/1.jpg');
    background-size: contain;
    background-repeat: no-repeat;
    background-position: center;
  }
}

```

```
    height: 250px;
}
]
```

La balise principale du template du composant ShopProduct est flexible verticale `d-flex flex-column` de classe `product` définie ci-haut. Elle contient les éléments suivants :

1. Une `div` de classe `product-image` et contiendra l'image définie dans la classe.
2. Une `div` flexible verticale de `padding 10px` contenant :
 - (a) Un titre `h4` : Alakazam.
 - (b) Un paragraphe `p` contenant une description. Par exemple : Carte Pokéémon : Alakazam 1/102 Set de Base Wizards Française
 - (c) Un division `div` flexible verticale dont les éléments sont centrés et contenant les deux balises :
 - une balise `div` avec un ressort `flex-fill` contenant le prix : Prix: 1500€ (par exemple).
 - Un bouton ordinaire de classes `btn btn-primary` permettant d'ajouter le produit : Ajouter au panier

3.2 Classes pour les boutons

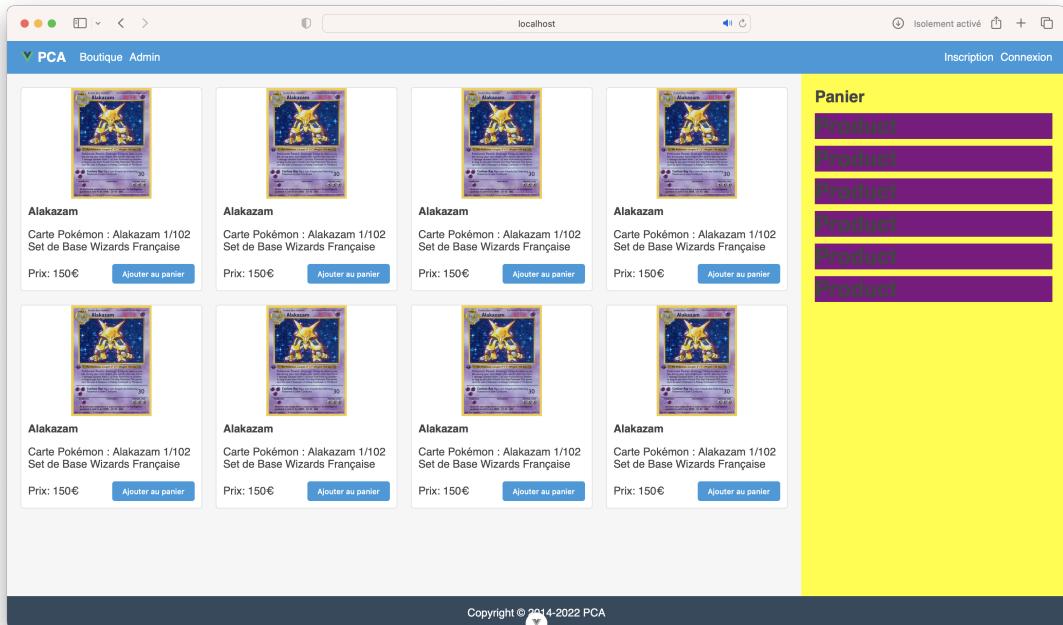
Modifiez `assets/base.scss` pour ajouter les classes pour nos boutons :

```
// buttons

.btn {
  padding: 8px 15px;
  border: 0;
  border-radius: var(--border-radius);
  cursor: pointer;
  font-weight: 500;
  transition: background-color 0.2s;
}

.btn-primary {
  background-color: var(--primary-1);
  color: var(--text-primary-color);
  &:hover {
    background-color: var(--primary-2);
  }
}

.btn-danger {
  background-color: var(--danger-1);
  color: var(--text-primary-color);
  &:hover {
    background-color: var(--danger-2);
  }
}
```



4 Mise en place du panier

4.1 Modification de CartProduct.vue

Nous mettons en place en dur le produit du panier pour terminer la mise en page. Le template contient une division `div` flexible verticale dont les éléments sont centrés et ayant les deux classes `mb-10 p-10` et contenant les trois balises

1. une balise `strong` avec un ressort `flex-fill` et une marge droite de `10px` contenant le produit : `Macbook Pro` (par exemple).
2. une balise `span` avec une marge droite de `10px` contenant le prix : `Prix: 1500€` (par exemple).
3. Un bouton danger `btn btn-primary` permettant de supprimer le produit : `Supprimer`

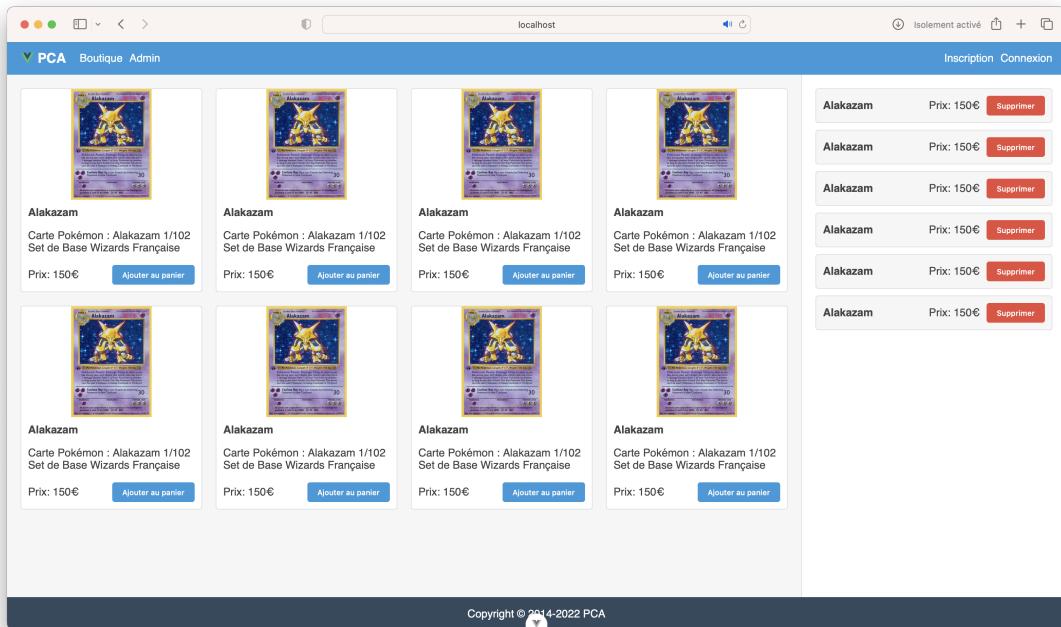
Le style `product` est défini par le code `sass` suivant :

```
.product {
  border: var(--border);
  border-radius: var(--border-radius);
  background-color: var(--gray-1);
}
```

4.2 Modification de App.vue

Nous modifions légèrement la classe `cart` :

```
.cart {
  grid-area: cart;
  border-left: var(--border);
  background-color: white;
}
```



5 Utilisation des props pour les produits

5.1 Crédation des interfaces

Créez un dossier `interfaces` dans `src`. Créer dans ce dossier un fichier `Product.interface.ts`:

```
export interface ProductInterface {
  id: number;
  title: string;
  image: string;
  price: number;
  description: string;
  quantity?: number | undefined ;
}
```

5.2 Données en dur

Pour avancer plus dans le projet et pour pouvoir utiliser des requêtes `HTTP` afin de récupérer les produits, nous avons mis les données `en dur` dans le fichier `product.ts` dans le dossier `data` dans `src`.

5.3 Modification de App.vue

Dans le composant racine, nous allons importer les données et les typer dans une propriété réactive. Nous passons en type générique un tableau de produits qui doivent respecter l'interface `ProductInterface`. Nous initialisons la propriété réactive avec nos données contenues dans `data`.

```
import type { ProductInterface } from '@/interfaces/Product.interface.ts'  
import data from './data/product';  
const products = reactive<ProductInterface[]>(data);
```

Nous allons ensuite les passer à notre composant enfant `Shop` en utilisant une liaison de données avec `v-bind`.

```
<Shop :products="products" class="shop" />
```

5.4 Modification de Shop.vue

Dans le composant `Shop.vue`, nous définissons la `prop` reçue depuis le composant parent grâce à `defineProps`, à savoir la liste de tous les produits.

Nous typons la `props` en passant le type générique `{ products: ProductInterface[] }` : cela signifie que nous recevons un objet contenant une propriété `products` qui a pour valeur un tableau contenant des `ProductInterface`.

```
defineProps<{  
    products: ProductInterface[];  
>();
```

Nous repassons ensuite la `prop` reçue plus bas dans l'arbre :

```
<ShopProductList :products="products" />
```

5.5 Modification de ShopProductList.vue

Dans le composant `ShopProductList`, nous récupérons également la `prop` que nous typons comme dans le composant parent `Shop`.

```
defineProps<{  
    products: ProductInterface[];  
>();
```

Cette fois, nous utilisons la directive `v-for` pour boucler sur les produits et nous utilisons une liaison de propriété pour passer le produit à chaque instance du composant `ShopProduct`.

```
<ShopProduct v-for="product of products" :product="product" />
```

5.6 Modification de ShopProduct.vue

Dans le composant `ShopProduct`, nous récupérons le produit en `props` :

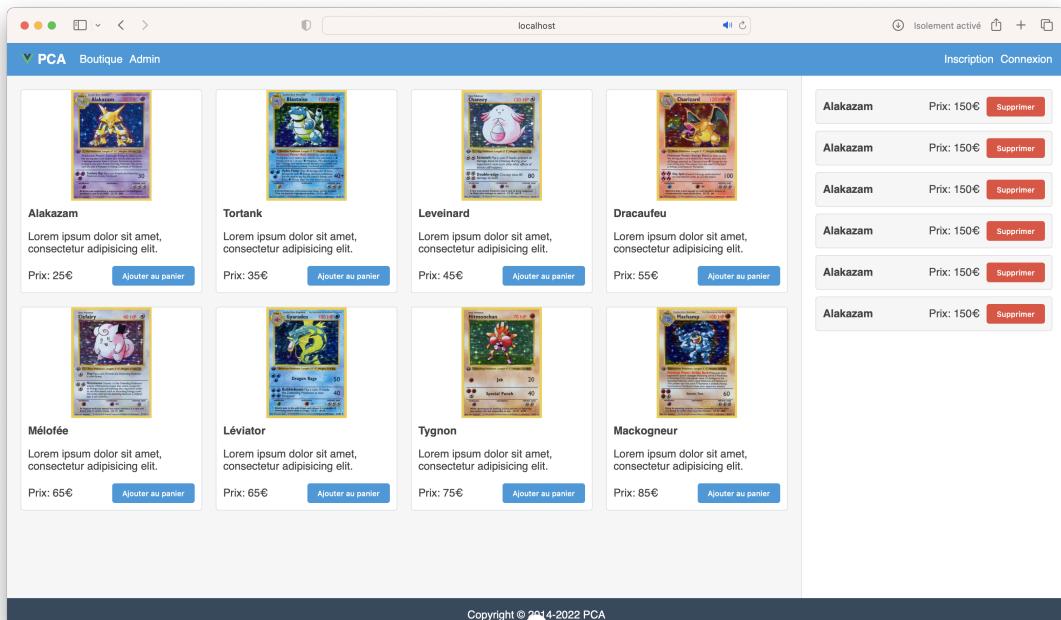
```
defineProps<{  
    product: ProductInterface;  
>();
```

et pour l'affichage nous utilisons la notation raccourcie de la directive `v-text` c'est-à-dire les doubles accolades `{{ }}`. Pour afficher les différentes propriétés du produit : `{{ product.title }}`, `{{ product.description }}`, `{{ product.price }}` dans les balises `h4`, `p` et `strong`. Quand à l'image supprimer dela classe `product` la ligne :

```
background-image: url('/src/assets/images/1.jpg');
```

Puis modifier la `div` de l'image en ajoutant un style bindé :

```
<div
  class="product-image"
  :style="{ backgroundImage: `url(${product.image})` }"
></div>
```



6 Ajout d'un produit dans le panier

6.1 Logique à mettre en place

De la même manière que nous faisons descendre la liste de produits dans l'arbre des composants dans cet ordre `App → Shop → ShopProductList → ShopProduct`, nous souhaitons maintenant propager de l'information dans ce sens :

`ShopProduct → ShopProductList → Shop → App.`

En effet, nous souhaitons que lorsqu'un utilisateur clique sur le bouton d'ajout au panier dans un composant `ShopProduct`, que cet événement remonte le long de l'arbre des composants

et qu'il puisse nous informer sur quel produit l'utilisateur souhaite ajouter. Pour ce faire, nous allons utiliser un événement personnalisé, qui contiendra l'`id` du produit à ajouter au panier. Cet événement sera émis depuis le composant `ShopProduct` concerné, puis sera réémis par `ShopProductList` puis par `Shop`.

Une fois que l'événement est arrivé au composant `App`, nous pourrons utiliser l'`id` du produit pour l'ajouter au panier. En effet, dans notre composant `App` nous centralisons les informations relatives au panier et aux produits.

6.2 Modification de `App.vue`

Dans le composant racine, nous modifions notre propriété réactive pour ajouter une propriété `cart` contenant les produits du panier. Nous initialisons le panier avec un tableau vide.

```
const state = reactive<{
  products: ProductInterface[];
  cart: ProductInterface[];
}>({
  products: data,
  cart: [],
});
```

Nous créons une fonction `ajouterAuPanier()` qui va permettre d'ajouter un produit dans le panier en utilisant son `id` unique. La logique est simple : nous récupérons le produit à l'aide de son `id` depuis la propriété réactive `state`, ensuite nous vérifions qu'il n'est pas déjà dans le panier, et enfin nous l'ajoutons au panier.

Notez bien que nous utilisons `{ ...product }` pour créer une copie de l'objet qui ne soit pas réactif (un `Proxy`) vers l'objet du produit contenu dans la propriété réactive.

Rappelez-vous en effet que les propriétés réactives contiennent des `Proxys` vers les objets que nous passons.

De cette manière, nous obtenons un nouvel objet qui a une nouvelle référence et qui est totalement différent de l'objet dans la propriété réactive.

```
function ajouterAuPanier(productId: number): void {
  const product = state.products.find((product) => product.id === productId);
  if (product && !state.cart.find((product) => product.id === productId)) {
    state.cart.push({ ...product });
  }
}

<Shop
  :products="state.products"
  @add-product-to-cart="ajouterAuPanier"
  class="shop"
/>
```

6.3 Modification de `Shop.vue`

Ici nous ajoutons la directive `v-on` pour écouter l'événement `add-product-to-cart` lorsqu'il est émis par le composant `ShopProductList`

Nous ne faisons que le retransmettre en démontrant un événement avec `defineEmits` qui est identique à celui reçu.

```
const emit = defineEmits<{
  (e: 'addProductToCart', productId: number): void;
}>();
```

L'événement qui va être retransmis est `addProductToCart` (notez bien le `camelCase`) et il contient un seul argument qui est le `productId` de type `number`.

```
<ShopProductList
  @add-product-to-cart="emit('addProductToCart', $event)"
  :products="products"
/>
```

6.4 Modification de `ShopProductList.vue`

Même logique, nous ne faisons que retransmettre l'événement provenant du composant `Shop-Product` vers le composant `Shop` :

```
const emit = defineEmits<{
  (e: 'addProductToCart', productId: number): void;
}>();
.....
<ShopProduct
  @add-product-to-cart="emit('addProductToCart', $event)"
  v-for="product of products"
  :product="product"
/>
```

6.5 Modification de `ShopProduct.vue`

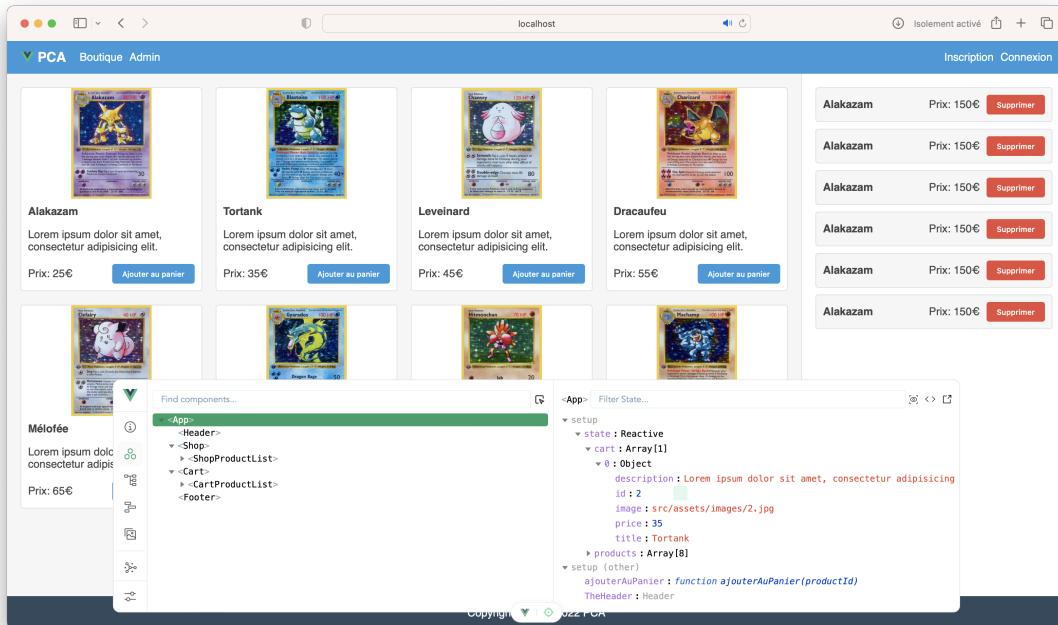
La logique est la suivante :

1. `@click` : Ici nous créons une liaison d'événement avec `v-on` sur l'événement `click` sur le bouton d'ajout au panier.
2. Nous créons un événement personnalisé '`addProductToCart`' qui va contenir un seul argument : le `productId` :

```
const emit = defineEmits<{ (e: 'addProductToCart', productId: number): void; }>();
```

3. Nous l'émettons avec `emit()` lors d'un clic sur le bouton. Le premier argument est le nom de l'événement en `camelCase` et le deuxième argument est l'`id` du produit.

```
@click="emit('addProductToCart', product.id)"
```



7 Affichage et suppression pour la partie panier

7.1 Logique à mettre en place

De la même manière que précédemment pour la partie **Shop**, nous souhaitons faire descendre l'information du contenu du panier le long de l'arbre des composants de cette manière :

App → Cart → CartProductList → CartProduct.

De cette manière, nous pourrons afficher le contenu du panier. Nous allons simplement utiliser une **prop** sur les différents niveaux de composants pour passer cette information.

Nous souhaitons également faire remonter l'information dans le sens inverse lorsque l'utilisateur clique sur le bouton "Supprimer" d'un élément du panier :

CartProduct → CartProductList → Cart → App.

Pour ce faire, nous allons utiliser un événement personnalisé, qui contiendra l'**id** du produit à supprimer du panier. Cet événement sera émis depuis le composant **CartProduct** concerné, puis sera réémis par **CartProductList** puis par **Cart**.

Une fois que l'événement sera arrivé au composant **App**, nous pourrons utiliser l'**id** du produit pour le supprimer du panier. La logique est donc similaire à celle utilisée pour la partie **Shop**.

7.2 Modification de **App.vue**

Dans **App.vue** : Nous définissons une fonction **supprimerDuPanier()** pour supprimer un article du panier lorsque nous recevons l'événement **removeProductFromCart**.

```

function supprimerDuPanier(productId: number): void {
  state.cart = state.cart.filter((product) => product.id !== productId);
}

.....
<Cart
  :cart="state.cart"
  class="cart"
  @remove-product-from-cart="supprimerDuPanier"
/>
.....

```

- `@remove-product-from-cart="supprimerDuPanier"` : nous écoutons l'événement `remove-product-from-cart` et enregistrons la fonction `supprimerDuPanier()` comme gestionnaire d'événement.

Notez bien que nous n'utilisons pas de parenthèses avec `supprimerDuPanier`. En JavaScript cela signifie que nous passons la fonction avec tous les arguments récupérés.

Notez également bien que l'événement côté `template` est en `kebab-case` (`remove-product-from-cart`) et qu'il soit être en `camelCase` côté script (`removeProductFromCart`).

- `:cart="state.cart"` : nous passons le contenu du panier avec une `props` au composant `Cart`.

7.3 Modification de Cart.vue

Voici le composant `Cart.vue` :

```

<script setup lang="ts">
...
defineProps<{
  cart: ProductInterface[];
}>();
const emit = defineEmits<{
  (e: 'removeProductFromCart', productId: number): void;
}>();
</script>

<template>
...
<CartProductList
  :cart="cart"
  @remove-product-from-cart="emit('removeProductFromCart', $event)"
/>
...
</template>

```

Il ne fait que transmettre dans un sens une `prop` et dans l'autre un événement. Vous pouvez le voir comme un "relais" des informations dans les deux sens.

7.4 Modification de CartProductList.vue

Voici le composant :

```

<script setup lang="ts">
...
defineProps<{
  cart: ProductInterface[];
}>();
const emit = defineEmits<{
  (e: 'removeProductFromCart', productId: number): void;
}>();
</script>

<template>
...
<CartProduct
  v-for="product of cart"
  :product="product"
  @remove-product-from-cart="emit('removeProductFromCart', $event)"
/>
...
</template>

```

Même chose, ce composant rapporte les informations dans les deux sens à chaque instance du composant **CartProduct**. Pour rappel, il y a un composant **CartProduct** pour chaque produit grâce à la directive **v-for**.

7.5 Modification de **CartProduct**

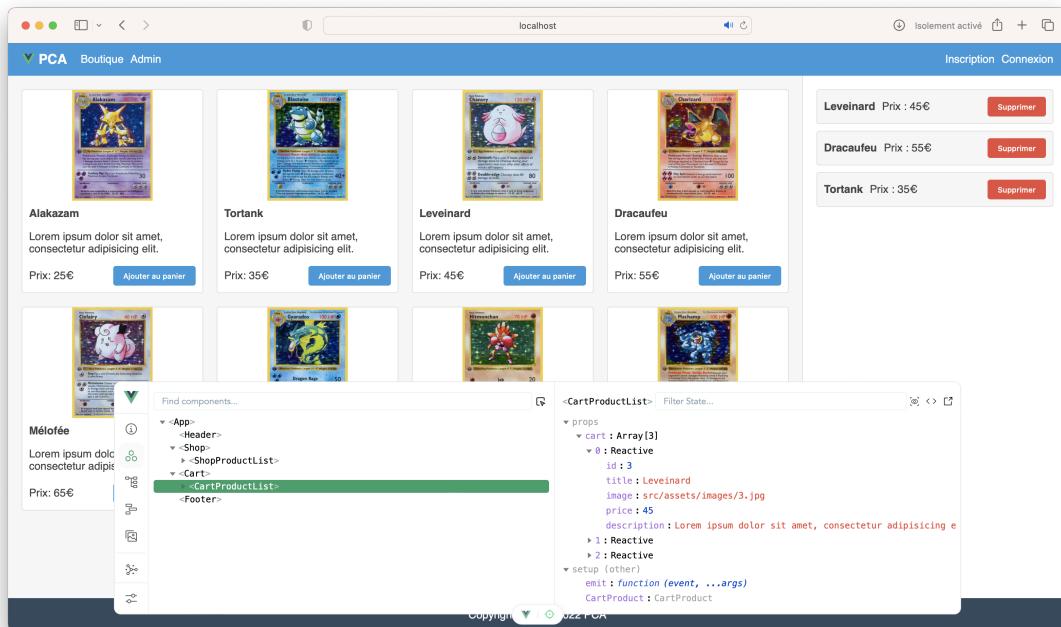
Voici le composant :

```

<script setup lang="ts">
...
defineProps<{
  product: ProductInterface;
}>();
const emit = defineEmits<{
  (e: 'removeProductFromCart', productId: number): void;
}>();
</script>
....
```

Pour l'affichage, utiliser les doubles accolades.

Le composant émet un événement **removeProductFromCart** lorsque l'utilisateur clique sur le bouton de suppression d'un article du panier. L'événement contient l'**id** unique du produit à supprimer. Il affiche les informations du produit contenues dans la **prop** reçue.



8 Gestion des quantités

8.1 Modification de CartProduct.vue

Nous affichons la quantité de chaque produit dans le panier :

```
<span class="mr-10">x {{ product.quantity }}</span>
```

8.2 Modification de App.vue

Nous gérons maintenant l'incrémentation ou la décrémentation de la quantité lors de l'ajout ou de la suppression d'un élément du panier.

Voici le composant :

```
<script setup lang="ts">
...
function ajouterAuPanier(productId: number): void {
  const product = state.products.find((product) => product.id === productId);
  if (product) {
    const productInCart = state.cart.find(
      (product) => product.id === productId
    );
    if (productInCart && productInCart.quantity) {
      productInCart.quantity++;
    } else {
      state.cart.push({ ...product, quantity: 1 });
    }
  }
}
```

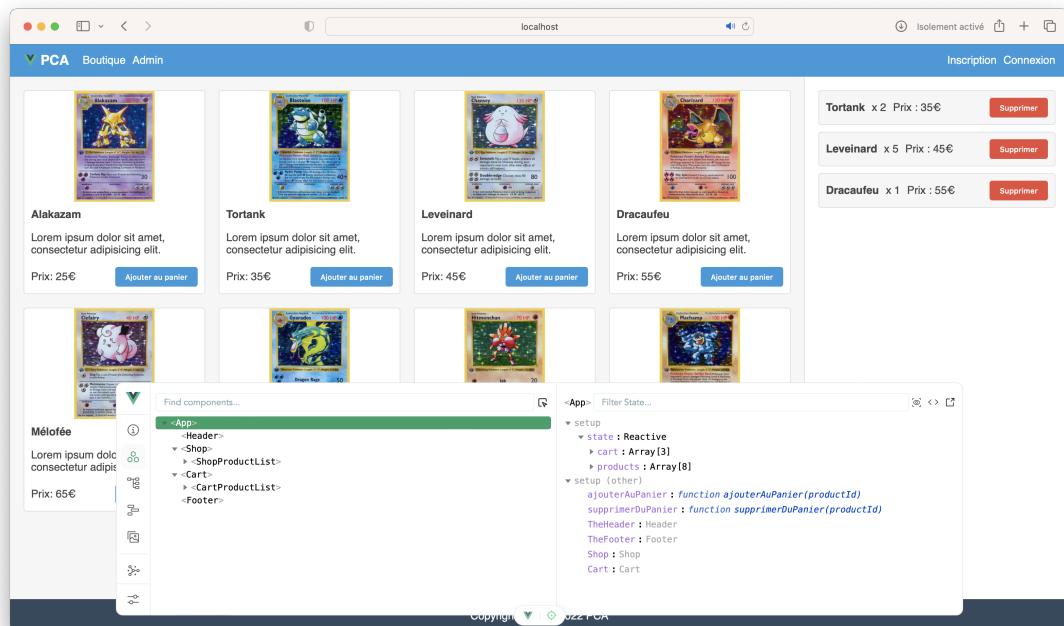
```

        }
    }

    function supprimerDuPanier(productId: number): void {
        const productFromCart = state.cart.find(
            (product) => product.id === productId
        );
        if (productFromCart?.quantity === 1) {
            state.cart = state.cart.filter((product) => product.id !== productId);
        } else if (productFromCart && productFromCart.quantity) {
            productFromCart.quantity--;
        }
    }
}

</script>

```



9 Affichage du total et affichage initial du panier

9.1 Crédation dossier pour les styles

Dans le dossier `assets`, créez un dossier `scss` et placez-y les fichiers `base.scss` et `debug.scss`. N'oubliez pas de modifier en conséquence les importations dans `App.vue` :

```

<style lang="scss">
@use './assets/scss/base.scss' as *;
@use './assets/scss/debug.scss' as *;

```

9.2 Ajout d'une classe

Dans le fichier `base.scss`, ajoutez la classe suivante :

```
.btn-success {
    background-color: var(--success-1);
    color: var(--text-primary-color);
    &:hover {
        background-color: var(--success-2);
    }
}
```

9.3 Modification de `Cart.vue`

Nous calculons le total du panier à l'aide de la fonction réactive suivante :

```
<script setup lang="ts">
...
const totalPrice = computed(() =>
    props.cart.reduce((acc, product) => acc + product.price * (product.quantity?product.quantity:0), 0)
);
</script>
```

Ajouter un bouton de commande affichant le total du panier :

```
<template>
    <div class="p-20 d-flex flex-column">
        ...
        <button class="btn btn-success">Commander {{ totalPrice }}</button>
    </div>
</template>
```

Pour afficher le bouton en bas de la page, ajouter un ressort `class="flex-fill"` à la balise `<CartProductList ...>`

9.4 Modification de `App.vue`

Nous utilisons une propriété exploitée pour calculer pour savoir si le panier est vide ou non. Si le panier est vide, nous n'affichons pas le composant panier grâce à la directive `v-if`. Nous utilisons également une liaison de classe avec la directive `v-bind` pour appliquer la classe `gridEmpty` si le panier est vide. Cette classe permet de modifier simplement la grille CSS.

```
<script setup lang="ts">
...
const cartEmpty = computed(() => state.cart.length === 0);
</script>
<template>
    <div
        class="app-container"
        :class="{
            gridEmpty: cartEmpty,
        }"
    >
        ...
    </div>
</template>
```

```

<Cart
    v-if="!cartEmpty"
    :cart="state.cart"
    class="cart"
    @remove-product-from-cart="supprimerDuPanier"
/>
...
</div>
</template>
<style lang="scss">
...
.gridEmpty {
    grid-template-areas: 'header' 'shop' 'footer';
    grid-template-columns: 100%;
}
...
</style>

```

