

TP3: Vue

Ibrahim ALAME

7 janvier 2026

1 Objectifs

Nous repartons du projet où nous l'avons laissé :

```
git clone https://github.com/ialame/vuejs-boutique-tp2-2.git
mv vuejs-boutique-tp2-2 vuejs-boutique-tp3
cd vuejs-boutique-tp3
npm i
```

Nous allons ajouter des filtres pour notre boutique : une barre de recherche, un filtre par prix et un filtre par catégorie.

1.1 Crédation de l'interface pour nos filtres

1. Dans le dossier `interfaces`, créez le fichier `Filters.interface.ts` :

```
export type Extension = 'jungle' | 'fossile' | 'expedition' | 'aquapolis' | 'all';

export interface FiltersInterface {
  search: string;
  priceRange: [number, number];
  extension: Extension;
}
```

2. Créez ensuite un fichier `index.ts` dans le dossier `interfaces` puis copiez dans ce fichier les `exports` :

```
export * from './Product.interface';
export * from './Filters.interface';
```

cela permettra de simplifier les importations de type. Il suffit d'indiquer dans chaque importation de type l'adresse '`@/interfaces`'. Exemple :

```
import type { Extension } from '@/interfaces'
```

3. Dans le dossier `data`, créez un fichier `filters.ts`. Dans ce fichier on définit une constante `DEFAULT_FILTERS` contenant les filtres par défaut : une chaîne de caractères vide, un intervalle de prix très large et toutes les extensions :

```
export const DEFAULT_FILTERS: FiltersInterface = {
  search: '',
  priceRange: [0, 10000],
  extension: 'all',
};
```

1.2 Modification de App.vue

Nous modifions le `state` réactif de notre composant racine, en ajoutant une propriété qui contient les filtres en cours. Nous avons des filtres sélectionnés par défaut contenus dans `DEFAULT_FILTERS`.

```
const state = reactive<{
    // ...
    filters: FiltersInterface;
}>({
    // ...
    filters: { ...DEFAULT_FILTERS},
});
```

1.3 Modification de l'interface Product

Modifiez le fichier `interfaces/ProductInterface.ts` pour ajouter une propriété `extension` :

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

2 Préparation pour le filtrage des produits

2.1 Modification de ShopProductList.vue

Nous ajoutons une `key` pour la directive `v-for` afin d'optimiser les performances de mise à jour des produits :

```
<template>
  <div class="grid p-20">
    <ShopProduct
      @add-product-to-cart="emit('addProductToCart', $event)"
      v-for="product of products"
      :product="product"
      :key="product.id"
    />
  </div>
</template>
```

2.2 Modification de Filters.interface.ts

Nous allons ajouter une interface `FilterUpdate` :

```
export interface FilterUpdate {
  search?: string;
  priceRange?: [number, number];
```

```

    extension?: Extension;
}

```

`FilterUpdate` contiendra le filtre appliqués par l'utilisateur à chaque fois qu'il modifie un filtre. Ainsi par exemple, s'il modifie l'intervalle de prix, l'objet contiendra `priceRange` uniquement. Même chose s'il effectue une recherche, nous aurons la propriété `search` uniquement. C'est pour cette raison que toutes les propriétés sont optionnelles avec `?`.

2.3 Modification de `App.vue`

Nous filtrons les produits à afficher en fonction de toutes les options des filtres que nous allons mettre en place en utilisant une propriété calculée. Nous écoutons l'événement que nous allons émettre lorsque l'utilisateur modifiera un filtre et nous modifions l'état des filtres en fonction de l'événement reçu.

```

<script setup lang="ts">
...
function updateFilter(filterUpdate: FilterUpdate) {
  if (filterUpdate.search !== undefined)
    state.filters.search = filterUpdate.search;
  else if (filterUpdate.priceRange)
    state.filters.priceRange = filterUpdate.priceRange;
  else if (filterUpdate.extension)
    state.filters.extension = filterUpdate.extension;
  else
    state.filters = { ...DEFAULT_FILTERS };
}

const filteredProducts = computed(() => {
  return state.products.filter((product) => {
    return product.title
      .toLocaleLowerCase()
      .startsWith(state.filters.search.toLocaleLowerCase()) &&
      product.price >= state.filters.priceRange[0] &&
      product.price <= state.filters.priceRange[1] &&
      (product.extension === state.filters.extension ||
       state.filters.extension === 'all')
  });
});
</script>

<template>
...
<Shop
  @update-filter="updateFilter"
  :products="filteredProducts"
  @add-product-to-cart="ajouterAuPanier"
  class="shop"
/>
...
</template>

```

Notez bien que nous passons maintenant la propriété calculée `filteredProducts` à notre composant `Shop` et non plus les produits du `state`.

Notez également que `{ ...DEFAULT_FILTERS }` permet de créer un nouvel objet différent lors de la réinitialisation des filtres pour ne pas avoir le même objet que dans le `state`.

3 Crédation du composant `ShopFilters`

3.1 Crédation du composant `ShopFilters`

Dans le dossier `src/shop` créez le fichier `ShopFilters.vue`. Ce sera le composant responsable des filtres de l'application.

3.2 Modification de `App.vue`

Nous passons en `prop` les filtres à notre composant `Shop` pour que celui-ci puisse les passer ensuite au composant responsable des filtres.

```
<Shop
      :filters="state.filters"
    />
```

3.3 Modification de `Shop.vue`

Nous modifions le composant `Shop.vue` qui va recevoir en `prop` les filtres depuis le composant `App.vue`. Il les repasse ensuite au composant `ShopFilters`. Il va également ré-émettre l'événement `updateFilter` qu'il va recevoir du composant `ShopFilters` pour mettre à jour le filtre modifié par l'utilisateur. Ici il ne fait donc que transmettre l'événement et les `props` dans les deux sens entre `App.vue` et `ShopFilters` :

1. Ajouter `filters` de type `FiltersInterface` à `defineProps`.
2. Ajouter à `defineEmits` l'événement `updateFilter` avec un paramètre `filterUpdate: FilterUpdate`.
3. Munir la `div` principale d'une flexibilité horizontale.
4. Insérer le composant `ShopFilters` avant `ShopProductList`

```
<ShopFilters
      :filters="filters"
      @update-filter="emit('updateFilter', $event)"
      class="shop-filter"
    />
```

5. Ajouter une classe "ressort" à `ShopProductList`.
6. Ajouter la classe suivante à la balise `style` :

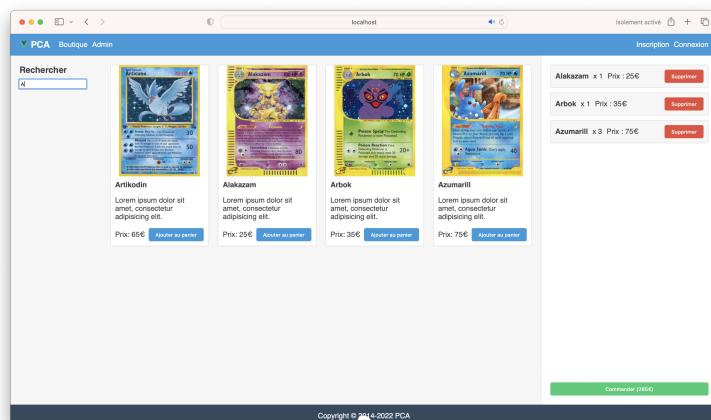
```
.shop-filter {
  flex: 0 0 200px;
}
```

3.4 Modification de ShopFilters.vue

Nous mettons en place le composant en créant le premier filtre pour la recherche par nom de produit. Lorsque l'utilisateur tape dans le champ de la recherche, il va émettre un événement `updateFilter` avec la valeur du champ :

1. Déclarer la propriété `filters` et l'événement `updateFilter`.
2. Ajouter une division `div` flexible verticalement avec `padding` de 20px.
3. Dans `div` ajouter une balise `section` avec une marge inférieur de 20px contenant les deux éléments :
 - Un titre `h3` avec une marge inférieur de 20px contenant le mot : Rechercher.
 - La zone de saisie `input` suivante :

```
<input  
  :value="filters.search"  
  @input="emit('updateFilter', { search: ($event.target as HTMLInputElement).value })"  
  type="text"  
  placeholder="Rechercher"  
/>
```



4 Filtrer les produits par prix

4.1 Modification de ShopFilters.vue

Nous ajoutons le filtre par intervalles de prix :

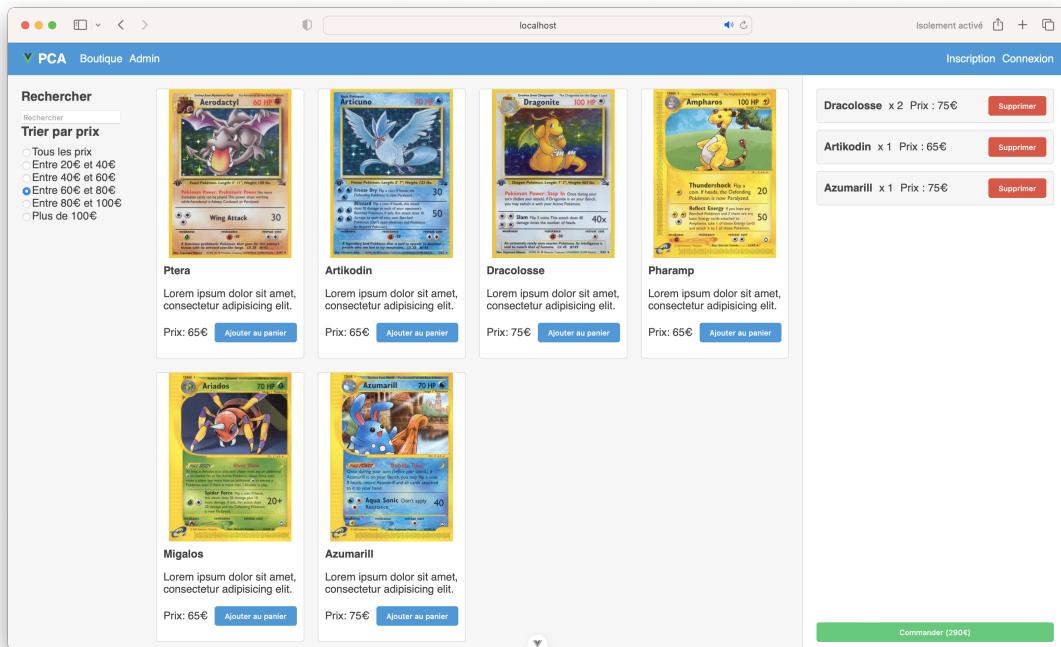
```
<section class="mb-20">  
  <h3 class="mb-10">Trier par prix</h3>  
  <div  
    class="mb-5"  
    v-for="priceRange of ([[0,1000],[20,40],[40,60],[60,80],[80,100],[100,1000]] as [number,number] [])">  
    <input  
      :checked="filters.priceRange[0] === priceRange[0]"  
      type="radio"  
      @input="emit('updateFilter', { priceRange })"
```

```

        name="priceRange"
        :id="priceRange[0].toString()"
    />
<label :for="priceRange[0].toString()">
{{{
    priceRange[0] === 0 ? 'Tous les prix':
    priceRange[0] === 100 ? 'Plus de 100€':
    `Entre ${priceRange[0]} et ${priceRange[1]}`
}}}
</label>
</div>
</section>

```

- `as [number, number]` : permet de préciser à TypeScript que nous avons bien un tableau contenant des tableaux avec exactement deux éléments de type nombre.
- `v-for="priceRange of` : permet d'itérer sur l'ensemble des intervalles de prix.
- `:checked="filters.priceRange[0] === priceRange[0]"` : nous cochons l'élément uniquement si la valeur minimale de l'intervalle contenue dans les filtres passés est la même que la valeur minimale de l'intervalle de l'itération en cours.
- `@input="emit('updateFilter', { priceRange })"` : permet d'émettre l'événement de mise à jour des filtres avec en valeur un objet contenant une propriété qui a en clé le filtre à modifier et en valeur l'intervalle sélectionné.
- `:id="priceRange[0].toString()"` : nous nous servions de la valeur Inférieure de l'intervalle comme identifiant HTML unique.



5 Filtrer les produits par extension

5.1 Modification de ShopFilters.vue

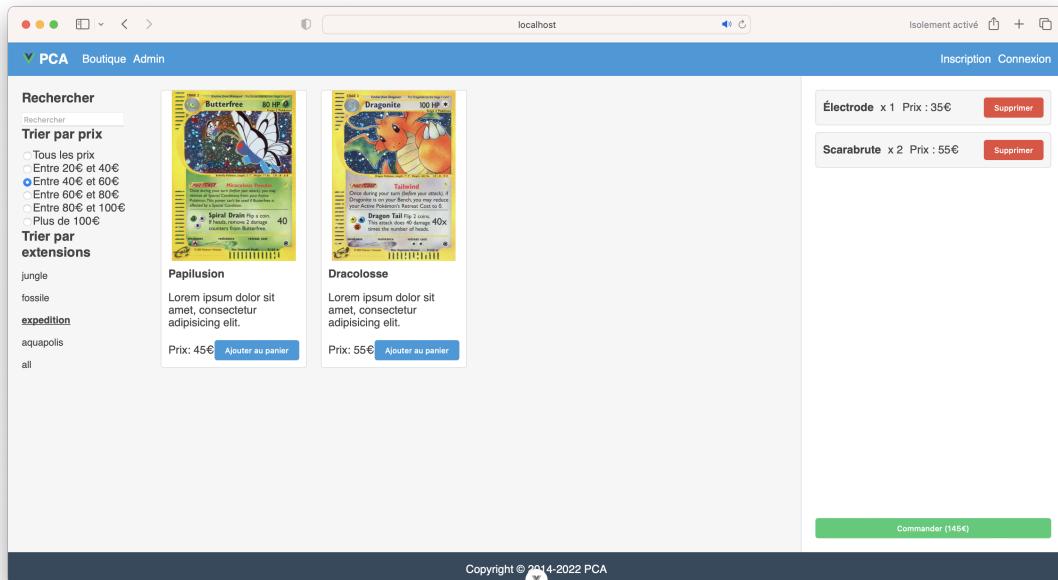
Nous ajoutons le filtre par extension :

```
<section class="mb-20 flex-fill">
  <h3 class="mb-10">Trier par extensions</h3>
  <p
    class="extension"
    :class="{ selected: filters.extension === extension }"
    v-for="extension in(['jungle', 'fossile', 'expedition', 'aquapolis', 'all'] as Extension[])"
    @click="emit('updateFilter', { extension: extension })"
  >
    {{ extension }}
  </p>
</section>
```

où `extension` est la classe de style suivante :

```
<style lang="scss" scoped>
  .extension {
    font-size: 14px;
    line-height: 18px;
    cursor: pointer;
    &:hover {
      text-decoration: underline;
    }
  }
  .selected {
    font-weight: bold;
    text-decoration: underline;
  }
</style>
```

`:class=" selected : filters.extension === extension "` : permet d'ajouter la classe `selected` si l'extension sélectionnée dans la `props filters` est celle de l'itération en cours.



6 Reset des filtres

6.1 Modification de `ShopFilters.vue`

Nous ajoutons la reset des filtres et l'affichage du nombre de résultat :

1. Ajouter à `Props` la propriété `nbrOfProducts` en tant que `number`.
2. Ajouter à la division principale, après la dernière section, les éléments suivants :

```
<small class="mb-5">
    Nombre de résultats:
    <strong>{{ nbrOfProducts }}</strong>
</small>
<button class="btn btn-danger" @click="emit('updateFilter', {})">
    Supprimer les filtres
</button>
```

6.2 Modification de `Shop.vue`

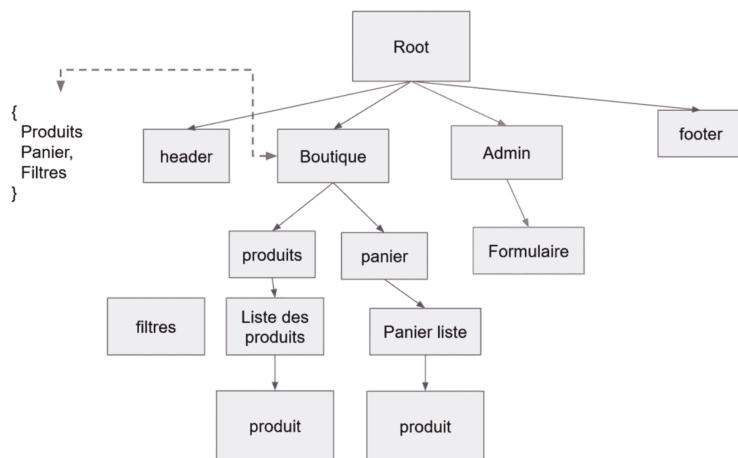
Il ne faut pas oublier de modifier le composant `Shop` pour passer en `prop` le nombre de résultats :

```
<ShopFilters
    :nbr-of-products="products.length"
/>
```

7 Introduction des fonctionnalités

7.1 Architecture des composants

Voici l'arbre des composants que nous aurons à la fin du TP :



Notez que l'état des produits, du panier et des filtres sera contenu dans le composant **Boutique**.

7.2 Crédation de nouveaux dossiers

1. Dans le dossier `src` créez un dossier `features`.

2. Dans ce dossier, créez les dossiers **boutique** et **admin**.
3. Dans le dossier **boutique** créez un dossier **components**. Déplacez-y les dossiers **Cart** et **Shop**.
4. Créez également dans **boutique** un dossier **data** et un fichier **Boutique.vue**.

7.3 Modification de App.vue

1. La plupart de la logique dans le composant racine est déplacée dans le composant **Boutique**. Copier alors le contenu du fichier **App.vue** dans **Boutique.vue**.
2. Dans **App.vue**, vider les trois parties : **script**, **template** et **style** :

```
<script setup lang="ts">

</script>

<template>

</template>

<style lang="scss">

</style>
```

3. Importer les données à partir du fichier **data/product.ts** :

```
import data from '@/data/product';
```

4. Définir la constante réactive **state** contenant **products** de type **ProductInterface[]** égal initialement à **data** :

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

5. Dans une division **div** de classe **app-container** on ajoute les trois éléments :

- Le composant **TheHeader** de classe **header**.
- Un composant dynamique **Component** initialement identifié à **Boutique** :

```
<Component :is="Boutique" :products="state.products"/>
```

- Le composant **TheFooter** de classe **footer**.

6. Importer le fichier style **base.scss** :

```
@import '@/assets/scss/base.scss';
```

7. Définir la classe **app-container** de la façon suivante :

```
.app-container {
    min-height: 100vh;
    display: grid;
    grid-template-areas: 'header' 'app-content' 'footer';
    grid-template-rows: 48px auto 48px;
}
```

8. Ajouter les trois classes **header**, **app-content** et **footer** :

```
.header { grid-area: header; }
.app-content { grid-area: app-content; }
.footer { grid-area: footer; }
```

Notez bien l'utilisation du composant dynamique qui nous permettra de changer le composant affiché.

7.4 Modification de Boutique.vue

Après avoir déplacé toute la logique de la racine :

1. Supprimer les deux composants `TheHeader`, `TheFooter` et la variable `data`.
2. Déplacer `products` de la fonction `reactive` vers `DefineProps` :

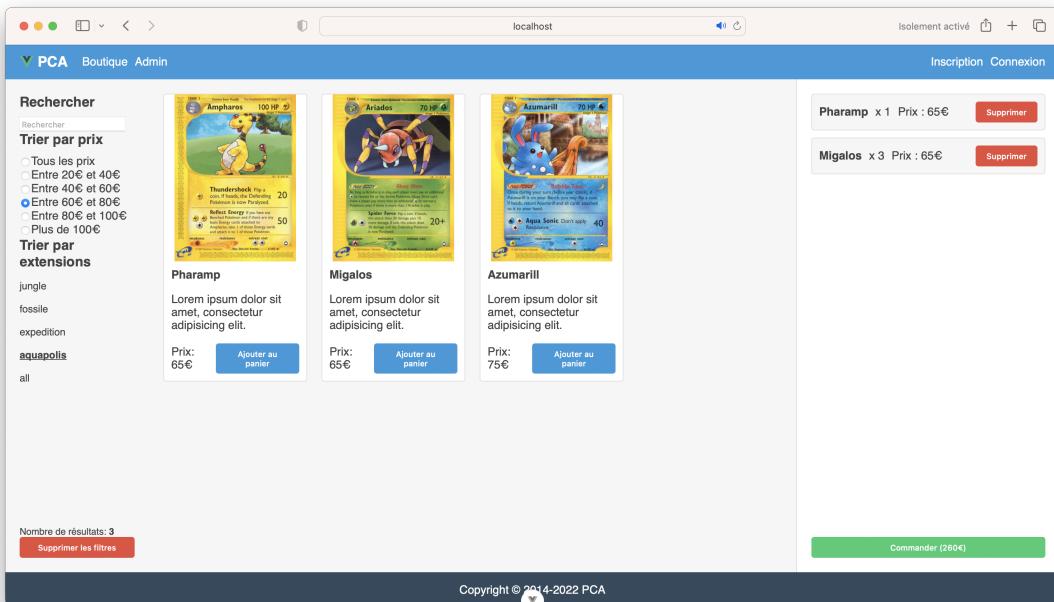
```
const props = defineProps<{
  products: ProductInterface[];
}>();
```

3. Rectifier `state.products` en passant à `props.products` dans l'expression de la valeur calculée `filteredProducts` et dans la fonction `ajouterAuPanier`.
4. Remplacer la classe `app-container` par :

```
.boutique-container {
  display: grid;
  grid-template-columns: 75% 25%;
}
```

5. Modifier la classe de la `div` principale en `boutique-container`.
6. Supprimer la propriété `grid-template-areas` de la classe `grid-empty`.
7. Supprimer la propriété `grid-area` de la classe `cart` et supprimer les autres classes `header`, `footer` et `shop`.

A ce niveau, l'application s'exécute de la même façon que dans la section précédente :



8 Navigation à l'aide d'un composant dynamique

8.1 Crédation du composant Admin

Dans le dossier `src/features/admin` créer le composant `Admin.vue`.

8.2 Crédation du fichier type.ts

Dans le dossier `interfaces` créé le fichier `type.ts` qui va contenir nos types suivants :

```
export type Page = 'Boutique' | 'Admin';
export type Extension = 'jungle' | 'fossile' | 'expedition' | 'aquapolis' | 'all';
```

N'oubliez pas d'ajouter au fichier `index.ts` dans le même dossier la ligne :

```
export * from './type';
```

Ainsi que l'import dans `Filters.interface.ts` et `Product.interface.ts` :

```
import type { Extension } from '@/interfaces'
```

8.3 Modification de App.vue

Dans le composant `App.vue` nous allons mettre en place les deux composants possibles dans le composant dynamique et la navigation entre les deux :

1. Importer le type `type Component` et donner lui un alias `C` :

```
import { reactive, type Component as C } from 'vue';
```

2. Ajouter à **state** la variable **page** de type **Page** initialement égal à 'Boutique'.
3. Soit **pages** l'ensemble de deux composants ; **Boutique** et **Admin** :

```
const pages: { [s: string]: C } = {
  Boutique,
  Admin
}
```

et soit **navigate** la fonction qui permet d'assigner à **page** de **state** l'argument **page** :

```
function navigate(page: Page): void {
  state.page = page;
}
```

4. Rendre la propriété **is** de **Component** sensible à la variation de la variable reactive **page** :

```
<Component :is="pages[state.page]" :products="state.products"/>
```

5. Pour faire passer les interactions entre la racine **App** et le composant **TheHeader** dans les deux sens on ajoute à **TheHeader** la propriété et l'événement suivants :

```
<TheHeader class="header" :page="state.page" @navigate="navigate"/>
```

8.4 Modification de Header.vue

Nous utilisons l'événement et la **props** définition dans le composant racine afin de pouvoir naviguer entre les composants **Admin** et **Boutique** :

```
<script setup lang="ts">
import type { Page } from '@/interfaces'
defineProps<{
  page: Page;
}>();

const emit = defineEmits<{
  (e: 'navigate', page: Page): void ;
}>();
</script>

<template>
...
<ul class="d-flex flex-row flex-fill">
  <li class="mr-10">
    <a
      :class="{active:page === 'Boutique'}"
      @click="emit('navigate','Boutique')"
      >Boutique </a>
  </li>
  <li>
    <a
      :class="{active:page === 'Admin'}"
      @click="emit('navigate','Admin')"
      >Admin</a>
  </li>
</ul>

```

```

...
</template>

<style lang="scss" scoped>
header {
  // ...
  a.active {
    text-decoration: underline;
  }
}
</style>

```

8.5 Modification de src/assets/scss/base.scss

Ajoutez une classe pour nos cartes :

```

.card {
  border: var(--border);
  border-radius: var(--border-radius);
  background-color:white;
  padding: 30px;
}

```

8.6 Crédation du composant ProductForm

Dans le dossier `features/admin` créez un dossier `components` dans lequel vous mettez le fichier `ProductForm.vue` :

```

<script setup lang="ts"></script>

<template>
  <div class="card">
    <h1>Formulaire</h1>
  </div>
</template>

<style scoped lang="scss">
.card {
  width: 100%;
  max-width: 500px;
}
</style>

```

8.7 Modification du composant Admin

Nous utilisons ce composant dans le composant `Admin.vue` :

```

<script setup lang="ts">
import ProductForm from './components/ProductForm.vue';
</script>

<template>

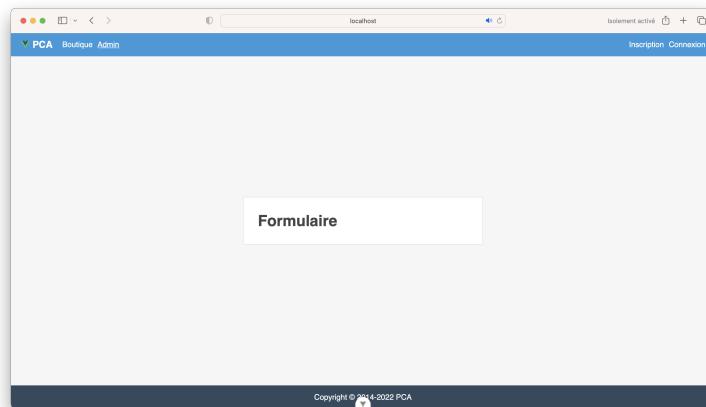
```

```

<div class="admin-container d-flex flex-row justify-content-center align-items-center">
  <ProductForm />
</div>
</template>

<style scoped lang="scss">
.admin-container {
  height: 100%;
}
</style>

```



9 Mise en place du formulaire

9.1 Installation

Commencer par installer les librairies `validate` et `zod` permettant de valider un formulaire. Taper dans le terminal de `WebStorm` :

```
npm i vee-validate zod @vee-validate/zod
```

9.2 Modification de `base.scss`

Ajoutez les classes suivantes pour le formulaire :

```

input, textarea, select {
  border: var(--border);
  border-radius: var(--border-radius);
  padding: 8px 15px;
}

label {
  font-size: 14px;
  font-weight: 500;
  color: var(--gray-3);
}

```

```

.form-error {
  color: var(--danger-1);
  font-size: 14px;
  font-weight: 500;
}

```

9.3 Modification de ProductForm.vue

Nous mettons en place le formulaire et la validation :

```

<script setup lang="ts">
import { useForm, useField } from 'vee-validate';
import { z } from 'zod';
import { toTypedSchema } from '@vee-validate/zod';
import type { ProductInterface } from '@/interfaces'

const required = { required_error: 'Veuillez renseigner ce champ' };
const validationSchema = toTypedSchema(
  z.object({
    title: z
      .string(required)
      .min(1, { message: 'Le titre doit faire au moins 1 caractère' })
      .max(20, { message: 'Le titre doit faire moins de 20 caractères' }),
    image: z.string(required),
    price: z
      .number(required)
      .min(0, { message: 'Le prix doit être supérieur à 0€' })
      .max(15000, { message: 'Le prix doit être inférieur à 150 00€' }),
    description: z
      .string(required)
      .min(10, { message: 'La description doit faire au moins 10 caractères' }),
    extension: z.string(required),
  })
);

const { handleSubmit, isSubmitting } = useForm({
  validationSchema,
});

const title = useField('title');
const image = useField('image');
const price = useField('price');
const description = useField('description');
const extension = useField('extension');

const trySubmit = handleSubmit((formValues) => {
  console.log(formValues);
});
</script>

<template>

```

```

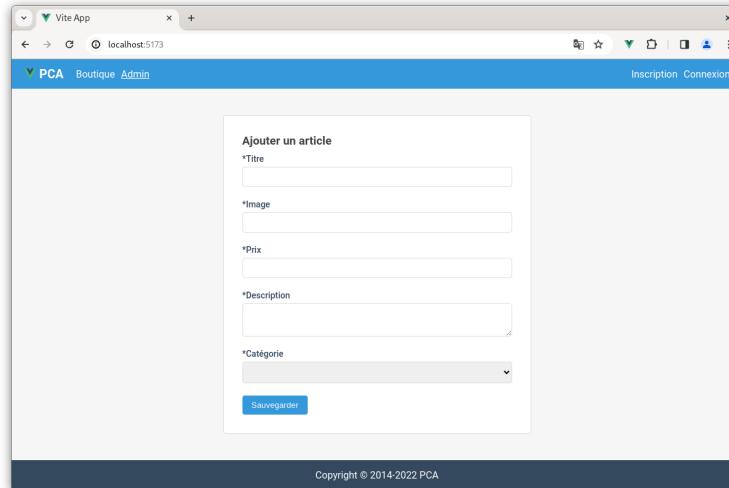
<div class="card">
  <h3 class="mb-10">Ajouter un article</h3>
  <form @submit="trySubmit">
    <div class="d-flex flex-column mb-20">
      <label class="mb-5">*Titre</label>
      <input ref="firstInput" v-model="title.value.value" type="text" />
      <small class="form-error" v-if="title.errorMessage.value">{{ title.errorMessage.value }}</small>
    </div>
    <div class="d-flex flex-column mb-20">
      <label class="mb-5">*Image</label>
      <input v-model="image.value.value" type="text" />
      <small class="form-error" v-if="image.errorMessage.value">{{ image.errorMessage.value }}</small>
    </div>
    <div class="d-flex flex-column mb-20">
      <label class="mb-5">*Prix</label>
      <input v-model="price.value.value" type="number" />
      <small class="form-error" v-if="price.errorMessage.value">{{ price.errorMessage.value }}</small>
    </div>
    <div class="d-flex flex-column mb-20">
      <label class="mb-5">*Description</label>
      <textarea v-model="(description.value.value as string)"></textarea>
      <small class="form-error" v-if="description.errorMessage.value">{{ description.errorMessage.value }}</small>
    </div>
    <div class="d-flex flex-column mb-20">
      <label class="mb-5">*Extension</label>
      <select v-model="extension.value.value">
        <option value="disabled">Choisissez une extension</option>
        <option value="jungle">Jungle</option>
        <option value="fossile">Fossile</option>
        <option value="expedition">Expedition</option>
        <option value="aquapolis">Aquapolis</option>
      </select>
      <small class="form-error" v-if="extension.errorMessage.value">{{ extension.errorMessage.value }}</small>
    </div>
    <button class="btn btn-primary" :disabled="isSubmitting">
      Sauvegarder
    </button>
  </form>
</div>
</template>

<style scoped lang="scss">

```

```
.card {
  width: 100%;
  max-width: 500px;
}
</style>
```

`image.value.value`: permet d'accéder à la propriété `value` renvoyée par `useField`, comme elle contient une `ref` qui est imbriquée dans l'objet, il n'y a pas d'accès automatique par `Vue` à la propriété `value` de la `ref` côté `template`(`unpacking`). Il faut donc y accéder par nous-mêmes.

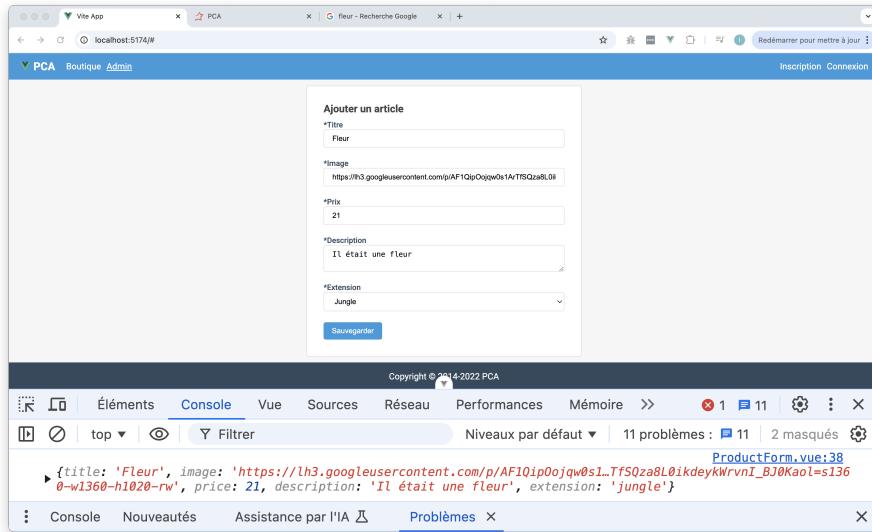


10 Envoi du formulaire

10.1 Modification de `ProductForm.vue`

Nous mettons en place l'envoi du formulaire dans la base des données (ici le fichier `product.ts`). La fonction `handleSubmit` permet de récupérer les données du formulaire et de les afficher dans la console :

```
const trySubmit = handleSubmit((formValues) => {
  console.log(formValues);
});
```



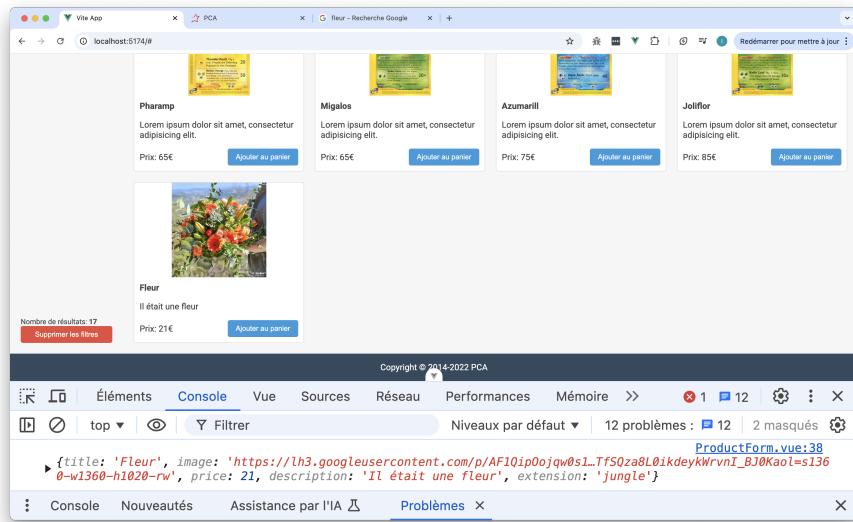
Pour ajouter la saisie dans le tableau `products` modifier la fonction `handleSubmit` dans `ProductForm.vue` :

```
const trySubmit = handleSubmit((formValues) => {
  console.log(formValues);
  props.products.push({ ...formValues, id:110 });
});
```

Il faut bien sûr acheminer `products` en tant que propriété `prop` entre la racine et `ProductForm.vue`. De plus l'identifiant `id` doit être incrémenté pour chaque saisie. (ici pour l'exemple `id:110`). Pour tester le formulaire on ajoute le produit suivant :

```
title : Fleur
image : https://lh3.googleusercontent.com/p/AF1QipOojqwOs1ArTfSQza8L0ikdeykWrvnI_BJ0Kaol=s1360-w1360-h1020-rw
Prix : 21
Description : Il était une fleur
Extension : Jungle
```

Après validation on bascule vers la boutique. Le produit fleur est affiché en dernier :



Travail optionnel (hors barème) :

1. Ecrire un code permettant de : Ajouter/Supprimer/Modifier un produit dans le fichier `product.ts`.
2. On veut qu'après validation du formulaire par le bouton **Sauvegarder** la page **Boutique** s'affiche.

11 Amélioration du CSS

11.1 Modification de `Shop.vue`

Nous modifions le composant `Shop.vue` pour rendre **scrollable** la liste des produits :

```
<template>
  <div class="d-flex flex-row">
    <!-- ... -->
    <ShopProductList
      class="flex-fill scrollable"
      @add-product-to-cart="emit('addProductToCart', $event)"
      :products="products"
    />
  </div>
</template>

<style lang="scss" scoped>
.scrollable {
  overflow-y: auto;
  height: calc(100vh - 96px);
}

// ...
</style>
```

11.2 Cration du fichier _mixins.scss

Dans le dossier `src/assets/scss` crezez le fichier `_mixins.scss` :

```
@mixin sm {
  @media (min-width: 576px) {
    @content;
  }
}

@mixin md {
  @media (min-width: 768px) {
    @content;
  }
}

@mixin lg {
  @media (min-width: 992px) {
    @content;
  }
}

@mixin xl {
  @media (min-width: 1200px) {
    @content;
  }
}
```

11.3 Modification de `ShopProductList.vue`

Nous utilisons nos `mixins` pour rendre la liste des produits `responsive` (c'est-a-dire qui s'adapte a la taille de l'cran) :

```
use '@/assets/scss/mixins' as m;
.grid {
  display: grid;
  grid-template-columns: 1fr;
  @include m.md {
    grid-template-columns: 1fr 1fr;
  }
  @include m.lg {
    grid-template-columns: 1fr 1fr 1fr;
  }
  @include m.xl {
    grid-template-columns: 1fr 1fr 1fr 1fr;
  }
  grid-auto-rows: 400px;
  gap: 20px;
}
```

Vite App

localhost:5167

Débuter avec Firefox Des exercices pour l... Ibrahim Alame VOUS AVEZ DIT PA... Historia-0048 statistique - Yahoo ... Autres marque-pages

PCA Boutique Admin

Inscription Connexion

Rechercher

Rechercher

Trier par prix

- Tous les prix
- Entre 20€ et 40€
- Entre 40€ et 60€
- Entre 60€ et 80€
- Entre 80€ et 100€
- Plus de 100€

Trier par extensions

jungle
fossile
expedition
aquapolis
all

Nombre de résultats: 16

Supprimer les filtres

55€ panier 65€ panier 65€ panier


Dragonite 100 HP
Pokemon Power: Step 1. Once during your turn, you may search up to 2 cards from your deck and add them to your hand. Then, you may search up to 2 cards from your deck and add them to your hand. Then, you may search up to 2 cards from your deck and add them to your hand.


Lapras 80 HP
Water/Fairy


Alakazam 100 HP
Psychic/Poison


Dracolosse 75€
Dracolosse


Lokhlass 85€
Lokhlass


Artikodin 25€
Artikodin


Arbok 70 HP
Poison Spray


Butterfree 80 HP
Butterfree


Dragonite 100 HP
Tailwind

Prix : Ajouter au panier

Prix : Ajouter au panier

Prix : Ajouter au panier

Pyroli x Prix : 1 45€ Supprimer

Dracolosse x Prix : 4 75€ Supprimer

Artikodin x Prix : 1 65€ Supprimer

Commander (410€)

Copyright © 2014-2022 PCA