Réalisation d'une application web VueJS et Vuetify

1 Introduction

On va utiliser ici les différentes notions abordées dans le sujet précédent. Le but ici est de développer une application web permettant de naviguer dans une liste de Pokémons. Un fichier json permet de stoker des informations, un peu comme une base de données, mais sans avoir à mettre en place un SGBD, tout en perdant les optimisations de celui-ci. Pour des quantités d'informations relativement faibles, ce format est tout à fait adapté. Pour ce TP, vous trouverez un fichier json sur Moodle, contenant une liste de types et une liste de Pokémons, dont voici la structure :

```
{
  types:[{nom:string,couleur:string}, ...],
  pokemons:[
  {
    id:int,
    nom:string,
    description:string,
    types:[{nom:string,couleur:string}, ...],
    image:string,
    base:{
      HP:int,
      Attack:int,
      Defense:int,
      SpAttack:int,
      SpDefense:int,
      Speed:int
    }
  }
  ...
  ]
}
```

Notre fichier contient un objet, celui-ci contenant un tableau de types, et un tableau de pokémons. Chaque type est un objet avec un nom et une couleur, et un pokémon est un objet possédant un numéro, un nom, une description, un tableau de types, un nom d'image et des statistiques. Pour voir le contenu du fichier, vous pouvez ouvrir directement celui-ci via Firefox ou Chrome, la navigateur se chargera de présenter les données sous un forme lisible pour un être humain.

Commençons par préparer votre répertoire de travail, afin d'avoir une base commune. Celui-ci devra présenter l'arborescence suivante :

- un fichier "serveur.js" contiendra votre code côté serveur ;
- le fichier "pokedex.json" sera placé à côté du serveur ;
- un répertoire "public" contiendra les ressources accessibles au client :
  - un fichier "index.html" pour la page principale ;
  - un dossier "images" contenant toutes les images des pokémons ;
  - un dossier "js" contenant vos futurs fichiers de composants.

2 Mise en place du serveur

Commençons par mettre en place un serveur NodeJS qui servira les fichiers du client tout en proposant une API gérant les données du fichiers JSON. Pour cela, créez un serveur express classique servant les fichiers en statique, auquel on ajoutera les fonctionnalités suivantes : le chargement de la base de données, et la mise en place de l'API pour notre application.

Charger la base de données se fait très simplement, on lit le fichier de manière synchrone, puis on parse le résultat pour obtenir l'objet JSON correspondant :

```
1 let database = JSON.parse(fs.readFileSync('pokedex.json'));
```

Après cette instruction, la variable `database` contient l'intégralité du contenu du fichier, sous la forme d'un objet js. Vous pouvez alors accéder à ses différents champs de la manière suivantes : `database.types`, `database.pokemons`, `database.pokemons[0].nom` ou encore `database.types[3].couleur`.

En ce qui concerne la mise en place de l'API se fera via le déploiement de deux méthodes `app.get`. La première permettra à l'utilisateur de récupérer la liste des types :

```
1   app.get('/types', function (req, res) {
2     res.send(database.types);
3   });
```

A chaque fois qu'un utilisateur fera appel à ce service, le serveur renverra la liste des types (i.e. un tableau). Le second service, que l'on appellera "/pokemons", devra permettre à l'utilisateur de récupérer la liste des pokémons répondant à un type, ou tous les pokémons si le type passé en paramètres est 'all'. Pour tester vos services, vous pouvez tester les requêtes "localhost :8080/types" et "localhost :8080/pokemons/all" directement dans votre navigateur : vous devriez obtenir les json correspondants.

3 Récupération des informations côté client

On développera l'interface dans le fichier "index.html", entre les balises `v-app`. On ajoutera par la suite différents composants imbriqués permettant de donner la structure voulue à la page. En parallèle, il vous faudra développer le modèle avec les données (liste de pokémons, liste de types, variables pour l'interface, ...) mais également les méthodes

permettant d'appeler l'API de notre serveur. De plus, il faudra que ces méthodes soient appelées lors de la création de la page. Pour cela, vous pourrez procéder de la sorte :

```
new Vue({
  el: '#app',
  vuetify: new Vuetify(),
  data:{
    pokemons:[],
    ...
  },
  created:function(){
    this.getTypes();
    this.getPokemons();
  },
  ...
```

Ainsi, les méthodes `getTypes` et `getPokemons` seront appelées dès la création de notre instance de Vue. Il faudra bien entendu créer ces méthodes, chacune d'entre elles faisant appel à axios pour effectuer une requête GET. Pour vérifier que la récupération se fait correctement, vous pouvez afficher la liste des pokémons récupérés, en essayant d'obtenir l'affichage suivant :

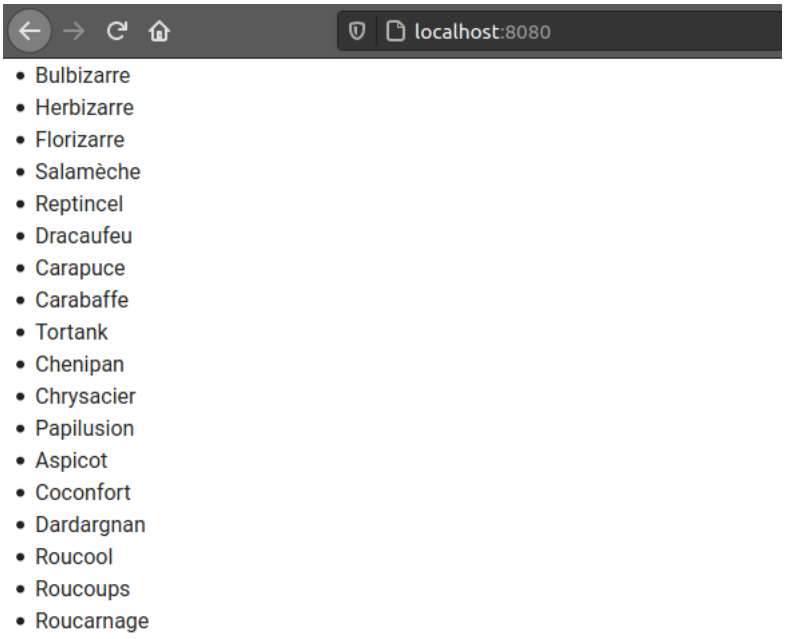

FIGURE 1 – Un premier affichage

Spoiler : vous pouvez facilement obtenir cet affichage via les balises ul et li, ainsi qu'avec un v-for bien placé.

4 Mise en place des filtres

On voudrait maintenant actualiser le contenu de la liste en fonction du type choisi par l'utilisateur. Pour cela, on doit :

- afficher une liste de sélection en haut de la page (balise select), contenant la liste des types sélectionnables ainsi que la valeur par défaut "Tous" ;
- faire en sorte que lorsque l'utilisateur choisit un type dans la liste :
  1. une requête est envoyée au serveur pour récupérer les pokémons correspondants ;
  2. la liste des pokémons est mise à jour en fonction de la réponse à la requête.

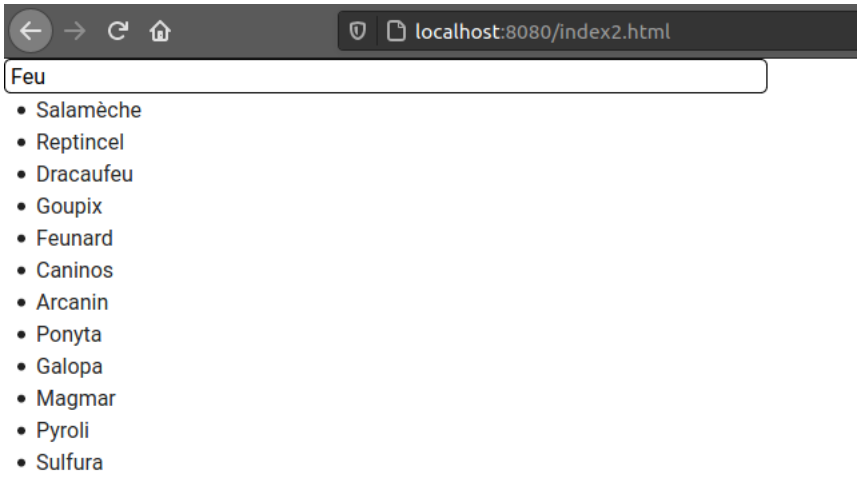

FIGURE 2 – Filtrer via le type

5 Mise en place de l'interface graphique

Une fois que votre moteur est mis en place et fonctionne correctement, vous pouvez attaquer la conception de l'interface avec Vuetify. Pour commencer, on va définir la zone d'affichage des pokémons de la manière suivante :

```
1 <v-container fluid class="ma-0 pa-0 d-flex flex-wrap
  ↪ justify-center">
2   <!--Affichage de tous les pokémons -->
3 </v-container>
```

Il s'agit d'un conteneur prenant toute la place disponible en largeur (via la propriété fluid), et possédant les propriétés css suivantes :

- ma-0 et pa-0 annulent les marges intérieures et extérieures ;
- d-flex transforme le composant en un conteneur flex, qui placera les éléments par défaut en ligne ;
- flex-wrap permet d'autoriser les éléments à passer à la ligne suivante si la place vient à manquer ;
- justify-center permet d'aligner les éléments de manière centrée sur l'axe primaire (ici horizontalement, vu que l'on place les éléments en ligne).

Par la suite, vous devrez imbriquer ce conteneur dans un autre, ce dernier affichant les éléments en colonne (barre de contrôles, conteneur de pokémon et pagination).

5.1 Composant pokemon

Pour l'instant, on n'affiche que le nom de chaque pokémon. Cependant, l'objet représentant un pokémon contient bien plus d'informations :

- le numéro, entre 1 et 151 ;
- le nom (français) ;
- un tableau de types, chaque case contenant un objet avec un nom et une couleur (sous la forme d'une chaîne compatible avec Vuetify) ;
- un objet contenant les statistiques de base : attaque, défense, hp, vitesse, attaque spéciale et défense spéciale ;
- une description (en anglais) ;
- une chaîne représentant le nom du fichier image du pokémon sur le serveur.

On va créer un composant qui prendra en paramètre un pokémon, et permettra d'afficher toutes ces informations. Pour cela, créez un fichier "pokemon.js", à placer dans le répertoire "public/js". Vous coderez à l'intérieur un composant permettant d'obtenir un affichage proche de celui-ci :

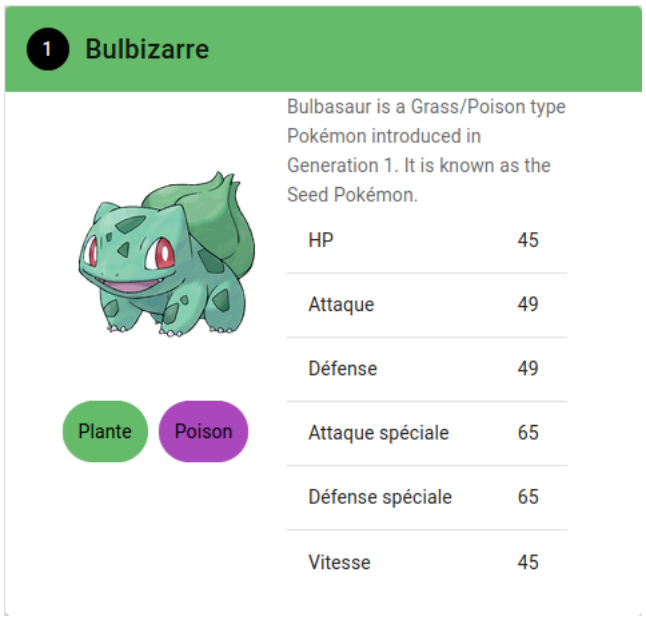

FIGURE 3 – Composant pokemon

Attention, il ne vous est pas demandé d'obtenir le même affichage au pixel près, vous devez juste vous en approcher, tout en ayant une certaine liberté sur les alignements, la taille des différents éléments, ... afin d'obtenir quelque chose qui vous plaira.

Dans un premier temps, on va identifier les différents graphique apparaissant dans notre composant :

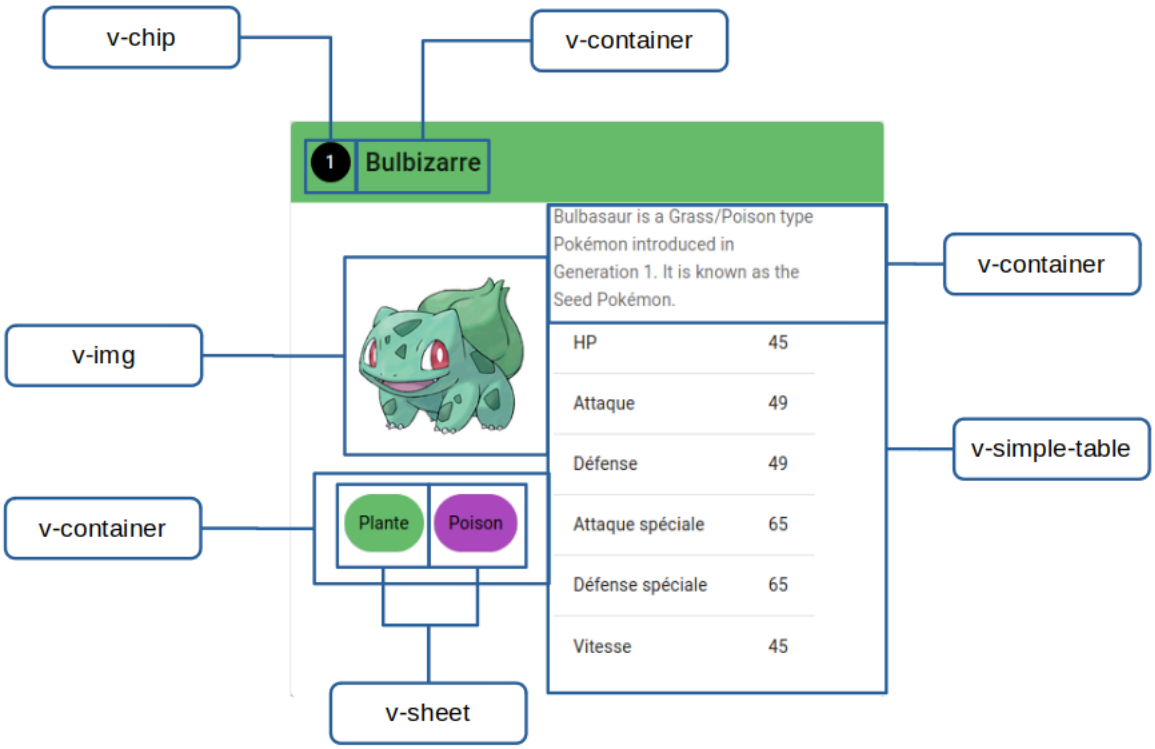

FIGURE 4 – Éléments graphiques

Il existe bien sûr des alternatives, mais on partira du principe que vous utilisez ces composants pour la suite. Pour plus d'informations sur l'utilisation de chacun de ces composants, référez vous à la documentation.

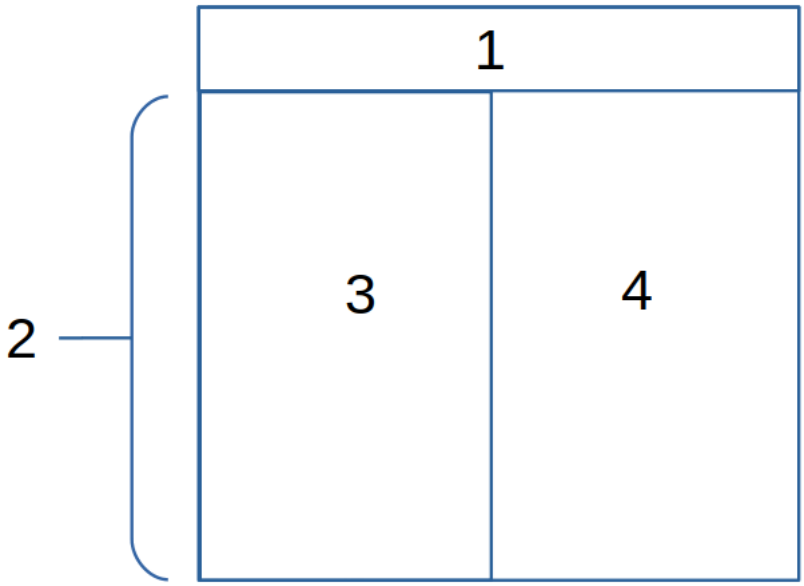

FIGURE 5 – Dispositions des conteneurs

En ce qui concerne le composant principal, on choisira une `v-card`, à l'intérieur de laquelle on définira un `v-card-title` et un `v-card-text`, correspondant respectivement aux zones 1 et 2 sur le schéma. Les zones 3 et 4 correspondent quant à elles à des `v-container`. On définira les règles suivantes :

- la zone 1 est un conteneur flex plaçant les éléments en ligne ;
- la zone 2 est un conteneur flex plaçant les éléments en ligne ;
- les zones 3 et 4 prennent chacun la moitié de l'espace via la classe col-6, et sont des conteneurs flex plaçant les éléments en colonne.

A vous de trouver les alignements qui vous conviennent le mieux (voir ici). Vous devrez également jouer sur les marges et padding des différents éléments. Enfin, le composant `v-card` doit voir sa largeur définie en fonction de la largeur de la fenêtre. Par exemple, sur un petit écran, on souhaiterait avoir une carte par ligne, alors que sur un grand écran, on pourrait vouloir en avoir 3 par ligne.

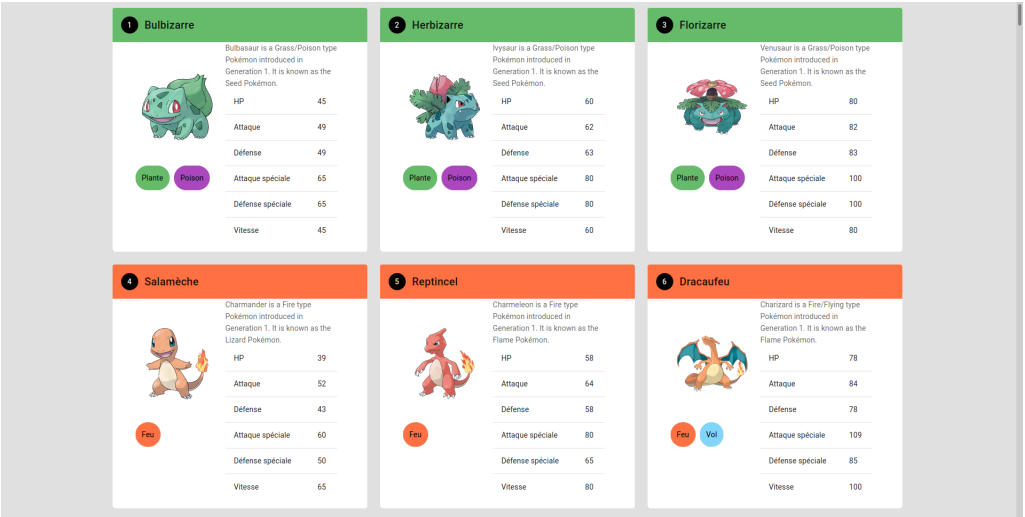

FIGURE 6 – Agencement sous forme de grille

5.2 La barre de contrôles

FIGURE 7 – Barre de contrôles

On va maintenant ajouter une barre de contrôles qui permettra à l'utilisateur de :

- sélectionner le type des pokémons à afficher, via un composant `v-select` ;
- sélectionner le mode d'affichage, entre clair et sombre, via un composant `v-switch`.

Pour le premier contrôle, vous avez juste à reprendre votre code utilisant les balises `select` et `option`, et l'adapter pour fonctionner directement avec le composant `v-select`.

Le deuxième contrôle va demander plus de travail : je vous conseille d'ajouter à votre modèle une donnée booléenne `darkMode`, initialisée à `false`. Cette variable `darkMode`

sera réglée par le `v-switch`, qui permettra de faire basculer sa valeur. Ensuite, la plupart des composants graphiques tels que `v-card`, `v-select`, ... possède un attribut `dark` pouvant être vrai ou faux. Ainsi, il suffira, pour quasiment tous les composants graphiques, d'ajouter à l'intérieur de la balise `:dark="darkMode"`. Enfin, si vous souhaitez modifier les classes css d'un éléments en fonction de la valeur de `darkMode`, vous pouvez procéder comme il suit (la balise v-container est définie sur plusieurs ligne pour des raisons de clarté, mais ce n'est pas obligatoire) :

```
<v-container :class="
'ma-0 pa-0 grey '+((!darkMode)?'lighten-2':'darken-2')
">
  <!--Contenu-->
</v-container>
```

Première remarque, l'attribut `class` prend pour une valeur une chaîne de caractères, composée des différentes classes css à appliquer, séparées par des espaces. Ici, on considère les classes css suivantes :

- ma-0 : pas de marges, à appliquer dans tous les cas ;
- pa-0 : pas de padding, à appliquer dans tous les cas ;
- grey : la couleur de fond est le gris, à appliquer dans tous les cas ;
- lighten-2 : la couleur de fond est claire, uniquement lorsque l'on utilise le mode clair ;
- darken-2 : la couleur de fond est foncée, uniquement lorsque l'on utilise le mode foncé.

En écrivant `:class="..."`, l'interpréteur s'attend à ce que l'intérieur des guillemets doubles soit une expression JavaScript, cette expression devra également générer une chaîne, d'où la présence de guillemets simples à l'intérieur. La chaîne est générée de la manière suivante :

- `'ma-0 pa-0 grey '` est le préfixe de la chaîne : on veut appliquer ces classes css dans tous les cas ;
- le suffixe de la chaîne est généré en fonction de la valeur de `darkMode` : si cette variable est à faux, alors on génère `'lighten-2'`, sinon `'darken-2'`.

Ainsi, la couleur de fond de conteneur sera gris clair et gris foncé respectivement lorsque `darkMode` sera à faux et `darkMode` sera à vrai.

Attention : la variable `darkMode` n'est accessible que depuis index.html, c'est-à-dire dans le fichier dans lequel est déclaré votre modèle. Pour y avoir accès dans vos composants, vous pouvez la passer en props :

```
<pokemon :pokemon="pokemon" :darkMode="darkMode"></pokemon>
```

5.3 Pagination

Vous aurez peut être remarqué la présence de lags dans l'interface. Ceci est probablement dû au fait que l'on affiche, sur une même page, 151 pokémons (donc 151 images, 151 tableaux, ...). Une solution simple est de mettre en place une pagination : on n'affichera simultanément qu'un certain nombre de Pokémons. Pour cela, vous allez utiliser deux choses :

- une nouvelle variable `pokemonsAffiches`, qui représente la liste des Pokémons à afficher ;
- le composant `v-pagination`, à placer en bas (ou en haut, comme vous voulez) de la page, qui va générer les contrôles permettant de changer de page.

Le composant s'utilise de la manière suivante :

```
<v-pagination
  @input="majPage"
  v-model="page"
  :length="nbPages"
  circle
>
</v-pagination>
```

On identifie quatre propriétés : `@input` permet de définir la méthode à appeler lors d'un changement de page (ici, la méthode va permettre de redéfinir `pokemonsAffiches`), `v-model` permet d'indiquer la variable dans laquelle stocker le numéro de la page courante, `:length` représente le nombre de pages (défini ici en fonction du nombre de Pokémons à afficher par page et du nombre de Pokémons récupérés depuis le serveur) et enfin `circle` définit la forme des boutons.

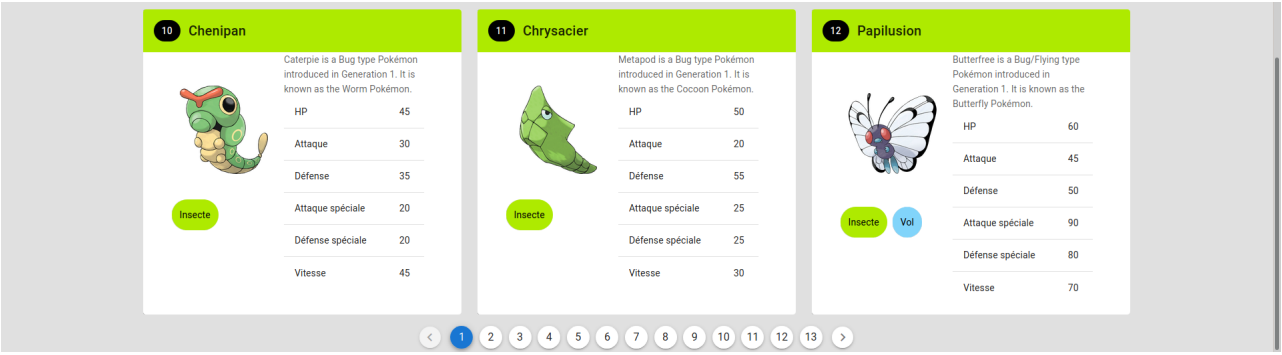

FIGURE 8 – Pagination

6 Bonus : affichage dynamique

En l'état, votre affichage est pratique sur un écran d'ordinateur ou de tablette, mais perfectible pour un smartphone. On va maintenant voir comment récupérer la taille de l'écran (plus précisément de la fenêtre) et l'utiliser pour générer un affichage dédié.

6.1 Récupération de la taille de la fenêtre

Récupérer la taille de la fenêtre n'est en soit pas bien compliqué. Il suffit d'appeler `window.innerWidth`, qui vous donnera la largeur actuelle de la fenêtre. Cependant, on aimerait également être prévenu lorsque cette valeur change. Pour cela, on va placer un *listener* sur la fenêtre :

```
mounted:function(){
  this.onResize();
  window.addEventListener('resize', this.onResize, { passive:
   ↪ true });
},
```

La méthode `mounted` est appelée après la création de la page et du modèle, mais avant le premier affichage. On fait appel à une méthode `onResize` développée par vos soins, qui va récupérer la taille de la fenêtre et déterminer si la politique d'affichage "mobile" s'applique. Juste après, on place un *listener* sur `window` qui, lors de la capture de l'évènement 'resize', appellera la méthode `onResize`. Il sera nécessaire, afin de nettoyer la mémoire, de supprimer ce *listener* lors de la fermeture de la page :

```
beforeDestroy:function(){
  if (typeof window === 'undefined') return;
  window.removeEventListener('resize', this.onResize, { passive:
   ↪ true });
},
```

6.2 Création d'un affichage adaptatif

Maintenant que l'on connaît en temps réel la largeur de la fenêtre, on va l'utiliser pour définir le composant à utiliser pour afficher les Pokémons. On choisira ici le composant v-expansion-panel. Chaque élément à afficher aura :

- un `v-expansion-panel-header`, qui contiendra le numéro, le nom et sera de la couleur du premier type du Pokémon concerné ;
- un `v-expansion-panel-content`, qui sera défini par un composant `pokemon-mobile` de notre création, reprenant en majorité la structure du composant `pokemon`, mais sans afficher le nom ni le numéro.

Lorsque l'on cliquera sur un élément de la liste, celui-ci se déploiera pour afficher le contenu :

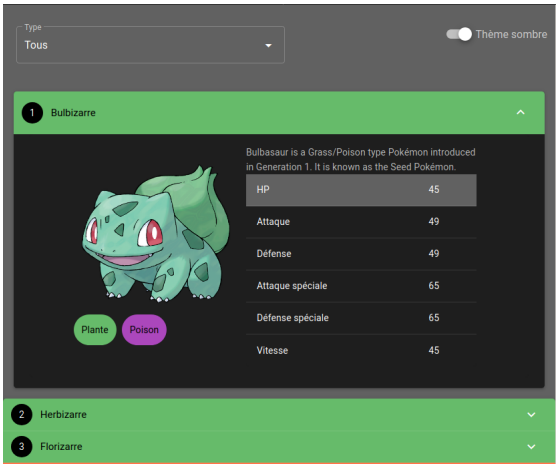

FIGURE 9 – Exemple d'agencement pour le mobile

Maintenant, il faut voir comment faire pour que, en fonction de la taille de la fenêtre, la liste des Pokémons s'affiche d'une façon où d'une autre. En fait, il suffit d'afficher le conteneur "grille" lorsque `window.innerWidth` est supérieure à un certain seuil, et le composant `v-expansion-panel` sinon. Cela se fait simplement via un `v-if` et un `v-else`, qui testeront la valeur d'une variable `affichageMobile`, dont la valeur sera définie dans la méthode `onResize`.