Projet Pokédex

Le but de ce projet est de créer un page affichant une liste de pokémon récupérée via une API Rest. On aura aussi une barre de recherche qui permet de filtrer la liste des pokémons et enfin lorsque l'on cliquera sur un pokémon on cherchera à afficher une pop-up qui contiendra des informations précises sur ce pokémon.

Toutes les fonctions et éléments utiles liées à Vue ou aux packages Node sont expliqués dans le cours.

Préparation du projet

Bien que nous ayons vu comment initialiser un projet Vue CLI, vous n'aurez pas besoin de reproduire cela pour la suite de ce projet. En effet, si ce n'est pas déjà fait vous allez cloner ce dépôt en local. Le dépôt par défaut contient déjà un Vue CLI initialisé ainsi qu'une architecture de base pour votre projet.

Une fois le projet cloné, ouvrez le dans VSCode, ouvrez aussi une fenêtre de terminal. Dans le dossier cloné, executez la commande.

```
npm install
```

Cela va installer en local tous les packages et dépendance nécessaires pour ce projet dans votre **node_modules**.

Pour lancer votre projet en local utilisez la commande:

```
npm run serve
```

Nous allons maintenant avancer étape par étape.

Architecture du projet

En plus des dossiers habituels d'un projet Vue CLI: package.json, public, babel-config.js, (...), on trouve aussi dans le dossier src/components des template de fichier .vue qui serviront de base pour les futurs parties du TP.

Tâcher de ne pas modifier les parties styles de ces fichiers si vous souhaiter garder une cohérence de CSS. Les classes et propriétés nécessaires afin d'avoir un style cohérent seront donnés au fil du TP. Rien ne vous empêchera à terme de les personnaliser et peut-être même d'ajouter un framework CSS pour les plus courageux !

main.js

Il contient déjà l'instanciation de notre composant App.vue ainsi que les paramètres par défaut de l'Application. Inutile de le modifier.

App.vue

Il s'agit du composant primaire de notre application, celui qui servira de componsant principal. Il contient la fameuse `<div id="app">` et utilise le composant Pokedex. C'est justement ce composant Pokedex que nous allons commencer à traiter dans la partie suivante.

config/config.json

Il s'agit d'un fichier qui contient l'URL de l'api permettant de récupérer les données des pokémons et la seconde URL sert de base de stockage des images des pokémons. Ne supprimez pas ces données vous en aurez besoin dans vos composants.

Actuellement si vous lancez votre application, elle n'affichera qu'une page blanche.

Les composants

Nous avons décidé de définir notre applications en plusieurs componsants que voici:

- PokemonList: le but de ce composant sera d'afficher la liste des pokémons
- PokemonDetail: son objectif est d'afficher la pop-up avec les détails d'un pokémon spécifique
- PokemonSearch: il s'agit d'une barre de recherche qui permettra de modifier la liste des pokémons en fonction de la recherche
- Pokedex: Les 3 composants précedants forment ensemble les fonctionnalités d'un pokédex. Ce sera donc ce composant qui sera en charge de contenir les autres.

Le composant Pokedex.vue

Nos différents composants vont être appelés au même endroit et communiqueront entre eux. Nous pourrions directement les mettre dans App.vue mais il est de convention de n'appeler qu'un composant dans l'App.vue. De plus structurellement parlant il est plus logique de créer un composant définissant un pokédex et toutes ses fonctionnalités.

Ce composant sera donc Pokedex.vue. Et il appelera tous les futurs composants que nous allons créer. Pour le moment créer le fichier et compléter le avec les appels des futurs composants `<PokemonDetail/>`, `<PokemonList/>` et `<PokemonSearch/>` que nous allons créer. Nous verrons durant ce tp s'il faut rajouter des choses supplémentaires.

Le composant PokemonList

Ce composant sert à afficher une liste de Pokémon. C'est par lui que nous allons commencer à travailler. Pour commencer à le tester n'oubliez pas de l'intégrer dans votre fichier Pokedex.vue afin qu'il s'affiche.

Partie `<script>`

Dans le cas du script, à la **création** du composant nous allons utiliser la bibliothèque axios afin de faire une requête vers l'URL de l'API. Vous devriez sûrement vous aider d'un hook de cycle de vie.

Pour récupérer la liste des pokémons il faudra ajouter le ciblage '/pokemon' à l'URL de l'API afin d'avoir la réponse voulue. N'hésitez pas à tester l'API depuis Postman afin de voir ce qu'elle renvoie ou via un console.log().

Dans cette réponse il y a la liste des pokémons que nous stockerons dans une propriété de **data** (par exemple *pokemons*). C'est cette propriété que nous utiliserons ensuite dans le `<template>`.

Pour générer les images dans le `<template>` il va nous falloir aussi récupérer l'IMG_URL et la stocker dans une propriété du **data**.

Pour rappel, les URL de l'API et de la base d'images sont stockées dans le config.json. N'hésitez pas à les importer dans votre composant pour les utiliser.

Partie `<template>`

Dans cette section on cherchera à afficher une liste de balise `<article>` qui contiendra chaque Pokémon récupéré. Cette balise contiendra des balises `<img>` et `<h3>` avec respectivement l'image et le nom du pokémon.

Attention afin d'afficher l'image il va falloir construire une url particulière avec le nom du pokemon. par exemple pour Tortank: https://img.pokemondb.net/sprites/bank/normal/blastoise.png

Dans la partie `<script>` vous avez normalement déjà récupéré l'IMG_URL, à vous de la compléter pour correspondre à l'exemple ci-dessus.

Le composant PokemonDetail

Ce composant sert à afficher le détail d'un Pokémon lorsque l'on cliquera sur un pokémon de PokemonList. Cela sous-entend qu'il va y avoir un lien entre ces deux composants.

Pour commencer à le tester n'oubliez pas de l'intégrer dans votre fichier Pokedex.vue afin qu'il s'affiche.

Partie `<script>` du PokemonDetail

Le composant *PokemonDetail* nécessitera une props *pokemonUrl* qui recevra l'url de l'api vers un pokémon en particulier. Ainsi lorsque l'on appelera notre div on pourra l'adapter au pokemon passé dans la props.

A la **création** du composant nous allons utiliser la bibliothèque axios afin de faire une requête vers l'URL stocké dans la props *pokemonUrl*. Vous devriez sûrement vous aider d'un hook de cycle de vie. La réponse sera stocké dans une propriété du **data**.

Pour générer les images dans le `<template>` il va nous falloir aussi récupérer l'IMG_URL et la stocker dans une propriété du **data**.

Nous aurons une méthode *closeDetail()* qui enverra un évènement au composant parent afin d'indiquer qu'il faut ferme/cacher le composant `<PokemonDetail>`. Vous trouverez plus d'information sur l'émission d'évènement dans la doc de la fonction $emit.

Partie `<template>` du PokemonDetail

Vous êtes assez libre sur la manière de construire l'HTML de la carte d'identité du Pokémon. Vous trouverez ci-dessous une liste non exhaustive de ce que vous pourriez afficher. A vous de voir quelles informations vous cherchez à afficher.

- Image du pokémon

- Nom
- Type
- Talents
- Tailles et Poids
- Sa lignée d'évolution

Afin de générer un css correct dans la dernière partie il est préférable d'inclure la totalité de vos informations à l'intérieur de la `<div class="data card-body">`.

Le bouton de fermeture existe déjà mais il faut ajouter un évenement lors d'un click. Cet évènement appelera la fonction *closeDetail()* définie dans le `<script>`.

Ajouter un évènement à PokemonList

C'est lors d'un clic sur un élément de la liste que s'affichera le composant `<PokemonDetail>`. La communication partira donc de `<PokemonList>` et remontera jusqu'à `<PokemonDetail>`. Pour ce faire, dans votre fichier *PokemonList.vue* il va falloir que lors du clic d'un `<article>` une fonction *showPokemonDetail()* soit déclenchée. Elle prendra en paramètre l'objet pokemon cliqué.

Cette fonction émettra ensuite un évènement pour communiquer avec son parent. Par convention l'évènement s'appelera *showPokemonDetailEmit* et il enverra le pokemon cliqué récupéré en paramètre. Vous trouverez plus d'information sur l'émission d'évènement dans la doc de la fonction $emit.

Modifier l'appel à PokemonDetail dans le fichier Pokedex.vue

Il ne faut pas oublier de rajouter les attributs et évènements à notre appel de `<PokemonDetail>`:

- La props pokemonUrl. Si votre props n'est pas encore automatiquement rempli et que vous voulez quand même tester votre composant vous pouvez lui donner la valeur par défaut suivante: https://pokeapi.co/api/v2/pokemon/1
- L'écoute de l'évènement closeDetail
- Une condition d'affichage du composant si un élément de la liste a été cliqué ou non. Vous pourriez utiliser une data booléen pour savoir cela.

Maintenant que nous avons ajouté l'émission d'un évènement *showPokemonDetailEmit* dans `<PokemonList>` il faut aussi l'écouter et appeler une fonction en réponse au niveau de l'appel du composant. N'oubliez pas que l'évènement envoie les informations d'un pokémon de la liste. Dans ces informations il y a l'url de l'api du pokémon en question. C'est peut-être cette url que nous aurons besoin de passer en props de notre composant `<PokemonDetail>`. A vous de réfléchir comment relier tout cela. Dîtes vous simplement que la communication de l'info part de `<PokemonList>` atteint `<Pokedex>` et doit aller jusqu'à `<PokemonDetail>` via sa props.

Utiliser le css par défaut

Rajouter les classes suivantes aux balises ci-dessous:

- image: Balise `<img class="image"/>`
- nom: Balise `<h2 class="card-title">`

- les types: Pour les types cela sera un peu plus compliqué. Définissez une balises `<div class="types">` qui contiendra la liste des types. Cette balise *types* contiendra une balise `<div class="type">` pour chaque type que possède le pokémon. Enfin un *type* est composé d'une balise `<span>` qui contient le nom du type. Le nom du type sera aussi utilisé comme valeur de *class*. En effet si vous regardez la partie`<style>` du composant une classe pour chaque type a déjà été créée.
- les autres informations comme la taille par exemple peuvent être écrite selon le modèle ci-dessous dans des `<div class="property">`:

```
<div class="property">
  <div class="left bold">Taille</div>
  <div class="right">La valeur contenant la taille du pokémon</div>
</div>
```

Le composant PokemonSearch

Nous allons maintenant créer une barre de recherche qui permettra de filter la liste des pokémons. Attention la liste des pokémons fournie est en anglais, vous devrez donc chercher les noms des pokémons en anglais.

Partie `<template>` du PokemonSearch

Il s'agit d'une banal barre de recherche. Le template possèdera donc un champ `<input />` de type *text*. Vous pouvez y ajouter des attributs que l'on retrouve souvent dans ce cas: un placeholder par exemple. Vous aurez besoin de stocker la valeur de ce champs dans une data. N'oubliez pas la directive *v-model* !

Partie `<script>` du PokemonSearch

Nous souhaitons que notre page se mette à jour juste en tapant dans le champs de recherche, pas besoin de validation, on changera tout en temps réel. Pour cela il va falloir que l'on émette un évènement pour indiquer à la liste qu'une recherche est en cours. D'ordinaire on émet des évènements via la directive *v-on*. Mais si je veux que l'évènement soit envoyé dès que la valeur de l'input change il y a un moyen bien plus simple avec une nouvelle propriété de l'instance Vue: watch

La propriété watch est assez simple à comprendre, vous définissez quelle valeur vous souhaiter surveiller, par exemple une data. Si cette data est modifié de quelconque manière que ce soit (ici via notre v-model de l'input) alors le watcher s'active est exécute le code indiqué dans sa fonction.

Dans l'exemple ci-dessous, dès que la data question est modifié, alors le code contenu dans la fonction s'execute et alimente la data answer.

```
  data: {
    question: '',
    answer: 'I cannot give you an answer until you ask a question!'
  },
  watch: {
    // whenever question changes, this function will run
    question: function (newQuestion, oldQuestion) {
      this.answer = 'Waiting for you to stop typing...'
```

```
    }
  },
```

Vous trouverez plus d'informations sur la propriété *watch* ici.

Maintenant que nous avons vu cet exemple l'idée dans notre projet est de surveiller la valeur data liée au champs `<input>` et dès qu'elle est modifié émettre un évènement *searchPokemonEmit* auquel on joindra la valeur de notre recherche.

Modification du composant Pokedex

Notre appel à `<PokemonSearch>` devra écouter l'émission de l'évènement *searchPokemonEmit* déclaré précedemment. Il déclenchera une méthode *setPokemonSearch* qui renseignera une **data** avec la valeur récupéré de l'évènement. Cette data sera ensuite passé dans une nouvelle props du `<PokemonList>`.

Modification du PokemonList

Afin d'affiner notre liste il va falloir passer la recherche récupéré depuis `<PokemonSearch>`comme props du composant `<PokemonList>`.

Dans le template il va maintenant falloir afficher l'`<article>`uniquement si le nom du pokémon en question contient la valeur de la recherche. Mais si vous essayez d'implémenter un *v-if* au même niveau qu'un *v-for* vous risquez de vous heurter à de gros problèmes. On pourrait créer des div supplémentaires mais il y a un moyen bien plus simple: et si nous faisions que la condition se passe dans une propriété computed et qu'ensuite cette propriété computed soit utilisé par le *v-for* ?

Définissez une **propriété computed** *pokemonsFiltered* qui représentera le tableau de la liste des pokémons mais filtrés. Cette propriété vérifiera si la valeur de recherche est vide ou non.

- Si elle est vide alors il retournera le tableau du data qui stockait jusqu'ici notre liste de pokémon. Ce qui reviendra au comportement que l'on a depuis le départ.
- Si la valeur de recherche est renseigné nous allons utiliser le package lodash qui permet de faire des opérations complexes sur des tableaux. La fonction de lodash que nous utiliserons sera la fonction filter. Nous préciserons comme tableau de départ la liste de nos pokémons non modifiés récupéré depuis l'API. Et nous ferons le filtrage de ce tableau sur la propriété name. La fonction filter renverra un nouveau tableau contenant uniquement les éléments qui match avec notre recherche. C'est le résultat de cette fonction filter que nous retournerons dans notre propriété computed

Vous trouverez plus d'informations sur le package lodash et ses fonctions dans la documentation.

Du côté du `<template>` dans notre boucle nous utiliserons désormais notre **propriété computed** au lieu de notre **data**.