Activité - Moteur de recherche de jeux vidéo

Au cours de cette activité, nous allons créer une application React.JS permettant d'obtenir des informations sur des jeux vidéo. Nous utiliserons l'API RAWG.IO dont vous trouverez une courte description en annexe de ce document.

Le principe de notre application est simple : Un champ de recherche, présent sur la page d'accueil, permettra de trouver une liste de jeux.

Une fois cette liste affichée, nous pourrons appuyer sur le jeu de notre choix pour pouvoir afficher la liste des détails le concernant.

Prérequis

Node.JS doit être installé sur votre machine.

Création du projet

Ouvrez un terminal et lancez la commande de création du projet :

```
npm create vite@latest games -- --template react
cd games
npm install
```

Puis, ouvrez le dossier nouvellement créé avec votre éditeur de code.

Démarrer l'application en lançant la commande `npm run dev` dans le dossier.

> **Comment faire ?** Si vous avez bien ouvert le dossier de votre application avec Visual Studio Code, vous devriez pouvoir utiliser le terminal intégré pour pouvoir taper cette commande au bon endroit. Utilisez le menu `Terminal` pour l'ouvrir.

Ajout de Tailwindcss

Nous allons utiliser le framework CSS `Tailwindcss` afin de gagner du temps sur l'élaboration du design de notre application. Cet outil est très facile à intégrer dans une application React.JS et une page est dédiée sur le site officielle : https://tailwindcss.com/docs/guides/create-react-app

Installer Tailwindcss dans le projet

Dans un terminal, lancer les commandes suivantes:

```
npm install -D tailwindcss @tailwindcss/vite
```

Configurer le projet pour utiliser Tailwindcss

1. Étape 1

Configurez le plugin dans vite.config.js Ajoutez le plugin Tailwind CSS dans la configuration de Vite. Modifiez votre fichier vite.config.js comme suit :

```js
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'
import tailwindcss from '@tailwindcss/vite'

// https://vite.dev/config/
export default defineConfig({
plugins: [react(), tailwindcss()],
})
```

2. Étape 2

   Ajouter TailwindCSS en haut de votre fichier `/src/index.css`, et vider le reste de ce fichier.

   ```
   @import "tailwindcss";
   ```

3. Étape 3

   Remplacer la fonction `App`, du fichier `/src/App.js` par celle-ci:

   ```
   function App() {
       return (
           <h1 className="text-3xl font-bold underline">
               Hello world!
           </h1>
       )
   }
   ```

4. Lancez votre application

   Dans un terminal, positionné dans le dossier de votre projet, lancez la commande `npm run dev`. Après un court instant, vous devriez voir un texte similaire à celui-ci:

   ```
   VITE v6.0.11  ready in 464 ms

   ➜  Local:   http://localhost:5173/
   ```

```
➜  Network: use --host to expose
➜  press h + enter to show help
```

Si votre terminal le permet, cliquez sur l'adresse `http://localhost:5173`. Dans le cas contraire, consultez-la simplement avec votre navigateur Internet.

Création de la page d'accueil

> ⚠️ **Les exemples ci-dessous ne contiennent pas toujours les "imports" nécessaires. Pensez à utiliser les fonctions de votre éditeur de code pour les retrouver facilement.**

Le composant "Home"

Comme évoqué plus haut, la page d'accueil de notre application affichera directement le moteur de recherche, ainsi que la liste des résultats qu'il trouvera.

Pour nous laisser la possibilité de naviguer entre plusieurs pages, nous allons créer un composant pour chaque page.

Commençons donc par créer un dossier "pages" dans le dossier `src` de notre projet et créons, à l'intérieur, un composant `Home` (pages/Home.jsx).

```
import React from "react";

const Home = () => {

  return (
    <form className="my-2 sm:w-full md:w-2/3 mx-auto flex px-2 text-2xl">
      <input type="text" className="rounded-l border border-gray-500 flex-grow px-4
      <button type="submit" className="bg-blue-700 rounded-r text-white px-4 py-2">
    </form>
  );
}

export default Home;
```

Afficher le composant "Home"

Pour afficher notre nouveau composant, nous devons modifier le fichier "App.js" qui contient le composant racine de notre application. Modifiez le fichier pour qu'il ressemble à celui-ci:

```
import './App.css';
import Home from './pages/Home';
import React from 'react';

function App() {
  return (
    <Home></Home>
  )
}

export default App;
```

Explications :

> Comme pour n'importe quel composant React, nous avons besoin d'importer "Home" pour pouvoir l'utiliser.

> 🕶️ **À ce stade, vous devriez voir apparaître un champ de saisie de texte et un bouton.**

Afficher une liste

Pour le moment, nous n'avons pas encore de données. Nous allons donc utiliser un tableau écrit en dur. Comme nos données impactent directement l'interface graphique, nous devons utiliser un "state" (useState).

Nous pouvons ensuite utiliser la méthode "map" des tableaux JavaScript pour réaliser une boucle:

```
import React, { useState } from "react";

const Home = () => {

    // On utilise un state pour garder nos données
    const [games, setGames] = useState([
        { id: 1, name: "Jeux 1", rating: 4.6 },
        { id: 2, name: "Jeux 2", rating: 3.5 },
        { id: 3, name: "Jeux 3", rating: 4.2 },
        { id: 4, name: "Jeux 4", rating: 1.5 },
        { id: 5, name: "Jeux 5", rating: 3.7 },
        { id: 6, name: "Jeux 6", rating: 5 },
    ]);
```

```
    return (
        <> {/* Un fragment doit être ajouté pour ne retourner qu'un seul composant
            <form className="my-2 sm:w-full md:w-2/3 mx-auto flex px-2 text-2xl">
                <input type="text" className="rounded-l border border-gray-500 flex
                <button type="submit" className="bg-blue-700 rounded-r text-white p
            </form>

            {/* Ajoutons notre liste */}
            <ul className="sm:w-full md:w-2/3 mx-auto px-2 text-2xl">
                {games.map(game => (
                    <li className="py-2 px-4 border-b border-gray-500 flex" key={ga
                        <div className="text-2xl font-bold flex-grow">{game.name}</
                        <div>{game.rating}</div>
                    </li>
                ))}
            </ul>
        </>
    );
}

export default Home;
```

Explications :

> Un composant React.JS ne peut retourner qu'une seule balise (qui peut contenir des sous-balises). Pour répondre à cette exigence, nous encadrons notre JSX par un composant nommé `fragment`, représenté par une balise vide.

> Pour le moment, les données sont écrites en dur dans le code. On utilise la méthode "React.useState" pour gérer un state puisqu'une modification de ces données doit impacter l'interface graphique.

> Nous utilisons la méthode "map" des tableaux JavaScript (sur le tableau "game") pour créer une boucle qui génère autant de \<li\/\> que nous avons de jeux dans notre tableau. Notez l'ajout d'un attribut key dans la balise `<li>`. Il permet à React d'identifier chaque entrée dans la boucle. Il doit donc contenir une valeur différente pour chaque ligne. L'ID d'un jeu étant unique, il s'agit d'une valeur intéressante.

> 🕶️ Vous devriez maintenant voir apparaître une liste

Gestion de la logique

1. Le champ de recherche

   Le champ de recherche contient une donnée "affichée". Sa modification impacte donc l'interface graphique et il faut donc utiliser un "state". Commençons donc par intégrer cela dans notre fonction "Home" :

   ```
   const Home = () => {
   const [searchText, setSearchText] = useState(''); // Ajouter cela
   // suite de la fonction ...
   ```

   Puis modifions la balise \<input\/\> de cette manière afin d'utiliser notre state :

   ```
   <input type="text" className="form-control" autoFocus={true}
   onInput={ e => { setSearchText(e.target.value) } }
   value={searchText}
   placeholder='Rechercher' />
   ```

   > La valeur de notre champ doit être liée à celle du state. Nous associons donc "searchText" à la valeur du champ.

   > La balise \<input\/\> peut lever un événement "onChange" qui est déclenché à chaque modification de sa valeur. Nous profitons donc de cet événement pour mettre à jour le state à chaque modification du champ.

   > 💡 Si vous souhaitez vérifier que le state change bien, affichez le contenu de la variable "searchText" en plaçant ce bloc de code dans votre JSX : `{searchText}`

2. Le bouton de recherche

   La balise \<form\/\> peut lever un événement "submit" lorsqu'elle est actionnée. Nous allons donc créer une fonction qui sera exécutée lorsque l'utilisateur pressera le bouton "Rechercher". Afin d'empêcher le rechargement de la page, lié à la soumission du formulaire, nous annulerons le comportement par défaut à l'aide de la méthode `preventDefault`. Ajoutons ceci juste avant l'instruction "return" de la fonction `Home` (qui est un composant React):

   ```
   const handleSearch = (e) => {
       e.preventDefault();
   ```

```
    alert("La recherche est lancée !");
}
```

Puis modifions la balise \<form\/\> afin de capter l'événement "submit" :

```
<form className="my-2 sm:w-full md:w-2/3 mx-auto flex px-2 text-2xl" onSubmit=
```

> 🕶️ Vous devriez maintenant voir un message apparaître lorsque vous appuyez sur le bouton "Rechercher".

Récupérer les données d'une API

Pour nos premiers pas, nous allons utiliser une adresse URL fixe (qui retourne toujours les mêmes résultats). Modifions la fonction "handleSearch" comme ceci :

```
const handleSearch = (e) => {
  e.preventDefault();
  const url = 'https://www.formacitron.com/games-api-fallback/games/';
  fetch(url)
    .then( response => response.json() )
    .then( data => { setGames(data.results) } )
    .catch( () => { alert('Une erreur est survenue') } )
}
```

> La fonction "fetch" prend en argument une URL et retourne une promesse. La valeur de retour est une donnée brute qu'il faut traiter pour obtenir la donnée JSON attendue. Pour cela, il faut appeler la méthode "json" de la réponse qui retourne également une promesse.

> Une fois la seconde promesse résolue, nous obtenons les données retournées par l'API. Parmi elles, une propriété nommée "results" est un tableau qui contient une liste de jeux. Nous pouvons donc mettre à jour notre state avec cette nouvelle liste.

> ⚙️ Nous pouvons maintenant retirer les données que nous avions fixées en dur dans le state : `const [games, setGames] = useState([]);`

> 🕶️ Vous devriez maintenant obtenir une liste de jeux issue de l'API lorsque vous appuyez sur le bouton "Chercher"

Améliorer l'apparence de la liste

Comme l'API est capable de retourner une illustration pour les jeux. Nous allons essayer d'ajouter une image pour chaque élément de la liste.

Modifions donc la liste pour y ajouter une image :

```
<ul className="sm:w-full md:w-2/3 mx-auto px-2 text-2xl">
  {games.map(game => (
    <li className="py-2 px-4 border-b border-gray-500 flex" key={game.id}>
    <img src={game.background_image} alt="" className="w-24 pr-2" />
      <div className="text-2xl font-bold flex-grow">{game.name}</div>
      <div>{game.rating}</div>
    </li>
  ))}
</ul>
```

Utiliser l'API RAWG.IO

Consulter l'annexe réservée à l'API RAWG.IO à la fin de ce document. Vous y trouverez les informations nécessaires pour vous inscrire, obtenir une clé d'API et faire vos premières requêtes.

Pour faire en sorte que notre application utilise l'API RAWG.IO, il suffit de modifier la fonction "handleSearch" de cette manière :

```
const handleSearch = (e) => {
e.preventDefault();

const apiKey = 'XXXXXXXXXXXXXXXXXXXXXXXXXXXX';
const url = `https://api.rawg.io/api/games?key=${apiKey}&search=${encodeURI(searchT
fetch(url)
.then( response => response.json() )
.then( data => { setGames(data.results) } )
.catch( () => { alert('Une erreur est survenue') } )
}
```

> ⚙️ Remplacez la suite de X par votre clé API personnelle.

> Comme la saisie de l'utilisateur peut contenir des caractères interdits dans une URL, il est nécessaire d'utiliser la fonction "encodeURI". Cette méthode remplace tous les caractères non autorisés dans une URL par leur équivalent codé.

> 🕶️ Notre recherche fonctionne !

Naviguer entre les pages

Maintenant que nous pouvons chercher des jeux et en sélectionner un, il nous faut afficher une seconde page qui présentera le détail du jeu sélectionné.

1. Ajouter React Router à votre projet

   React Router est proposé sous la forme d'un paquet externe que vous devez ajouter à votre application. Pour cela, lancez la commande suivante avec un terminal, dans le répertoire de votre projet :

   ```
   npm install react-router-dom
   ```

   > ⚠️ **Il peut être nécessaire d'arrêter, puis de relancer `npm run dev` après l'installation d'un module NPM. Gardez cela en tête au cas où le comportement de votre application n'est plus celui attendu.**

2. Création d'une seconde page

   Comme nous souhaitons afficher une page de détails pour le jeu de notre choix, nous allons créer un second composant. Pour l'instant, nous allons y placer le minimum, et nous y reviendrons quand la navigation fonctionnera.

   > ⚙️ Créez un fichier "Details.jsx" dans le répertoire pages et remplissez-le comme ceci pour le moment :

   ```
   const Details = () => {
       return (
           <div>Ceci est la seconde page</div>
       )
   }

   export default Details
   ```

3. Une page d'erreur

   En cas d'erreur d'url, nous devons prévoir une page d'erreur. Nous allons donc créer un autre composant `ErrorMessage.js` dans le dossier `pages`.

```
const ErrorMessage = () => {
    return (
        <h1>Oups, vous vous êtes égaré !</h1>
    )
}

export default ErrorMessage;
```

4. Le composant racine de navigation

Nous allons mettre en place un composant de navigation dans le fichier App.js de cette manière :

```
import './App.css';
import Home from './pages/Home';
import React from 'react';
import { createBrowserRouter, RouterProvider } from 'react-router-dom';
import ErrorMessage from './pages/ErrorMessage';
import Details from './pages/Details';

function App() {

  // Création du routeur
  const router = createBrowserRouter([
    {
      path: "/",
      element: <Home />,
      errorElement: <ErrorMessage />,
    },
    {
      path: "/details/:slug",
      element: <Details />,
    },
  ], { basename: "/" })

  return (
    <RouterProvider router={router}></RouterProvider>
  );
```

```
}

export default App;
```

> Le routeur vient remplacer le composant racine de notre application. On peut y définir différentes "routes". Une route définie une forme d'URL attendue, ainsi que le composant à afficher en conséquence.

> La seconde route attend un argument ":slug". Cet argument pourra être récupéré dans le composant à afficher. Nous aurons besoin de savoir quel est le jeu qui a été choisi par l'utilisateur.

> Le second paramètre de la fonction `createBrowserRouter` ({ basename: "/"}), permet de définir l'url racine de notre application. Plus tard, si nous hébergeons notre application dans un sous répertoire, nous pourrons modifier la valeur `/` pour qu'elle reflète cet emplacement.

5. Naviguer de Home vers Details

Pour naviguer vers la page de détails, il suffit d'encadrer chaque élément de la liste avec un composant \<Link\/\>. Comme nous aurons besoin de savoir quel jeu afficher, et que nous avons prévu un paramètre «slug» dans l'URL, nous allons créer un lien contenant cette information.

Modifiez la liste du composant Home de cette manière:

```
<ul className="sm:w-full md:w-2/3 mx-auto px-2 text-2xl">
  {games.map(game => (
    <li className="py-2 px-4 border-b border-gray-500" key={game.id}>
      <Link to={`/details/${game.slug}`} className="flex">
        <img src={game.background_image} alt="" className="w-24 pr-2" />
        <div className="text-2xl font-bold flex-grow">{game.name}</div>
        <div>{game.rating}</div>
      </Link>
    </li>
  ))}
</ul>
```

N'oubliez pas l'import de la balise Link de React Router

```
import { Link } from "react-router-dom";
```

> 🕶️ Vous pouvez maintenant naviguer de la page de recherche vers la page "Details" en cliquant sur un jeu dans la liste.

6. Récupérer le slug du jeu dans la page de détails

   Il est possible de récupérer les paramètres d'URL grâce à un "hook" spécifique nommé "useParams". Nous pouvons donc recevoir le slug dans le composant Details en le modifiant de cette manière :

```
import { useParams } from "react-router-dom"

const Details = () => {
const params = useParams();
return (
<div>Ceci est la page du jeu dont le slug est "{params.slug}"</div>
)
}
```

   Afin de rendre le code plus lisible, vous pouvez utiliser la déstructuration comme ceci :

```
import { useParams } from "react-router-dom";

const Details = () => {
const {slug} = useParams();
return (
<div>Ceci est la page du jeu dont le slug est "{slug}"</div>
);
}
export default Details;
```

> 🕶️ Votre page de détails affiche maintenant le slug du jeu sélectionné

Votre nouvelle mission

Vous avez maintenant tous les éléments pour concevoir la page de détails. Il vous faudra réaliser une nouvelle requête API pour obtenir les informations du jeu souhaité, puis afficher comme vous le souhaitez les informations retournées.

Comme les données sont à charger au démarrage du composant, vous aurez besoin du hook React "useEffect". La documentation est ici : https://react.dev/reference/react/useEffect. *Attention*: N'oubliez pas d'ajouter un tableau vide en second argument de useEffect, sous peine de déclencher une boucle infinie qui épuisera votre quota d'accès à l'API.

Certaines données de l'API RAWG contiennent des balises HTML. Si vous rencontrez des difficultés pour les afficher, faites une recherche sur `dangerouslySetInnerHTML`.

> ⚙️ **Terminez maintenant cette application.**

ANNEXE

Utilisation de l'API RAWG.IO

Au cours de cette activité, nous allons utiliser l'API du site RAWG.IO. C'est une base de données des jeux vidéo accessible gratuitement.

Inscription

Pour pouvoir utiliser cette API, vous devrez obtenir une clé d'API. C'est une sorte de grand mot de passe qui permettra de prouver à l'API que vous avez le droit de l'utiliser

Rendez-vous sur le site https://rawg.io/login/?forward=developer et inscrivez-vous.

Une fois inscrit et connecté, vous devriez pouvoir créer une clé API que vous devrez conserver. Elle sera utilisée pour chaque requête envoyée à l'API.

> 💡 Dans les exemples suivants, à chaque fois que vous verrez "{apiKey}", vous devrez remplacer cette expression par votre clé API (y compris les accolades).

> 💡 Dans les URL que vous trouverez ci-dessous, les expressions entre accolades correspondent toujours à un bloc qu'il faut remplacer.

Principe

L'API RAWG.IO s'interroge grâce à de simples requêtes HTTP. Vous pouvez donc tester les différentes fonctionnalités en tapant les adresses directement dans votre navigateur web.

Toutes les requêtes commencent par : https://api.rawg.io/api/

Cette adresse sera complétée en fonction de ce qu'on souhaite obtenir.

Rechercher un jeu

Pour rechercher des jeux, vous devrez utiliser l'adresse suivante :

https://api.rawg.io/api/games?key={apiKey}&search={titreDuJeu}

> 💡 {titreDuJeu} devra être remplacé par le nom du jeu que vous cherchez.

> ⚠️ Attention : les paramètres que vous passez dans l'URL doivent respecter la structure de l'URL. Ils ne doivent donc pas contenir de caractères réservés (&,\/, :, etc.)

L'API retourne un nombre de résultats limité (20). Il peut être nécessaire de compléter l'adresse avec l'argument `&page={numeroDeLaPage}`. L'argument devra être remplacé par le numéro de la page souhaitée.

> ⚙️ Faites un essai dans votre navigateur et étudiez la structure retournée.

Consulter la fiche d'un jeu

Pour pouvoir réaliser cette action, vous devrez connaître le "slug" correspondant à ce jeu. Un slug est une chaîne de caractères unique, compréhensible par un humain, et servant d'identifiant. Vous pourrez trouver cette information dans les données retournées par la recherche à l'étape précédente.

Pour consulter la fiche d'un jeu, vous utilisez l'URL suivante :

https://api.rawg.io/api/games/{slug}?key={apiKey}

> 💡 L'argument "{slug}" doit être remplacé par le slug du jeu souhaité.

Consultez la documentation

La présente annexe présente les fonctionnalités de base de l'API. Je vous invite à consulter la documentation officielle pour découvrir toutes les autres.

> 💡 Documentation officielle de l'API : https://api.rawg.io/docs/

Adresse de secours

Il arrive parfois que le site RAWG.IO soit en panne. Dans ce cas, vous pouvez utiliser l'adresse suivante comme base :

https://www.formacitron.com/games-api-fallback/games

> ⚠️ Cette adresse fournit une courte extraction du site RAWG. Elle retournera toujours la même liste de 6 jeux. Elle est donc à utiliser en dernier recours pour ne pas perdre de temps.