Exécuter des requêtes HTTP en AJAX dans une application React

L'AJAX, *Asynchronous JavaScript + XML* (nom historique car on pourra récupérer toutes sortes de réponses, souvent du JSON), c'est le fait de faire une requêtes HTTP depuis un script Javascript exécuté dans le navigateur vers un serveur et de récupérer et utiliser la réponse obtenue dans ce script, sans rechargement de la page. Cela permet de récupérer des données dans un script et de s'en servir pour modifier une partie de la page consultée par exemple, sans pour autant recharger toute la page.

Dans nos applications avec React, on décrit beaucoup d'interactions et on souhaite souvent modifier une partie du document. Il est fréquent d'avoir besoin d'échanger des données avec un serveur, que ce soit pour récupérer des données pour alimenter notre interface, ou pour envoyer des données pour les faire persister sur un serveur.

Pour effectuer une requête HTTP depuis un script plusieurs options s'offrent à nous :

- Via XMLHttpRequest (MDN) – old school et pratique, mais syntaxe lourde
- Via fetch (MDN) – new school et pratique, pourquoi pas
- Via une bibliothèque (comme Axios) – offrira des petites possibilités de configuration parfois bien pratiques

Voyons comment déclencher nos appels dans nos composants. Puis on détaillera la solution Axios.

1. Déclencher un appel dans un composant

1.1 En réponse à une interaction

```
const Demo = () => {
  const handleClick = () => {
    // je pourrais déclencher ma requête http ici
  };
  return (
    <button onClick={handleClick}>Faire une action</button>
  );
}
```

1.2 Pour un chargement initial

1.2.1 En classe

```
class Demo extends React.Component {
  componentDidMount() {
    // je pourrais déclencher ma requête http ici
  }
  render() {
    return (
      <div>Hello</div>
    );
  }
}
```

1.2.2 En fonction

```
const Demo = () => {
  useEffect(() => {
    // je pourrais déclencher ma requête http ici
  }, []);
  return (
    <div>Hello</div>
  );
}
```

Attention : Faire une requête HTTP n'est jamais instantané. D'où le A de AJAX pour **Asynchrone**. En attendant la réponse, notre application n'est pas en pause. On pourra donc décrire l'application dans un état de chargement tant qu'une réponse n'est pas arrivée, puis quand elle arrive décrire un état chargé.

2. Faire de l'AJAX avec Axios

Axios est une bibliothèque qui va construire des XMLHttpRequests et des Promesses de manière pratique.

On l'installe yarn add axios

2.1 Sous le capot

Si on veut comprendre un peu plus en détail comment fonctionne Axios, on doit déjà comprendre ce qu'est une promesse ou *Promise* en anglais

Définition MDN – Javascript.info

On va construire une promesse avec le constructeur Promise et lui passer en argument une fonction qui sera exécutée immédiatement. Cette fonction contient notre tâche longue, des instructions qui prendront un certain temps. Cette fonction acceptera 2 callbacks en paramètres, un premier représentant « quoi faire quand tout s'est bien passé », un second représentant « quoi faire en cas d'erreur ». On

décidera d'exécuter ces callbacks au moment opportun dans la fonction passée au constructeur Promise.

L'objet donné par le constructeur Promise est notre promesse, sur laquelle on peut appeler une méthode then à qui on passe en argument le callback qui définit « quoi faire quand tout s'est bien passé ». Via une méthode catch on passera en argument le callback qui définit « quoi faire en cas d'erreur ».

Un exemple

```
console.log('1. Avant la construction promesse');
const promesse = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('hello');
  }, 1000);
});

promesse.then((message) => {
  console.log('3. Après la résolution de la promesse', message); //
hello
});

console.log('2. Après la construction de la promesse');
```

Reproduisons une version simplifiée de ce que fait Axios, juste pour mieux visualiser

```
function fakeAxios(url) {
  // axios viendra nous retourner une promesse
  return new Promise((resolve, reject) => {
    // dans laquelle on fait une XHR
    const xhr = new XMLHttpRequest();
    xhr.open("GET", url);
    // et où on appelle quoi faire quand la réponse arrive
    xhr.onload = () => resolve(xhr.responseText);
    xhr.onerror = () => reject(xhr.statusText);
    xhr.send();
  });
}
// ainsi on pourra aisément créer nos promesses
const promise =
fakeAxios('https://oclock-open-apis.now.sh/api/blog/posts');
// et décrire quoi faire quand la réponse arrivera
promise
  .then((response) => {
    console.log('reponse obtenue');
    console.log(response);
```

```
  })
  .catch((error) => {
    console.error(error);
  });
```

2.2 Utilisation dans la vraie vie

Inutile de réinventer la roue maintenant qu'on a vu le principe. La documentation d'axios fourmille d'exemples pratiques

Cela donne

```
import axios from 'axios';

axios({
  url: '/posts',
  method: 'GET',
  baseURL: 'https://oclock-open-apis.vercel.app/api/blog';
})
  .then((response) => {
    console.log('reponse :', response);
  })
  .catch((error) => {
    console.error('error :', error);
  })
  .finally(() => {
    console.log('à faire en cas de succès ou d'erreur')
  });
```

Ou si l'on préfère via une méthode alternative

```
axios.get('/posts', {
  baseURL: 'https://oclock-open-apis.vercel.app/api/blog',
})
  .then((response) => {
    console.log('reponse :', response);
  })
  .catch((error) => {
    console.error('error :', error);
  })
  .finally(() => {
    console.log('qqch à faire en cas de succès ou d'erreur')
  });
```

Comparer l’approche fonctionnelle apportée par les hooks pour gérer ses effets de bord à l’approche objet

Il pouvait arriver qu’on ait besoin de décrire des effets à exécuter après un rendu. Pour cela on se plaçait dans des composants en classes et on pouvait définir les méthodes des Lifecycles componentDidMount et componentDidUpdate pour réagir après un rendu, après le travail de ReactDom. Typiquement, on pouvait ainsi prévoir :

- Des modifications du DOM au dessus de notre application React (sur window, body, …)
- La lecture d’infos du DOM réel via des ref par exemple (pour connaitre une largeur calculée ou une position)
- Le déclenchement d’appels AJAX initiaux

```
class Demo extends React.Component {

  constructor(props) {
    // on pose les props et le state sur l'instance
  }

  componentDidMount() {
    // après le rendu initial
  }

  componentDidUpdate() {
    // après les rendus de mise à jour
  }

  render() {
    // on retourne l'élement React
  }
}
```

C’était pratique. Mais on peut préférer l’approche fonctionnelle apportée par les hooks souvent plus digeste.

Le hook d’effet useEffect rempli exactement le rôle des méthodes des lifecycles citées précédemment mais dans un composant écrit en fonction.

La doc React

Exemples

```
import React, { useEffect } from 'react';
```

```
const Demo = ({ title }) => {
  // on passe un callback en argument lorsqu'on execute useEffect,
cette fonction sera exécutée après chaque rendu
  // après que le DOM réel ait été calculé
  // c'est l'équivalent du componentDidMount + componentDidUpdate
  // puisqu'on réagit après le rendu initial ET les rendues de mises
à jour
  useEffect(() => {
    document.title = title;
  });
  return <h1>{title}</h1>;
};
```

useEffect accepte un 2ème argument, un tableau de dépendances. Très pratique pour l'optimisation

```
const Demo = ({ title, text }) => {
  // ici l'effet sera exécuté si title change,
  // s'il n'y a que text qui change l'effet ne se sera pas exécuté
  // niveau optimisation c'est top, on évite d'exécuter des
instructions qui n'ont pas besoin de l'être
  useEffect(() => {
    document.title = title;
  }, [title]);
  return (
    <>
      <h1>{title}</h1>
      <p>{text}</p>
    </>
  );
};
```

On peut exploiter ce tableau de dépendances pour limiter notre effet au premier rendu, au rendu initial du composant.

Pour cela on passe un tableau vide

```
const Demo = ({ title }) => {
  // ici on n'exécutera l'effet qu'après le rendu initial
  useEffect(() => {
   fetchDataFromApi();
  }, []);
  return <h1>{title}</h1>;
```

```
};
```

Pour résumer :

```
useEffect(() => {}); // après tous les rendu
useEffect(() => {}, []); // uniquement au rendu initial
useEffect(() => {}, [toto, tata]); // après tous les rendu où toto et/ou tata a changé
```

Expliquer les risques liés aux failles XSS

Il peut vite arriver qu'on ouvre sans le faire exprès des failles XSS dans nos applications

C'est quoi

Les failles XSS ou Cross Site Scripting consistent, pour un pirate, à injecter des scripts sur un autre site. Ainsi ces scripts sont exécutés dans le navigateur d'utilisateurs pensant utiliser un site de confiance et pourront par exemple envoyer les infos saisies par ces utilisateurs vers un serveur malintentionné. Ou bien de manière plus flagrante ces scripts pourraient juste rediriger sur un site de pub.

On peut lire les cheatsheets d'OWASP pour plus d'infos sur la sécurité (il y en a beaucoup)

ex : Je récupère des données d'une API, sauf qu'il y a du html dedans

```
const data = 'coucou ça va <img src="/existepas" onerror="javascript: /*du js malintentionné*/ window.location.href = `https://youtu.be/dQw4w9WgXcQ`;" />';
```

Si derrière j'injecte ce html, la faille est ouverte

```
// pas bien
document.body.innerHTML = data;
```

De même avec React si j'utilise dangerouslySetInnerHTML qui permet de placer du HTML interprété – doc

```
// pas bien
function createMarkup() {
  return {__html: data};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

Comment se protéger

On préférera *dans la mesure du possible* ne jamais interpréter le HTML

```
// bien
document.body.textContent = data;

// bien
function MyComponent() {
  return <div>{data}</div>;
}
```

Si on n'a pas le choix, si c'est un besoin d'interpréter le HTML : il faut impérativement nettoyer tout le HTML qu'on injectera.

On ne fera jamais confiance à une donnée reçue par notre application, même si cette donnée vient d'une API qu'on a codé nous même.

On n'est jamais à l'abri de laisser traîner une erreur quelque part. Donc autant nettoyer chaque fois qu'on injecte du html interprété.

Pour ça on peut y aller à coup d'expressions régulières pour analyser les chaines de caractères, détecter le html et le nettoyer.

Ou bien on peut utiliser un outil qui le fait pour nous, *pour lequel toute une communauté peut participer à la robustesse de cet outil.*

On a choisi DOMPurify qui gère très bien le nettoyage du HTML

```
// bien
document.body.innerHTML = DOMPurify.sanitize(data);
// bien
function createMarkup() {
  return {__html: DOMPurify.sanitize(data)};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

En se référant à la doc, on voit en plus qu'on peut configurer notre nettoyage pour autoriser certaines choses

```
function createMarkup() {
  return {__html: DOMPurify.sanitize(data, {
    ALLOWED_TAGS: ['strong', 'em'],
  })};
```

```
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

Organiser ses appels aux hooks dans ses propres hooks

Nos hooks personnalisés seront tout simplement des fonctions qui vont contenir une logique réutilisable (comme n'importe quelle fonction en fait). Comme n'importe quelle fonction ils seront paramétrables et pourront retourner une valeur.

On parle de hooks parce qu'ils pourront contenir des appels à d'autres hooks, c'est à dire des appels à des fonctions qui ne fonctionnent que dans des composants écrits sous forme de fonction. Rendant ainsi nos hooks personnalisés, à leur tour, utilisables uniquement dans un composant écrit sous forme de fonction.

Pour identifier facilement un hook, on le nommera par convention avec nom commençant par **use**

Voir la doc

Exemple :

```
import { useEffect } from 'react';
import { useLocation } from 'react-router-dom';

// notre définition de hook personnalisé
function useTitle() {
  let title;
  // on peut y appeler d'autres hooks
  const location = useLocation();

  switch (location.pathname) {
    case '/angular':
      title = 'Découvrez Angular';
      break;
    case '/react':
      title = 'Apprenez React';
      break;
    case '/oclock':
      title = 'Une bonne école';
      break;
    default:
      title = 'Dev Of Thrones';
```

```
  }

  useEffect(() => {
    document.title = title;
  });

  return title;
}

export { useTitle };
```

Dans des composants on pourra l'utiliser

```
import React from 'react';
import { useTitle } from 'src/hooks';

import './style.scss';

const Title = () => {
  const title = useTitle();
  return (
    <h1 className="title">{title}</h1>
  );
};

export default Title;
```

Employer une solution de routeur

Nous avions déjà appliqué la solution react-router-dom pour répondre à notre problématique de router.

Une fois confrontés de nouveau à cette problématique, comme tout bon développeur, nous nous sommes appuyés sur les projets passés et la documentation pour retrouver les solutions.

On retrouve notamment principalement le composant BrowserRouter comme prérequis, Link pour changer l'adresse et alimenter l'historique au clic sur un lien et Route pour conditionner le rendu de composants en fonction de l'adresse actuelle.

Toutefois comme dans n'importe quel projet, nous avons dû faire face à des problématiques inédites et trouver des solutions.

La problématique est la suivante : comment récupérer la partie changeante d'une route paramétrée ?

Lorsque je me rends sur /article/hello-world

Si j'ai placé la route paramétrée suivante dans mon projet

```
<Route path="/article/:slug">
  <Article />
</Route>
```

Si j'ai besoin de la partie changeante de mon adresse, ici « hello-world » dans le composant Article, soit je bricole ma solution en passant par window.location, soit plusieurs solutions fournies par react-router-dom s'offrent à moi :

- Utiliser le hook useParams dans le composant
- Utiliser les props *component* ou *render* du composant Route pour avoir accès à des props supplémentaires dans le composant
- Utiliser le high order component withRouter pour alimenter mon composant en props supplémentaires

Un peu de détails sur la solution *useParams*

Il s'agit d'un *hook* fourni par React Router. Comme toute fonction hook, elle est utilisable dans un composant. Elle offre la capacité à mon composant d'obtenir les informations de la route paramétrée au dessus de ce composant. On récupère un objet avec en clé les noms que j'ai donné aux parties changeantes de ma route et en valeurs les informations de l'adresse actuelle

```
import React from 'react';
import { useParams } from 'react-router-dom';

const Article = () => {
  const params = useParams();
  const slug = params.slug;
  // ou avec destructuring : const { slug } = useParams();
  return <p>Vous êtes sur l'article possédant le slug {slug}</p>;
};
```