Jean-Baptiste Cazaux.
Développeur & formateur react.

React par la pratique

Partie 2

Rentrons dans Reactjs en développant une application qui permet de chercher des gifs animés et de les afficher. L'application est simple car elle n'est composée que d'un champ de saisie pour rechercher les images, et des images renvoyées par le serveur. Nous utiliserons l'API que le site Giphy met à disposition des développeurs.

L'application sera construite de façon itérative, en respectant les bonnes pratiques de développement. L'outil create-react-app nous permettra de générer le squelette de l'app, sans avoir à se soucier des détails des configurations de Webpack et Babel.

Les sources de cette application sont disponibles sur Github : github.com/jbcazaux/react-giphy. Elles sont là soit juste pour regarder le résultat, soit pour aider à suivre le tutoriel.

Avant toute chose, il faut installer l'environnement de développement.

- Pour installer Node, le plus simple est sans doute d'utiliser nvm (Node Version Manager), disponible sur Windows, Linux et macOS.
- Ensuite il faut soit *cloner* le repository :
  - Installer git ;
  - git clone **https://github.com/jbcazaux/react-giphy.git**
- Soit créer un projet dès le début :
  - Lancer *npx create-react-app react-giphy* (react-giphy ou un autre nom ;))
- Lancer *npm install* pour télécharger les dépendances.

Plusieurs fichiers ont été générés dans le répertoire. Il faudra en supprimer certains et en modifier ou créer d'autres.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    <title>React-Giphy</title>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

public/index.html peut être simplifié, il faut juste retenir le * * qui est l'élément qui accueillera l'application Reactjs. Le fichier sera modifié automatiquement par Webpack qui y inclura les sources js contenant notre application et les librairies, le tout minifié.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

ReactDOM.render(<App />, document.getElementById('root'));
```

src/index.js est le point d'entrée de l'application Reactjs. Il faut indiquer le composant react racine (ici *App*) et l'élément html (ici *root*) dans lequel l'application sera construite.

```
import React from 'react';

class App extends React.Component { // Les composants React doivent étendre React.Component

    constructor(props) {
        super(props);
        this.state = { // this.state permet de stocker l'état de notre composant
            query: '' // ici la saisie de l'utilisateur
        };
    }

    render() {
        return (
            <div className="App">
                <form>
                    <input type="text" value={this.state.query}/>
                </form>
            </div>
        );
    }
}

export default App;
```

App.js est le fichier contenant notre composant racine. La méthode *render()* retourne du JSX. C'est une façon d'écrire le template html associé au composant.

Dans cette première version, il s'agit d'afficher simplement un champ texte qui permettra de saisir la recherche.

En laissant le code comme ceci, si on lance l'application (*npm start*) et que l'on tente de saisir du texte dans le champ input, rien ne va s'afficher.

En effet React prône les flux **unidirectionnels**. On a un lien state.query input, et pas input state.query. Pour mettre à jour la valeur affichée dans le champ *input*, il faudra modifier la valeur de *state.query*.

Afin de faire ceci on va ajouter la détection d'une saisie par l'utilisateur, grâce à l'attribut *onChange*.

```
import React from 'react';

class App extends React.Component {

    constructor(props) {
        super(props);
        this.state = {
            query: '',
        };
        this.updateQuery = this.updateQuery.bind(this); // afin que 'this' puisse être utilisé dans la méthode
    }

    updateQuery(event) {
        // mise à jour de l'état du composant
        this.setState({query: event.target.value});
    }

    render() {
        return (
            <div className="App">
                <form>
                    <input type="text" value={this.state.query} onChange={this.updateQuery}/>
                </form>
            </div>
        );
    }
}

export default App;
```

onChange va appeler la méthode indiquée (ici *updateQuery*) en passant en paramètre l'évènement. Il faut ensuite aller chercher la valeur de la saisie dans *event.target.value*.

En relançant l'application (il suffit d'enregistrer le fichier, le serveur va se mettre à jour tout seul), on peut contrôler que le champ *input* répond bien à la saisie.

On va maintenant faire la requête vers Giphy et afficher les résultats. Les résultats seront stockés dans *state.results*. La requête sera

relancée chaque fois que la saisie évolue. Pour appeler l'API il faut une clé. Une simple inscription sur https://developers.giphy.com suffit à en récupérer une gratuitement. L'exécution de *setState()* étant asynchrone (React peut décider de regrouper plusieurs *setState()* pour tous les faire en une fois), on va déclarer un callback, qui sera appelé une fois le *setState()* réellement effectué.

On va appeler *axios.get()* qui permet de faire un appel ajax REST et qui retourne une promesse. Il n'y a plus qu'à lire le contenu de la réponse de l'API et de stocker le résultat. Une fois l'état (*state*) mis à jour, React rappelle automatiquement la méthode *render()*, qui, elle, va lire le contenu du résultat stocké dans *state.results*, et l'afficher. *Axios* est une librairie qui ne fait pas partie de React, il faut donc l'installer: *npm install --save axios*.

```
import React from 'react';
import axios from 'axios';

class App extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      query: '',
      results: []
    };
    this.onQueryUpdate = this.onQueryUpdate.bind(this);
    this.fetchGifs = this.fetchGifs.bind(this);
  }

  onQueryUpdate(event) {
    // fetchGifs sera appelé après la mise à jour du state
    this.setState({query: event.target.value}, this.fetchGifs);
  }

  fetchGifs() {
    axios
      .get('http://api.giphy.com/v1/gifs/search', {
        params: {
          api_key: 'OSGK9oQq63idExZPvIlGHf5qSVXlth56',
          limit: 2,
          q: this.state.query
        }
      })
      .then(response => response.data)
      .then(giphydata => this.setState({results: giphydata.data})) // stocker le résultat dans state.results
      .catch(err => console.log(err));
  }

  render() {
    return (
      <div className="App">
        <form>
          <input type="text" value={this.state.query} onChange={this.onQueryUpdate}/>
        </form>
        <div>
          {this.state.results.map(result => <img src={result.images.downsized.url}
                                                 alt={result.title}
                                                 key={result.id}/>)}
        </div>
      </div>
    );
  }
}

export default App;
```

L'affichage se fait en itérant sur le tableau des résultats, ce tableau étant stocké dans *state.results*. Chaque résultat étant un objet avec plein de propriétés. Parmi ces propriétés on retrouve l'*url* du gif, un *titre* et un *identifiant* unique. Nous allons nous servir de ces informations pour créer une balise *<img>* pour chaque gif à afficher.

On utilise la méthode map qui permet de transformer chaque élément du tableau en un élément JSX, ici une balise *<img>*.

A chaque lettre saisie, une requête à Giphy est envoyée, et une liste d'images est affichée à la réponse du web service.

Nous avons une application qui fonctionne mais avec plusieurs problèmes : nous lançons des requêtes http superflues (il faudrait attendre la fin de la saisie pour lancer une recherche), et nous n'avons pas du tout découpé notre application en composants.

Voici comment améliorer le code :

- Créer un composant racine *App* qui gardera dans son état (*state*) les résultats de la recherche. App sera responsable d'appeler la méthode *fetchGifs* de notre API ;
- Créer un composant *SearchForm* qui s'occupera de proposer une saisie à l'utilisateur et à qui on passera une méthode à appeler chaque fois que la saisie change ;
- Créer un composant *Gifs* à qui on passera le tableau des résultats de recherche, et qui pour chaque résultat affichera un composant *Gif* ;
- Créer un composant *Gif* qui aura comme rendu un élément *<img>* ;
- Sortir la logique d'appel REST en dehors du composant, dans un fichier dédié (*src/api.js*).

Et nous allons ajouter quelques fonctionnalités à l'application :

- Pouvoir choisir le nombre de résultats à afficher (paramètre *limit* dans la requête) ;
- Ajouter du style grâce à une feuille CSS !

Et pour aller plus loin il faudra :

- Ajouter une image par défaut le temps que le gif soit téléchargé ;
- Interrompre la requête ajax à l'API si une nouvelle saisie est effectuée par l'utilisateur (et que la réponse n'est pas encore arrivée) ;
- Interrompre le téléchargement d'un gif si le composant Gif est supprimé avant la fin du téléchargement ;
- Ecrire les tests de chaque composant.

```
import React from 'react';
import './App.css';
import { SearchForm } from "./SearchForm";
import * as api from "./api";
import Gifs from "./Gifs";

class App extends React.Component {

  constructor(props) {
    super(props);
    this.state = { results: [] };

    this.fetch = this.fetch.bind(this);
  }

  fetch(query, limit) {
    api.fetchGifs(query, limit) // api.fetchGif est défini dans le fichier api.js
      .then(results => results && this.setState({ results: results })) // mise à jour du state
      .catch(err => console.log('fetch error ', err.message));
  }

  render() {
    return (
      <div className="App">
        <SearchForm onUpdate={this.fetch}/>
        <Gifs gifs={this.state.results}/>
      </div>
    );
  }
}

export default App;
```

App.js est épuré. Il n'a pas à savoir comment l'utilisateur effectue sa saisie, comment sont affichés les gifs et comment est construite la requête Ajax.

App.js crée un composant *SearchForm* en lui passant un attribut *onUpdate*. *onUpdate* pointe sur la méthode *fetch(query, limit)* du composant App. Cela signifie qu'une méthode *onUpdate* sera disponible dans le composant *SearchForm*, et que lorsque celle-ci sera appelée, c'est *fetch(query, limit)* qui sera exécutée.

C'est un concept clé dans React, les composants enfants se voient passer dans leurs *props* des valeurs et des méthodes. Les composants enfants peuvent appeler les méthodes qu'on leur passe ainsi. Ce n'est pas au composant *SearchForm* de savoir ce qu'il se passe lorsque la saisie de l'utilisateur change, il a juste à appeler la méthode qu'on lui passe. C'est une séparation des responsabilités intéressante car le code est bien découpé et notre composant *SearchForm* peut être réutilisé ailleurs dans le code pour d'autres traitements que de lancer une requête Ajax.

```
import React from 'react';

export class SearchForm extends React.Component {

  constructor(props) {
    super(props);
    this.state = { query: '', limit: 2 }; // état par défaut

    this.updateQuery = this.updateQuery.bind(this);
    this.updateLimit = this.updateLimit.bind(this);
    this.update = this.update.bind(this);
  }

  updateQuery(e) {
    // mise à jour de 'query' dans le state
    this.setState({ query: e.target.value }, this.onUpdate);
  }

  updateLimit(e) {
    // mise à jour de 'limit' dans le state
    this.setState({ limit: e.target.value }, this.onUpdate);
  }

  update() {
    // on appelle la méthode onUpdate, passée par le parent et accessible dans this.props.onUpdate
    this.props.onUpdate(this.state.query, this.state.limit);
  }

  render() {
    return <form>
      <label>Recherche : <input type="text" value={this.state.query} onChange={this.updateQuery}/></label>
      <label>Nombre de resultats : <input type="range" min={1} max={25} value={this.state.limit}
                                          onChange={this.updateLimit}/></label>
    </form>;
  }
}
```

On se sert d'un champ *<input type="text">* pour la saisie de la recherche et d'un *<input type="range">* pour choisir le nombre de résultats. Chaque fois qu'un des inputs change, on enregistre la nouvelle valeur dans l'état, et on prévient le parent que la saisie a été modifiée.

```
import React from 'react';
import Gif from "./Gif";

export default class Gifs extends  React.Component {
  render() {
    return <div className="gifs">
      {this.props.gifs.map(gif => <Gif gif={gif} key={gif.id}/>)}
    </div>
  }
}
```

Gifs est un composant simple qui prend le tableau de résultats dans ses *props*, et va afficher une liste de composants *Gif*.

Un composant sans état (sans *state*, aussi appelé *stateless*) et qui n'a qu'une méthode *render()*, peut être écrit sous forme d'une fonction. Cette fonction prend comme paramètres les *props* et retourne du JSX.

Nous allons réécrire le composant *Gifs* sous forme de fonction.

```
import React from 'react';
import Gif from "./Gif";

export default (props) => (
    <div className="gifs">
      {props.gifs.map(gif => <Gif gif={gif} key={gif.id}/>)}
    </div>
)
```

```
import axios from "axios/index";
import apiKey from "./api-key";

export const fetchGifs = (query, limit) => {
  return axios
    .get('http://api.giphy.com/v1/gifs/search', {
      params: {
        api_key: apiKey,
        limit: limit,
        q: query
      }
    })
    .then(response => response.data)
    .then(giphydata => giphydata.data)
    .catch(err => console.log(err));
};
```

Dans cette première version du fichier *api.js*, l'annulation de la requête n'est pas encore gérée. Le fichier *api-key.js* contient simplement la ligne : *export default 'my-api-key';*

A retrouver sur le repository Github du tutoriel

Tester ses composants de façon unitaire est primordial dans le développement d'applications React. Les librairies les plus utilisées sont *Jest* et *Enzyme*.

Pour gérer l'annulation de la requête vers Giphy, il faut utiliser le *cancel token* d'Axios. Ce n'est pas trop complexe mais cela rajoute un peu trop de code pour le montrer ici.

Afin d'interrompre les chargements d'images si le composant est supprimé (par exemple après une nouvelle saisie de l'utilisateur), on se base sur l'élément html *Image*.