Chapitre 3
Créer ses premiers composants

1. La déclaration d'un premier composant

Les composants sont des briques réutilisables destinées à être affichées et qui permettent d'ordonner le code. Ils évitent ainsi la duplication de code, séparent les différentes fonctions et fonctionnalités, et permettent d'uniformiser l'interface utilisateur. Ils sont incontournables lors de la création d'applications en React (il est impossible de maintenir un code en React sans en utiliser). Toutefois le concept de composant dépasse de très loin, dans le domaine du développement front-end, le cadre du développement en React, et est utilisé dans de nombreuses bibliothèques front-end (Angular, SolidJS, Svelte, etc.). Ces connaissances peuvent donc ouvrir à l'apprentissage d'autres *fraweworks* et bibliothèques qui se basent sur ce concept.

React a connu de nombreuses évolutions depuis ses débuts. L'une des évolutions les plus conséquentes est la disparition progressive des composants en classe au profit des composants fonctionnels. Il convient de garder à l'esprit que certaines entreprises, lors des entretiens, peuvent poser des questions sur ce sujet quand bien même les composants en classe tombent en désuétude. Il existe en effet de nombreuses entreprises ayant une base de code en React avec des composants en classe qui migrent vers des composants fonctionnels, et qui, pour ce faire, ont besoin de développeurs qui sont familiers avec les deux syntaxes.

Avec la syntaxe fonctionnelle, créer un composant en React n'a jamais été aussi simple. Pour ce faire, il suffit de créer un nouveau fichier et y déclarer une fonction dont le nom est par convention en *PascalCase*. Il n'existe pas de conventions strictes pour les noms de fichiers concernant les composants. En principe, on utilisera la *snake-case*, mais la *PascalCase* est aussi souvent utilisée. Le plus important est de conserver la cohérence sur les normes de nommages choisies.

La fonction du composant, en règle générale, va retourner un bloc de code HTML, bien que d'autres types de valeurs puissent être retournées (`null`, `string`, `number`, `boolean`, etc.). Pour appeler un composant, il faut l'importer dans le fichier où on souhaite l'afficher, puis utiliser une syntaxe de balise HTML classique : le nom du composant doit être entouré de chevrons comme une balise orpheline. Déclarer un composant `MyComponent` signifie qu'à l'usage on obtiendra `<MyComponent />`. Ce composant est un composant sans props ni états internes qui ne fait qu'afficher les mêmes données. L'appeler ne fera donc qu'injecter la valeur de retour de `MyComponent` dans le composant parent. Appeler plusieurs fois de suite `<MyComponent />` injectera donc plusieurs fois la valeur de retour du composant.

Il nous faut aussi brièvement évoquer la notion de parentalité. En effet, a été évoquée la notion de composant «parent» et de composant « enfant ». La relation de parentalité s'établit lorsqu'on appelle un composant dans un autre. Si `ComponentA` est invoqué dans `ComponentB`, `ComponentA` est l'enfant de `ComponentB`. La relation de parentalité est inversée si `ComponentB` est appelé dans `ComponentA`.

```
// components/greetings.tsx

export function Greetings() {
  return (
    <div>
      <p>Hello world !</p>
    </div>
  );
}

// app.tsx
import { Greetings } from "./greetings";
```

```
export default function App() {
  return (
    <div>
      <Greetings />
      <Greetings />
    </div>
  );
}
```

2. Le transfert de données entre composants

Pour l'instant, ce composant n'est pas modulable : il affichera toujours la même chose. C'est ici qu'intervient le concept de prop. Pour que notre composant devienne modulable et qu'il n'affiche pas toujours la même valeur nous avons besoin d'y injecter des données. Par exemple, nous pourrions décider que notre composant affiche un nom d'utilisateur ainsi que son adresse électronique. Ces données sont ce qu'on appelle des props (propriétés). En matière de syntaxe à l'usage, elles sont très similaires aux attributs des balises HTML.

La philosophie de React repose sur une approche unidirectionnelle. C'est-à-dire que les données transitent des composants parents vers les composants enfants, mais il n'est pas possible de faire transiter des données des composants enfants vers les composants parents, seulement de manière indirecte via des fonctions de rappel. Cette approche distingue React d'autres bibliothèques, notamment Angular, qui, lui, se base sur du transfert de données bidirectionnel. Cela n'a pas d'impact en termes de performances, mais encourage certains patrons de développements au détriment d'autres.

2.1 La déclaration des props

Pour injecter des props dans un composant, la première étape est de typer notre composant en déclarant ses props avec le mot-clé TypeScript `type` ou avec le mot-clé `interface`. La documentation de TypeScript recommande l'usage du mot-clé `interface`, mais il est courant de retrouver les deux manières de déclarer dans le code des applications professionnelles. De même, les conventions de nommage varient d'un projet à l'autre. Mais ici, les types seront toujours déclarés en répétant le nom du composant accolé au mot `Props`, ce qui, pour `MyComponent`, donne `MyComponentProps`. Une fois le type déclaré, il doit être passé en paramètre à la fonction du composant et peut être utilisé comme un objet JavaScript.

Au sein du composant, les props ne peuvent pas être modifiées (comme nous l'avons dit, les données ne transitent que des parents vers les enfants), elles sont immuables. Bien que cette règle soit implicite, il est possible de l'expliciter via le système de types TypeScript en ajoutant le type utilitaire `Readonly<T>`, indiquant clairement l'immutabilité de l'objet représentant les props du composant. L'utilisation de ce type utilitaire est très souvent omise bien qu'elle offre davantage de clarté et de sécurité (tenter de modifier une prop avec ce type utilitaire déclenche une erreur TypeScript).

```
type Email = '${string}@${string}';

interface UserCardProps {
  name: string;
  email: Email;
}
export function UserCard(props: Readonly<UserCardProps>) {
  return (
    <div>
      <h3>{props.name}</h3>
      <p><em>{props.email}</em></p>
    </div>
  );
}
```

Remarque

Déstructurer ou ne pas déstructurer, telle est la question. Les développeurs JavaScript expérimentés savent qu'il est possible, ici, de déstructurer l'objet passé en paramètre. De même que d'autres choses étudiées plus haut, il s'agit de conventions et de préférences, le plus important étant que toute l'équipe de développement applique de manière consistante les conventions choisies. Une syntaxe déstructurée donnerait la ligne :

```
export function UserCard({ name, email }: Readonly<UserCardProps>)
```

Remarque

Ne pas déstructurer les props des composants permet de distinguer ce qui est injecté depuis l'extérieur (les props) des variables internes du composant.

2.2 L'utilisation des props

Pour utiliser le composant `UserCard`, il faudra l'importer dans le composant parent, puis l'appeler comme une balise orpheline. Néanmoins, le code ne devrait pas fonctionner car il est désormais nécessaire de renseigner les props. Pour ce faire, la syntaxe élémentaire est assez similaire à celle des attributs des balises HTML : `prop={value}`. Le même composant peut alors être appelé avec des valeurs différentes, nous permettant ainsi d'obtenir différents rendus et d'améliorer considérablement la réutilisabilité des composants.

```
import { UserCard } from "./user-card";

export default function App() {
  return (
    <div>
      <UserCard
        name={"John Doe"}
        email={"john.doe@company.com"}
      />
      <UserCard
        name="Jane Smith"
        email="jane.smith@company.com"
      />
    </div>
  );
}
```

Si l'exemple ci-dessus peut sembler contenir une coquille au niveau de la manière dont les props sont affectées (avec ou sans accolades), elle nous permet d'introduire une autre particularité de l'affectation des props. La syntaxe de base pour attribuer des props est celle étudiée plus haut. Toutefois, ce n'est pas l'unique syntaxe possible. L'exemple précédent permet de démontrer qu'avec le type `string` il est possible de se passer des accolades. Le type `boolean` offre une autre particularité syntaxique intéressante, puisque si la valeur est `true`, il est possible de se passer d'écrire `prop={true}`, `prop` suffit. Soit une prop de type `isOnline: boolean`, à l'usage, cela donnera la syntaxe : `<UserCard isOnline />`.

2.3 Les props facultatives et les valeurs par défaut

Il peut être rapidement redondant de devoir écrire une à une toutes les props d'un composant, particulièrement si certaines d'entre elles ont souvent la même valeur. L'utilisation du symbole `?` (point d'interrogation) sur un attribut de l'interface représentant les props du composant permet de pallier cette redondance. `name?: string;` rend la prop `name` facultative. Ne pas renseigner `name` à l'usage du composant ne provoquera pas d'erreur.

De la même manière, renseigner une valeur par défaut peut éviter des redondances. Les valeurs par défaut ne sont, par principe, applicables qu'aux props facultatives (les autres devant obligatoirement être renseignées à l'usage du composant). Il existe deux syntaxes pour affecter des valeurs par défaut, selon qu'on utilise la déstructuration de props ou non.

```
interface UserCardProps {
  ...
  isOnline?: boolean;
}

// Without destructuration
export function UserCard(props: Readonly<UserCardProps>) {
  const isOnline = props.isOnline ?? false;

  return ...;
}

// With destructuration
```

```
export function UserCard({ name, email, isOnline = false }:
Readonly<UserCardProps>) {
  return ...;
}
```

Remarque

Il faut toutefois se méfier de la surutilisation des propriétés facultatives. L'utilisation d'une propriété facultative ne doit pas altérer le fonctionnement normal du composant. Une pratique courante et pourtant discutable consiste à ne pas afficher le contenu du composant si une propriété n'est pas renseignée. Une telle pratique brise le principe selon lequel un composant ne doit avoir qu'une responsabilité (Single Responsibility Principle), puisqu'il doit désormais gérer à la fois sa responsabilité initiale et son état d'affichage. Voici à quoi ressemble cette pratique le plus souvent :

```
export function Component(props: Readonly<ComponentProps>) {
  if (!props.name) return null;

  return ...;
}
```

2.4 L'affichage conditionnel

Il peut être assez commun de vouloir afficher ou ne pas afficher certains éléments JSX selon certaines conditions. Par exemple, on pourrait vouloir afficher une pastille verte pour indiquer qu'un utilisateur est connecté, et, à l'inverse, ne pas l'afficher s'il n'est pas connecté. Pour injecter des valeurs au milieu du code HTML, nous avons vu qu'il était possible d'utiliser les accolades. Ces accolades peuvent en réalité être utilisées pour injecter toutes sortes de code JavaScript.

Ainsi, l'opérateur booléen `&&` permet d'afficher un élément JSX si la valeur passée à gauche est *truthy* (n'importe quelle valeur sauf `false`, `undefined`, `null`, `NaN`, `0`, `""` et quelques autres). La valeur passée à droite doit être l'élément JSX à afficher. Il suffit ensuite d'entourer l'expression booléenne d'accolades. Il est bien sûr possible de mettre plusieurs conditions ou valeurs à gauche, tant que l'on pense à indiquer la priorité avec des paramètres sur l'élément JSX à afficher.

Remarque

Bien qu'il soit très peu connu et très peu utilisé, l'opérateur || permet, au contraire, d'afficher une valeur si la valeur passée à gauche est falsy.

Bien que certains développeurs l'ignorent, le JSX n'invente rien ici, et se contente de reprendre le fonctionnement des opérateurs booléens tels qu'ils existent en JavaScript comme dans de nombreux autres langages de programmation. Le principe de l'opérateur `&&` est d'exécuter du code jusqu'à rencontrer une valeur *falsy*. À l'inverse, le principe de l'opérateur `||` est d'exécuter du code jusqu'à la première valeur truthy. Cela peut se vérifier très facilement en JavaScript :

```
// exec will never be called.

true || exec();
false && exec();
```

C'est le même principe qui est appliqué en JSX, considérant qu'une valeur de type élément JSX sera toujours truthy.

```
interface UserCardProps {
  name: string;
  email: string;
  isOnline?: boolean;
}
export function UserCard(props: Readonly<UserCardProps>) {
  return (
    <div>
      {props.isOnline && <OnlineUserIcon />}
      <h3>{props.name}</h3>
      <p><em>{props.email}</em></p>
    </div>
  );
}
```

Il existe à l'opérateur `&&` une limite bien connue sur laquelle trébuche tout développeur React à un moment ou à un autre. En effet, toutes les valeurs falsy ne produisent pas le même rendu. Si `undefined` ou `false` n'affichent rien, tel n'est pas le cas de `0` ou `""` (chaîne de caractères vide). Ces valeurs sont considérées renderables (affichables) par JSX, et seront donc affichées, alors que le but le plus souvent visé par le développeur est de ne rien afficher (y compris l'affichage d'une chaîne de caractères vide est problématique car elle

rajoute des nœuds vides au DOM).

La solution à ce problème est l'usage de la fonction de conversion `Boolean` ou la conversion en booléen via la double négation `!!` :

```
interface UserCounterProps {
  count: number;
}
export function UserCounter(props: Readonly<UserCounterProps>) {
  return (
    <div>
      {Boolean(props.count) && <p>{props.count} user(s)
is/are online !</p>}
      {/* or... */}
      {!!props.count && <p>{props.count} user(s) is/are online !</p>}
    </div>
  );
}
```

L'opérateur ternaire est une autre option pour l'affichage conditionnel. Il est particulièrement bienvenu pour le cas où des éléments JSX doivent être affichés dans tous les cas, que la valeur soit truthy ou non. Il s'utilise de la même manière que l'opérateur ternaire en JavaScript, mais les deux valeurs de retour seront des éléments JSX :

```
<div>
  {props.isOnline ? <OnlineUserIcon /> : <OfflineUserIcon />}
</div>
```