**Qu'est-ce que le développement front-End ?**

Le développement frontal fait référence à ce que l'utilisateur final (aussi communément appelé le « client ») peut voir. Dans les formes les plus basiques, le développement frontal se compose de HTML, CSS et JavaScript.

En tant que développeur, vous constaterez qu'il est très facile pour votre front-end (site Web, application Web, etc.) de devenir très complexe et de comporter de nombreuses pièces mobiles différentes. Cela rend la résolution de problèmes beaucoup plus difficile lorsque vous devez parcourir un labyrinthe de code pour trouver le problème.

Finalement, les développeurs ont décidé qu'il devait y avoir une meilleure façon de gérer tout ce code, ils ont donc créé des bibliothèques qui pourraient faciliter la vie. React était l'une de ces bibliothèques.

React a été développé par : Facebook.

**Pourquoi React ?**

React est l'une des bibliothèques JavaScript les plus populaires pour les applications Web frontales.

Voici quelques-uns des avantages de React :

Vitesse :

Les sites Web interactifs doivent mettre à jour le DOM (Document Object Model) chaque fois qu'un changement se produit. Ce processus est généralement ingénieux et lent.

Par rapport aux autres bibliothèques qui manipulent le DOM, React utilise un DOM virtuel, permettant de mettre à jour uniquement les parties du site Web qui ont changé. Cela augmente considérablement la vitesse des mises à jour, car les applications Web modernes peuvent contenir des milliers d'éléments.

Nous en apprendrons plus sur le DOM virtuel dans les leçons à venir.

Facilité d'utilisation :

React permet aux développeurs de regrouper du code connexe, ce qui rend la création et la maintenance d'applications à grande échelle beaucoup moins complexes.

Support

React a une communauté incroyablement large et est open source. Il est maintenu par Facebook et la communauté. Commençons et plongeons dans la création de notre première application React !

**Que signifie DOM ?** Modèle d'objet de document

**Ajout de React :**

React peut être ajouté à un site Web sans outils ni installations spéciaux. Tout d'abord, nous devons ajouter la bibliothèque React sous la forme de deux balises de script à l'en-tête de notre document HTML :

<script src="https://unpkg.com/react@16/umd/react.development.js" crossorigin></script>   
  
<script src="https://unpkg.com/react-dom@16/umd/react-dom.development.js" crossorigin></script>

**Ajout de React :**

Où dans le document HTML devez-vous placer les balises de script ? **Dans le head**

**Ajout de React :**

Après avoir ajouté les balises de script requises, nous pouvons commencer à créer notre application React!

Nous ajoutons un conteneur, qui sera utilisé pour afficher quelque chose en utilisant React.

<div id="container"></div>

HTML

Vous pouvez utiliser n'importe quel identifiant pour votre conteneur. Il sera utilisé par React pour trouver le conteneur et y ajouter du contenu.

Maintenant, il est temps pour notre premier code React! Affichons un message simple sous forme d'en-tête :

<script type="text/babel">

ReactDOM.render(

<h1>Hello, React! </h1>, document.getElementById(‘container’)

)</script>

Le code trouve le conteneur div et y ajoute l'en-tête h1.

Ne vous inquiétez pas de la nouvelle syntaxe. Nous le couvrirons dans les leçons à venir.

**Ajout de React :**

Remplissez les blancs pour créer un élément div avec l'id "root".

<div id="root"></div>

**Créer une application React**

Dans la leçon précédente, nous avons appris à ajouter React à un document HTML simple à l'aide des balises de script.

Cependant, les vraies applications Web ont une échelle différente, contiennent plusieurs fichiers, utilisent des bibliothèques tierces, etc.

Facebook a créé un outil pratique appelé create react app qui facilite la configuration d'un projet React avec une simple commande !

Pour commencer, assurez-vous d'avoir une version récente de Node installée sur votre machine.

Exécutez les commandes suivantes dans le terminal pour créer et démarrer une application React appelée "my-app":

npx create-react-app mon-application

cd mon-application

npm start

Cela installera toutes les dépendances requises, configurera et démarrera le projet sur localhost:3000.

Ceci est la sortie par défaut de notre projet dans le navigateur.

Create React App nous permet de nous concentrer sur le code, plutôt que d'installer et de configurer différents outils.

**Create React App**

**Drag and drop the commands to create a new React app named "contacts" and start it.**

npx create-react-app contacts

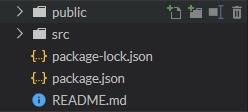
cd contacts

npm start

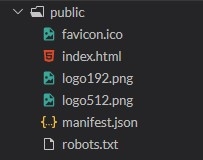
**Structure du projet**

Explorons la structure de notre projet en l'ouvrant à l'aide d'un éditeur de code.

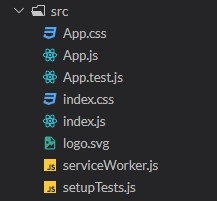
Nous utiliserons Visual Studio Code dans notre exemple, mais vous êtes libre d'utiliser n'importe quel éditeur de code.



Le dossier public contient des fichiers liés à la façon dont l'application s'affichera sur le client, le plus important d'entre eux étant index.html, qui est le modèle HTML de notre page :



Le dossier src contient tous les fichiers JavaScript, CSS et image qui seront compilés dans un fichier bundle et injectés dans index.html



Comment React est-il compilé dans un fichier bundle ? Il utilise ce qu'on appelle un "chargeur de fichiers". Dans le cas de create react app, Webpack est utilisé.

Webpack crée un fichier "bundle" contenant le contenu de plusieurs fichiers qui doivent être "regroupés" ensemble et tout est ajouté dans un seul fichier. Au lieu de faire en sorte que le fichier HTML recherche plusieurs fichiers, ce qui peut ralentir considérablement les temps de chargement, il n'a qu'à trouver un seul fichier.

N'oubliez pas que tous les fichiers CSS et JS doivent être ajoutés au dossier src, sinon Webpack ne les verra pas.

Bien qu'il existe d'autres fichiers dans le dossier src fournis avec create-react-app lors de sa génération, les deux fichiers ci-dessous sont les seuls fichiers critiques :

• index.js : Ce fichier est le point d'entrée dans notre application. Dans notre code, une méthode appelée ReactDOM.render() est utilisée pour trouver un élément avec id="root" dans le HTML et ajouter notre application React à l'intérieur de cet élément (similaire à la leçon précédente).

• App.js : ce fichier est le composant principal qui sera rendu au DOM, qui inclut actuellement l'image du logo React et le texte par défaut, que nous voyons dans la sortie.

Nous en apprendrons davantage sur les composants dans les leçons à venir.

**Project Structure**

**Drag and drop the files into their corresponding folders:**

public

index.html

src

index.js

index.css

**Modification de la sortie**

Maintenant que nous savons comment créer et exécuter un projet React, changeons la sortie par défaut en un simple message Hello.

Pour ce faire, nous devons ouvrir src/index.js et modifier le code comme suit :

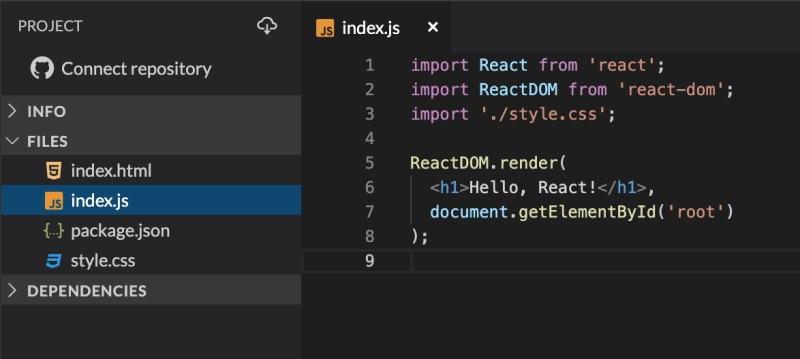
Une fonctionnalité vraiment intéressante de create-react-app est Fast Refresh, qui reflète automatiquement les modifications apportées au code dans le navigateur.

**Modification de la sortie**

Remplissez les blancs pour afficher "J'apprends React" dans l'élément avec id="test":

ReactDOM.render(<p>J'apprends React</p>, document.getElementById('test'));

**StackBlitz**

To make it easier to play around with React, we will be using **StackBlitz** as our online playground to enable changing and running real React code.  
  
Here is the the same project on StackBlitz:[**Try it on StackBlitz**](https://stackblitz.com/edit/hello-react-example?file)  
  
We have removed all the extra files, such as the logo images, to make the project structure simpler.  
Now we have the following files:  
**index.html**: The HTML page template.  
**index.js**: The entry point of our app.  
**style.css**: the stylesheet for our project.  
**package.json**: holds various metadata relevant to the project, like dependencies.

Tap **Try It on StackBlitz**and play around with the code!

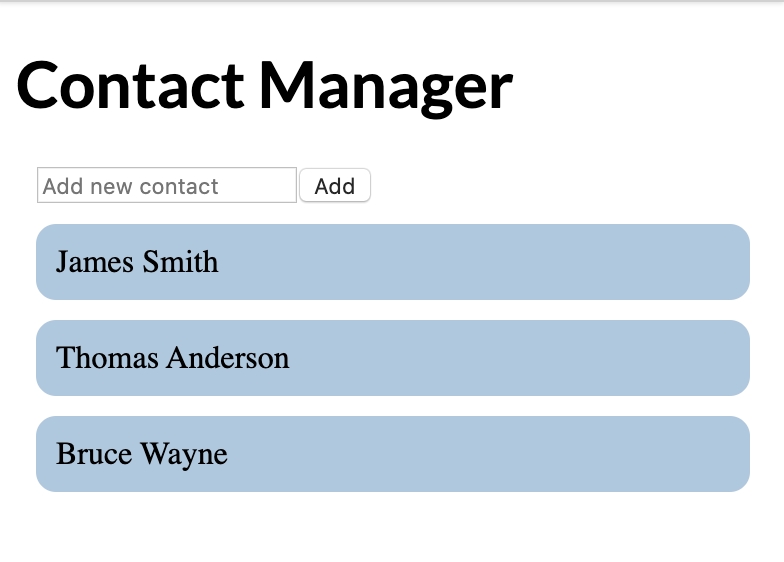
**StackBlitz**

Lequel des fichiers suivants représente le point d'entrée de l'application React? index.js

**Projet de cours**

Tout au long de ce cours, nous vous aiderons à pratiquer et à créer votre propre application Contact Manager à l'aide de React, afin que vous conserviez ce que vous avez appris et que vous puissiez l'utiliser.

Notre gestionnaire de contacts permettra de visualiser la liste des contacts et d'en ajouter de nouveaux à la liste.



N'ayez pas peur du code du projet. Au moment où vous aurez terminé le cours, tout aura un sens complet et semblera très faisable.

Nous le garantissons !

**Module 1 Quiz**

**Which of the following bundling tools does Create React App use?**

Webpack

**Which folder does the index.html file live in?**

public

**ReactDOM.render is which of the following:**

Used to add the React application inside the element

**To see the list of project dependencies, we need to go to which file?**

package.json

**Which of the following is the term that refers to your computer?**

localhost

**Complete the code to output "Hey there" in the element with id="root":**

ReactDOM.render(<h1>Hey there</h1>,

document.getElementById(”root”));

**Qu'est-ce que JSX ?**

Dans le module précédent, nous avons utilisé le code suivant pour afficher une sortie à l'aide de React :

ReactDOM.render(   
<h1>Hello, React!</h1>,   
document.getElementById('root')   
);

Essayez-le sur StackBlitz

Commençons à décomposer le code et à en comprendre chaque partie.

Nous allons commencer par l'élément <h1>Hello, React !</h1>.

Comme vous pouvez le voir, l'élément n'est pas entre guillemets pour représenter une chaîne. C'est comme un élément HTML, mais nous l'utilisons directement dans le code JavaScript !

C'est ce qu'on appelle JSX, et c'est une extension de syntaxe à JavaScript. Cela nous permet de créer des éléments d'interface utilisateur directement dans le code JavaScript !

React ne nécessite pas l'utilisation de JSX, cependant, il est courant dans la communauté React d'utiliser JSX car il facilite le développement d'interfaces utilisateur et permet à React d'afficher des messages d'erreur et d'avertissement utiles.

# What is JSX?

# Rearrange the blocks to create a valid HTML element with the text "This is some content".

**Introduction à JSX**

Reprenons notre code :

ReactDOM.render(   
<h1>Hello, React!</h1>,   
document.getElementById('root')   
);

Le code appelle la méthode de rendu de React et lui transmet deux arguments, un élément JSX et un conteneur. La méthode render affiche l'élément fourni dans le conteneur, qui, dans notre cas, est l'élément HTML avec id="root".

Essayez-le sur StackBlitz

Lorsque vous appelez la méthode de rendu, tout contenu existant du conteneur est remplacé. C'est pourquoi, généralement, les conteneurs sont vides dans le HTML.

**Expressions dans JSX**

Nous pouvons utiliser n'importe quelle expression JavaScript dans JSX en utilisant des accolades.

**For example:**

const name = "David";   
const el = <p>Hello, **{name}**</p>;   
  
ReactDOM.render(   
el,   
document.getElementById('root')   
);

Essayez-le sur StackBlitz

Dans l'exemple ci-dessus, nous utilisons le nom de la variable dans l'élément JSX.

Comme vous pouvez le voir, JSX peut être utilisé comme des expressions JavaScript. Vous pouvez affecter une expression JSX à une variable, la renvoyer à partir d'une fonction, etc.

Expressions dans JSX

Avec quoi les expressions JavaScript doivent-elles être enveloppées pour les ajouter au JSX ?

Curly braces

**Attributs dans JSX**

Nous pouvons spécifier des attributs à l'aide de guillemets, comme en HTML :

<div **id="name"**></div>

Lorsque vous utilisez une expression JavaScript comme valeur d'attribut, les guillemets ne doivent pas être utilisés :

<div **id={user.id}**></div>

React DOM utilise des propriétés camelCase au lieu des noms d'attributs HTML. Par exemple, class devient className dans JSX.

Remplissez les blancs pour utiliser la valeur de "x" comme nom de classe pour l'élément <div>.

**const x ={myClass} ;**

**const el = <div className={x}></div>**

**Comment fonctionne JSX ?**

Lorsque les expressions JSX sont compilées, elles sont converties en objets JavaScript, représentant des éléments React.

React utilise ensuite ces éléments pour construire le DOM HTML correspondant et l'afficher dans le navigateur.

Créons une application de compteur, qui incrémente une variable de compteur toutes les secondes et l'affiche sur la page sous forme de paragraphe :

let counter = 0;   
  
function show() {   
counter++;   
const el = <p>{counter}</p>;   
ReactDOM.render(   
el, document.getElementById('root')   
);   
}   
  
setInterval(show, 1000);

Essayez-le sur StackBlitz

Nous utilisons setInterval pour appeler la fonction show toutes les secondes et rendre l'élément counter sur la page.

L'une des grandes fonctionnalités de React est qu'il ne met à jour que les éléments qui nécessitent une mise à jour. Vous pouvez inspecter la sortie HTML de l'exemple ci-dessus et voir que seul le texte du paragraphe est mis à jour toutes les secondes.

En pratique, la plupart des applications React appellent ReactDOM.render() une fois.

Nous apprendrons comment mettre à jour des éléments sans appeler la méthode render dans les prochaines leçons.

Remplissez les blancs pour appeler la fonction "calculer" toutes les 5 secondes.

setInrerval(calculate, 5000)

**DOM virtuel**

Nous avons appris dans la partie précédente que React ne met à jour que les éléments nécessaires.

Cela permet aux applications React d'être beaucoup plus rapides que les applications construites avec d'autres technologies frontales.

Mais comment React y parvient-il ?

**React utilise un DOM virtuel, qui est une représentation légère du DOM.**

**Lorsqu'un élément est modifié, il est d'abord mis à jour dans le DOM virtuel. Ce processus est rapide, car le DOM virtuel est représenté par des objets simples.**

**Après cela, React compare le DOM virtuel à son state précédent et n'applique que les mises à jour du DOM nécessaires pour amener le DOM à state souhaité.**

**DOM signifie Document Object Model et est une arborescence.**

**Virtual DOM**

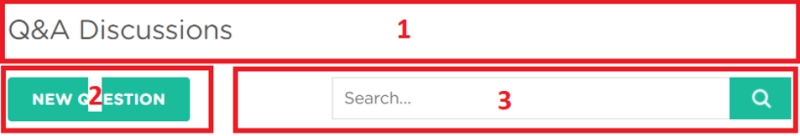
**DOM represents the page as a:**

Tree

**Composants**

Les composants vous permettent de diviser la page en parties indépendantes et réutilisables.

Visualisons cela en jetant un coup d'œil à un élément de la page Discussions SoloLearn :



Notez que la page peut être divisée en plusieurs parties. Chacune de ces "parties" est un composant.

Le titre est un composant, le bouton "nouvelle question" est un composant et la barre de recherche est son propre composant.

Cela facilite l'organisation des sauts et des limites de la page, mais plus important encore, les composants nous permettent, en tant que développeurs, de séparer les préoccupations les unes des autres.

La séparation des préoccupations est un principe de programmation qui stipule que chaque préoccupation doit être séparée en éléments individuels.

Par exemple, dans le diagramme ci-dessus, le bouton "nouvelle question" (2) doit être cliqué si un utilisateur souhaite ajouter une nouvelle question, tandis que la barre de recherche (3) serait utilisée si l'utilisateur souhaite rechercher les questions existantes.

Lequel des éléments suivants est un principe de conception pour diviser un programme informatique en sections distinctes afin de traiter chaque composant individuellement ?

Separation of Concerns

**Functional Components**

Dans React, vous pouvez utiliser deux types de composants : les composants fonctionnels et les composants de classe.

Dans cette partie, nous parlerons des composants fonctionnels.

Un composant fonctionnel est une simple fonction JavaScript :

function Hello() {   
return <h1>Hello world.</h1>;   
}

Le code ci-dessus a défini un composant fonctionnel appelé Hello, qui renvoie un simple élément React.

Notez que le nom du composant fonctionnel commence par une lettre majuscule. C'est absolument essentiel. Si nous commençons le nom d'un composant par une lettre minuscule, le navigateur traitera notre composant comme un élément HTML normal au lieu d'un composant.

**Functional Components**

**Fill in the blanks to make this a usable functional component, that returns a div element:**

 HelloWorld() 

 <div>Hello, world.</div>;

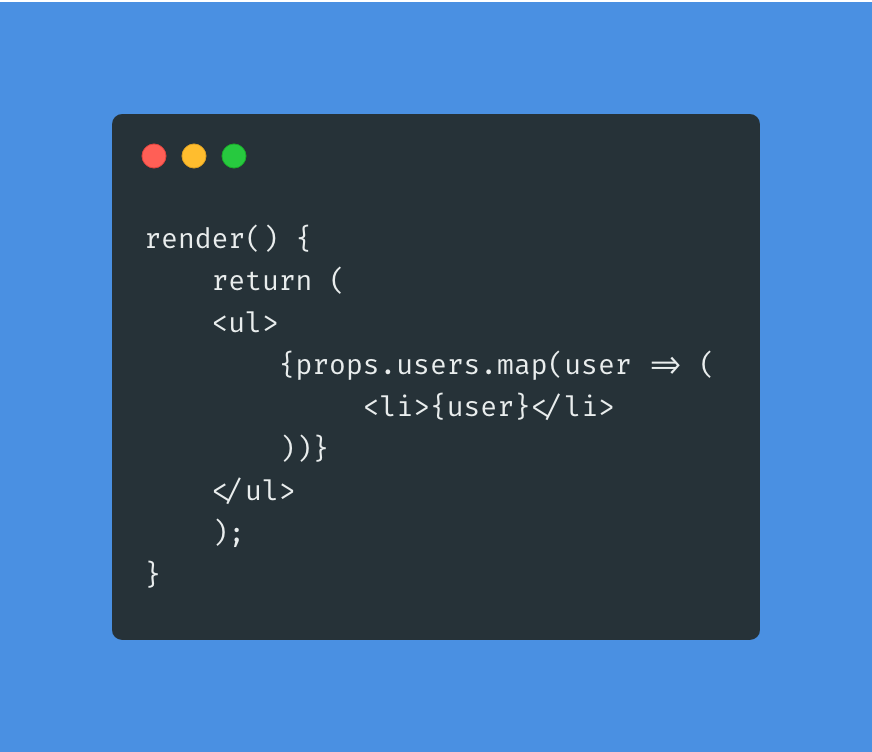
}

# Comprendre les types de composants react js

# https://media-exp1.licdn.com/dms/image/C4D12AQHkhdya6PuuMw/article-cover_image-shrink_720_1280/0/1568106028089?e=1649289600&v=beta&t=Xi0uoBM0nd0Ig5UqUuGkldKeQMpyrVr38Nk7sQ2IwKY

Si vous travaillez avec React depuis un moment, vous devrez admettre qu’il est difficile de créer proprement des applications réactives. Au fur et à mesure que de nouveaux composants sont ajoutés, l’application devient de plus en plus complexe et vous finirez par en arriver à un point où vous devrez prendre des décisions importantes, telles que l'emplacement de vos données et la gestion de l'état. L’implémentation des mauvaises décisions rendra votre application plus susceptible des erreurs, vous aurez également un code difficile à lire et à maintenir. Je vais vous présenter les differents types de composants utilisés dans l’organisation des applications React.

**1- Presentational components(Composants de présentation)**



Les composants de présentation sont les composants dont le seul travail consiste à rendre une vue. Essentiellement, ils ne contiennent aucune logique métier. C'est pourquoi ils sont parfois appelés "dumb components". Cela signifie qu'ils n'ont pas d'accès direct à Redux ou à d'autres sources de données. Les données leur sont transmises via des **props**.

**2- Container Component(Composant conteneur)**



Il convient de garder à l’esprit que les composants du conteneur et ceux de la présentation vont de pair. En fait, vous pouvez les considérer comme faisant partie du même modèle de conception. Lorsque les composants de présentation ne gèrent pas l'état, les composants de conteneur le font. Lorsque les composants de présentation sont généralement des "enfants" subordonnés dans une hiérarchie de composants, les composants de conteneur sont presque toujours les "parents" des composants de présentation.

**3-  Functional Component(Composant fonctionnel)**



Les composants fonctionnels sont que des fonctions. La seule contrainte pour un composant fonctionnel est d'accepter les propriétés en tant qu'argument et de renvoyer un fichier JSX valide, BASTA!

**Rendering Components**

Pour afficher le composant, nous devons créer l'élément JSX correspondant.

Par exemple, pour notre composant défini par l'utilisateur Hello :

const el = <Hello />;

Maintenant, nous pouvons utiliser notre élément défini par l'utilisateur et le rendre sur la page :

function Hello() {   
return <h1>Hello world.</h1>;   
}   
  
const el = <Hello />;   
ReactDOM.render(   
el,   
document.getElementById('root')   
);

Remember, all component names need to start with a capital letter.

# Rendering Components: Fill in the blanks to define a component called Section and render it on the page.

Section() {

return <div>A section</div>;

}

const el = <Section />;

ReactDOM.

(, document.getElementById('root'));

# Class Components

Les composants de classe sont généralement utilisés lorsqu'il existe des interactions utilisateur plus avancées, telles que des formulaires et des animations.

Tous les composants de classe doivent étendre la classe React.Component.

Nous pouvons réécrire notre composant fonctionnel Hello en tant que composant de classe :

class Hello extends React.Component {   
render() {   
return <h1>Hello world.</h1>;   
}   
}

Essayez-le sur StackBlitz

Les composants de classe doivent avoir une méthode de rendu, qui est chargée de dire ce que la page doit afficher. Nous découvrirons les caractéristiques et les différences des composants fonctionnels et de classe dans les prochaines leçons.

**Class Components: Fill in the blanks to complete the class component:**

Greeting extends React.Component {

() {

return <div>Hello World.</div>;

}

**Props**

Les composants fonctionnels peuvent accepter des arguments, similaires aux fonctions JavaScript. Ces arguments sont appelés props et représentent un objet.

Par exemple, nous pouvons utiliser des props dans notre composant Hello :

function Hello(**props**) {   
return <p>Hello, **{props.name}**!</p>;   
}

Maintenant, nous pouvons ajouter un attribut name à notre élément :

const el = <Hello **name="David"** />;

La valeur de l'attribut sera transmise au composant lors du rendu.

Essayez-le sur StackBlitz

Un élément peut avoir plusieurs attributs personnalisés, qui seront transmis au composant à l'aide de l'objet props. Vous pouvez utiliser n'importe quel nom personnalisé pour vos attributs.

**Props**

**What is the output of this code? function Test(props) { return <p>{"A"+props.title}</p>; } const el = <Test title="X" />; ReactDOM.render( el, document.getElementById('root') );**

X

**Components using Components**

Les composants peuvent utiliser d'autres composants pour générer une sortie.

**For example:**

function App() {   
return <div>   
<Hello name="David" />   
<Hello name="James" />   
<Hello name="Amy" />   
</div>;   
}

Ici, notre composant App utilise le composant Hello trois fois, chaque fois avec un nouvel attribut de nom. Essayez-le sur StackBlitz

En règle générale, il est recommandé de diviser les composants complexes en plusieurs composants plus petits, réutilisables.

Par exemple, un composant Post peut utiliser un composant Avatar, un composant Image, un composant Date, etc.

**Components using Components**

**You can use a component in other components multiple times.** True

**Props in Class Components**

Props sont accessibles dans les composants de classe à l'aide de this.props.

**For example:**

class Hello extends React.Component {   
render() {   
return <p>Hello, {**this.props.name**}!</p>;   
}   
}

Essayez-le sur StackBlitz

Une chose importante à considérer est que les accessoires sont en lecture seule, ce qui signifie que les composants ne peuvent pas modifier leurs accessoires.

Les applications interactives doivent généralement modifier les données et les éléments de la page.

Nous apprendrons comment modifier la sortie des composants à l'aide de l'état dans la prochaine leçon.

**Fill in the blanks to access the url property of the props object in the class component:**

class Image extends React.Component {

render() {

return <img src={

.props.} />;

}}

**Un exemple**

Maintenant que nous savons comment créer des composants et leur transmettre des données, créons une liste de courses.

Chaque article de notre liste aura un nom et un prix.

Par example:

<Item name="Cheese" price="4.99" />

Le composant Item rendra un simple élément div avec les données :

function Item(props) {   
return <div className="item">   
<b>Name:</b> {props.name} <br />   
<b>Price:</b> {props.price}   
</div>;   
}

Nous pouvons maintenant utiliser notre composant et créer plusieurs éléments pour notre liste de courses :

<Item name="Cheese" price="4.99" />   
<Item name="Bread" price="1.5" />   
<Item name="Ice cream" price="24" />

Essayez-le sur StackBlitz

Nous avons ajouté quelques styles CSS simples pour séparer visuellement les éléments. Appuyez sur Essayez-le sur StackBlitz pour voir le résultat et jouer avec le code.

**How did we apply CSS styles to the Item elements?**

using a CSS class

**State**

Jusqu'à présent, nous avons appris à transmettre des données aux composants à l'aide props.

De nombreuses applications Web ont besoin de leurs composants pour modifier leurs données, par exemple après une interaction de l'utilisateur (clic sur un bouton, soumission d'un formulaire, etc.).

Cependant, props ne peuvent pas être modifiés.

Afin de permettre aux composants de gérer et de modifier leurs données, React fournit une fonctionnalité appelée state.

State est un objet qui est ajouté en tant que propriété dans les composants de classe.

Par example:

class Hello extends React.Component {   
**state = {   
name: "James"   
}**  
render() {   
return <h1>Hello {**this.state.name**}.</h1>;   
}   
}

Essayez-le sur StackBlitz

Comme vous pouvez le voir, l'état n'est qu'un simple objet, qui contient des paires clé:valeur.

Comme pour les props, les valeurs sont accessibles à l'aide de this.state.

Maintenant, lorsque le composant est rendu, state est initialisé avec la valeur donnée et il y aura un en-tête indiquant "Hello James.".

Essayez-le sur StackBlitz L'objet d'état peut contenir plusieurs paires clé: valeur, séparées par des virgules.

**Fill in the blanks to initialize the state of the component with name and age values.**

class Person  React.Component {

= {

name: "David",

: 42

}

...

}

**Changing State**

State ne doit pas être modifié directement. Au lieu de cela, React fournit une méthode setState(), qui peut être utilisée pour modifier state.

**Par example:**

**this.setState**({   
name: "James",   
age: 25   
});

Vous devez transmettre un objet avec les nouvelles paires clé: valeur à la méthode setState.

Pourquoi devrions-nous utiliser setState, au lieu de simplement changer directement les valeurs des propriétés de l'objet ?

La réponse révèle l'une des fonctionnalités les plus utiles de React : lorsque setState est appelé, React restitue automatiquement le composant affecté avec le nouvel état !

Habituellement, le changement de state se produit dans les gestionnaires d'événements. Nous allons voir un exemple dans la partie suivante !

Lorsque le state change à l'aide de la méthode setState, React est informé et restitue immédiatement le composant avec state mis à jour.

**Drag and drop to change the value of the "count" property in the state to 42.**

this.setState({

count: 42

});

**Application compteur**

Pour mieux comprendre le fonctionnement de state, créons une application de compteur, qui incrémente le compteur à chaque fois qu'un bouton est cliqué.

Nous commençons par créer notre composant Counter, qui comprend le compteur et un bouton :

class Counter extends React.Component {   
state = {   
counter: 0   
}   
render() {   
return <div>   
<p>{this.state.counter}</p>   
<button>Increment</button>   
</div>;   
}   
}

Nous avons initialisé notre compteur à la valeur 0 dans l'état.

Maintenant, nous devons ajouter un gestionnaire d'événements click au bouton et incrémenter le compteur dans l'état.

Voici le code final:

class Counter extends React.Component {   
state = {   
counter: 0   
}   
increment = () => {   
**this.setState({   
counter: this.state.counter+1});**  
}   
render() {   
return <div>   
<p>{this.state.counter}</p>   
<button **onClick={this.increment}**>Increment</button>   
</div>;   
}   
}

Essayez-le sur StackBlitz

L'événement onClick appelle la fonction d'incrémentation de notre composant, qui utilise setState pour modifier la valeur de notre compteur. Lorsque state est modifié, React déclenche automatiquement un nouveau rendu du composant.

Appuyez sur Essayez-le sur StackBlitz pour voir le compteur en action !

Notez que le gestionnaire d'événements utilise la syntaxe camelCase et que la fonction du gestionnaire est passée entre accolades.

Nous aborderons les gestionnaires d'événements plus en détail dans les prochaines leçons.

**Counter App**

**When a component's state is changed, the whole DOM of the page is reloaded.**

False

**Props vs State**

En guise de récapitulatif, voici un résumé des principales différences entre **Props** et **State** :

- Nous utilisons **Props** pour transmettre des données aux composants.

- Les composants utilisent state pour gérer leurs données.

- **Props** sont en lecture seule et ne peuvent pas être modifiés.

- **State** peut être modifié par son composant à l'aide de la méthode **setState()**.

- La méthode **setState()** entraîne un nouveau rendu du composant affecté.

Les composants qui ont un **STATE** sont appelés avec **STATEFUL**, tandis que les composants qui n'utilisent pas de **STATE** sont appelés sans **STATELESS**.

**Props vs State**

**The Person component receives a "name" attribute using props. Fill in the blanks to use that value as the initial value for the "name" in the state.**

class Person extends React.Component {

state = {

name: ..name

}}

**Hooks**

La version antérieure de React permettait d'utiliser le state uniquement avec les composants de classe.

Dans les récentes itérations de React, une nouvelle fonctionnalité appelée Hooks a été introduite, permettant d'utiliser le state à l'intérieur des composants fonctionnels.

Tout d'abord, nous devons importer le crochet useState :

import React, **{ useState }** from 'react';

useState renvoie une paire, la valeur de state actuel et une fonction, qui vous permet de modifier le state.

useState prend un argument, qui est la valeur initiale de state.

Regardons un exemple :

function Hello() {   
**const [name, setName] = useState("David");**  
  
return <h1>Hello {name}.</h1>;   
}

Essayez-le sur StackBlitz

Dans l'exemple ci-dessus, nous créons une variable de state name et une fonction setName. La syntaxe des crochets est appelée déstructuration de tableau. Il affecte la variable name à la valeur de state actuel, et setName à la fonction qui permet de changer le state. Vous pouvez nommer ces variables comme bon vous semble.

Ensuite, nous passons "David" comme valeur initiale de notre variable name à useState().

Vous pouvez créer plusieurs variables de state avec leurs méthodes set correspondantes. Utilisez simplement des instructions séparées pour chaque variable à l'aide du hook useState.

**Hooks**

**Fill in the blanks to define a state variable called "count" with an initial value of 42:**

const [, setCount = useState();

**Counter App using Hooks**

Nous pouvons maintenant réécrire notre application Counter de la leçon précédente en utilisant un composant fonctionnel et des hooks !

**Here is the code:**

function Counter() {   
const [counter, setCounter] = useState(0);   
  
function increment() {   
setCounter(counter+1);   
}   
  
return <div>   
<p>{counter}</p>   
<button onClick={increment}>   
Increment   
</button>   
</div>;   
}

Essayez-le sur StackBlitz

Comme vous pouvez le voir, comparé au composant de classe, le code est beaucoup plus court et plus facile à lire et à comprendre. C'est l'une des raisons pour lesquelles l'équipe React a créé Hooks.

N'oubliez pas que les crochets ne peuvent être utilisés qu'à l'intérieur des composants fonctionnels.

# Les hooks sont des fonctions qui permettent de "s'accrocher" aux fonctionnalités de React à partir des composants de la fonction.

# Counter App using Hooks

# The useState() hook can be used in class components: False

**Lifecycle Methods**

React fournit des méthodes de cycle de vie spéciales pour les composants de classe, qui sont appelées lorsque les composants sont montés, mis à jour ou démontés.

**Le montage est le processus lorsqu'un composant est rendu sur la page.**

**Le démontage est le processus lorsqu'un composant est supprimé de la page.**

**La méthode componentDidMount est appelée lorsqu'un composant est rendu sur la page.**

Par exemple, nous pouvons utiliser componentDidMount dans notre application Counter pour définir la valeur initiale du compteur :

componentDidMount() {   
this.setState({counter: 42});   
}

Essayez-le sur StackBlitz

Cela définira une valeur initiale du compteur lorsque le composant sera rendu.

componentDidMount est généralement utilisé pour remplir le state à l'intérieur d'un composant lors de son montage initial sur le DOM.

De même, la méthode de cycle de vie componentWillUnmount() est appelée juste avant que le composant ne soit supprimé du DOM. Il peut être utilisé pour libérer des ressources prises par le composant.

**Lifecycle Methods**

**Given the code below, what will be the value of x after <div><Test /><Test /></div> has been rendered on the page? let x = 0; class Test extends React.Component { componentDidMount() { x+=3; } render() { return null; } }**

6

**componentDidUpdate**

Une autre méthode de cycle de vie est componentDidUpdate(), qui est appelée lorsqu'un composant est mis à jour dans le DOM.

On peut par exemple alerter la valeur courante du compteur lorsqu'elle est incrémentée :

componentDidUpdate() {   
alert("Number of clicks: " + this.state.counter);   
}

[**Try it on StackBlitz**](https://stackblitz.com/edit/react-state-counter-update-example?file)

**componentDidUpdate()** is only called when the component is updated.

**Drag and drop to create a component that logs the value of the "name" state variable when it gets updated.**

class

Test extends React.Component {

state = {

name: "Test"

}

componentDidUpdate() {

console.log (this.state.name);

}

...

}

**The useEffect Hook**

Les méthodes de cycle de vie que nous avons couvertes ne sont disponibles que pour les composants de classe.

Cependant, React fournit un crochet spécial appelé useEffect pour rendre les méthodes de cycle de vie disponibles dans les composants fonctionnels. Il combine les méthodes componentDidMount, componentDidUpdate et componentWillUnmount en une seule.

Par exemple, nous pouvons obtenir le comportement de notre dernier exemple en utilisant un composant fonctionnel Counter :

function Counter() {   
const [counter, setCounter] = useState(0);   
  
**useEffect(() => {   
alert("Number of clicks: " + counter);   
});**  
  
function increment() {   
setCounter(counter+1);   
}   
return <div>   
<p>{counter}</p>   
<button onClick={increment}>Increment</button>   
</div>;   
}

Essayez-le sur StackBlitz

Lorsque vous exécutez le code, vous remarquerez que la boîte de dialogue d'alerte apparaît également lors du premier rendu. Cela est dû au fait que, par défaut, useEffect s'exécute à la fois, après le premier rendu et après chaque mise à jour.

Pour appeler la méthode uniquement lorsque quelque chose change, nous pouvons lui fournir un deuxième argument :

useEffect(() => {   
//do something   
}, **[count]**);

Désormais, la méthode useEffect() ne s'exécutera que si le nombre change.

Pour imiter componentWillUnmount, useEffect peut renvoyer une fonction qui nettoie après :

useEffect(() => {   
// do something   
  
**return () => {   
// cleanup   
};**   
});

Vous pouvez avoir plusieurs effets dans le même composant.

Tout comme avec le crochet useState, nous devons importer useEffect pour pouvoir l'utiliser : import React, {useState, useEffect} from 'react';

**The useEffect Hook**

**Fill in the blanks to import the required hooks to be able to use state and mimic lifecycle methods in functional component.**

importReact, { useState, useEffect} from 'react';

**Event Handling**

La gestion des événements dans React est très similaire à la gestion des événements dans le DOM.

La seule différence est que les noms d'événements utilisent la syntaxe CamelCase et que le gestionnaire d'événements doit être passé entre accolades.

Par exemple, pour gérer l'événement click sur un bouton :

<button **onClick={handleClick}**>   
My Button   
</button>

Cliquer sur le bouton appellera la fonction handleClick du composant. Explorons notre application Counter:

function Counter() {   
const [counter, setCounter] = useState(0);   
  
function increment() {   
setCounter(counter+1);   
}   
return <div>   
<p>{counter}</p>   
<button **onClick={increment}**>Increment</button>   
</div>;   
}

Essayez-le sur StackBlitz

L'événement onClick appelle la fonction d'incrémentation, qui incrémente la variable d'état du compteur.

Découvrez la même application Counter créée à l'aide d'un composant de classe ici.

**Event Handling**

**What will be the text on the button after it is clicked 3 times? function Toggle() { const [val, setVal] = useState("ON"); function toggle() { setVal((val=="ON")?"OFF":"ON"); } return <button onClick={toggle}>{val}</button>; }**

OFF

**Handling User Input**

L'une des façons courantes dont les utilisateurs interagissent avec les pages Web consiste à utiliser des champs de texte.

Nous pouvons gérer la saisie de l'utilisateur dans React en utilisant l'événement onChange du champ de texte.

Lorsque la valeur du champ de texte change, Event Handler est appelé, mettant à jour la valeur du champ dans l'état du composant.

De cette façon, vous avez toujours la valeur réelle du champ de texte dans l'état.

Créons une application pour convertir des kilomètres en miles. Nous prendrons la valeur Km à partir d'un champ de texte et calculerons la valeur en miles lors de la saisie :

function Converter() {   
const [km, setKm] = useState(0);   
  
function handleChange(e) {   
setKm(e.target.value);   
}   
function convert(km) {   
return (km/1.609).toFixed(2);   
}   
  
return <div>   
<input type="text" value={km}   
**onChange={handleChange}** />   
<p> {km} km is {convert(km)} miles </p>   
</div>;   
}

Essayez-le sur StackBlitz

Notre composant Converter comprend un champ de texte, qui appelle la fonction handleChange lorsque sa valeur change.

La fonction handleChange met à jour le state avec la valeur actuelle du champ de texte, provoquant un nouveau rendu du composant et affichant la valeur en miles correspondante, qui est calculée à l'aide de la fonction de convert.

La valeur du champ de texte est accessible via l'objet e, qui représente l'événement React. Il est transmis à la fonction de Event Handler en tant qu'argument et peut être utilisé pour accéder à l'objet event.

**Handling User Input**

**Fill in the blanks to call the "doSomething()" function when the value of the textfield changes.**

<input type="text" value={val}

onChange={doSomething} />

**Forms**

In the previous part, we learned how to handle user input in text fields. Text fields are usually part of a form. Similar to the previous example, React form elements keep their state and update it based on user input.  
This way you always have the data of your form at your disposal in the state.  
To demonstrate this, we will create a form, that will add numbers every time the form is submitted and display the sum.  
Our form contains an input field and a submit button:

function AddForm() {   
const [sum, setSum] = useState(0);   
const [num, setNum] = useState(0);   
  
function handleChange(e) {   
setNum(e.target.value);   
}   
  
function handleSubmit(e) {   
setSum(sum + Number(num));   
e.preventDefault();   
}   
  
return <form **onSubmit={handleSubmit}**>   
<input type="number" value={num} o**nChange={handleChange}** />   
<input type="submit" value="Add" />   
<p> Sum is {sum} </p>   
</form>;   
}

**JSX**

[**Try it on StackBlitz**](https://stackblitz.com/edit/react-simple-calc-form-example?file)  
  
In the code above, the value of the input is controlled by React (we keep the value in the state).  
When the form is submitted using the submit button, the handleSubmit function gets called, which updates the value of sum in the state.  
An input form element whose value is controlled by React in this way is called a "controlled component".

Notice the e.preventDefault(); statement. This statement prevents the default behavior of the form, which, by default, reloads the page when submitted. In JavaScript we would use return false; for that, but in React we need to call preventDefault().

**Forms : What do we use to store the form values?** state

**Lists**

Les applications Web contiennent généralement des éléments répétitifs, tels que des listes ou des sections, où le même élément DOM est répété avec un ensemble de données différent.

Prenons un tableau de chaînes : const arr = ["A", "B", "C"];

Nous devons rendre un élément de liste <li> pour chaque élément du tableau. Nous pouvons définir un composant MyList et lui transmettre le tableau en tant que prop en utilisant un attribut de données personnalisé : <MyList data={arr} />

Maintenant, lorsque le tableau est accessible via Props, nous pouvons écrire la logique du composant :

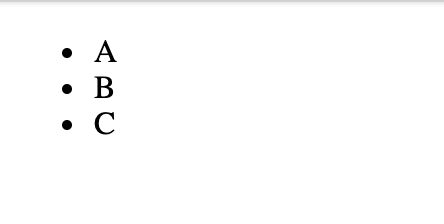
function MyList(props) {   
const arr = props.data;   
const listItems = arr.map((val) =>   
<li>{val}</li>   
);   
return <ul>{listItems}</ul>;   
}

Essayez-le sur StackBlitz

Nous prenons le tableau d'entrée props entrants, parcourons le tableau à l'aide de la fonction JavaScript map et renvoyons un élément <li> pour chaque élément.

Le tableau résultant est stocké dans la variable listItems.

Ensuite, le composant renvoie le tableau listItems à l'intérieur d'une balise <ul>.



Ce code génère un avertissement indiquant que chaque élément a besoin d'une clé unique. Nous examinerons les clés dans la partie suivante.

**Lists: What is the output of this code? const arr = [2, 3, 1, 7]; const res = arr.map(x => x \* 3); console.log(res[1]);** 

**Keys**

Chaque élément d'une liste doit avoir un attribut clé.

Les clés agissent comme une identité unique, identifiant chaque élément.

Il s'agit généralement d'identifiants issus de vos données ou d'index générés automatiquement.

**For example:**

const listItems = arr.map((val, index) =>   
<li **key={index}**>{val}</li>   
);

**JSX**

[**Try it on StackBlitz**](https://stackblitz.com/edit/react-simple-list-keys-example?file)

Les clés sont importantes, car elles identifient de manière unique les éléments, aidant React à comprendre quels éléments ont été modifiés, ajoutés ou supprimés.

**Keys**

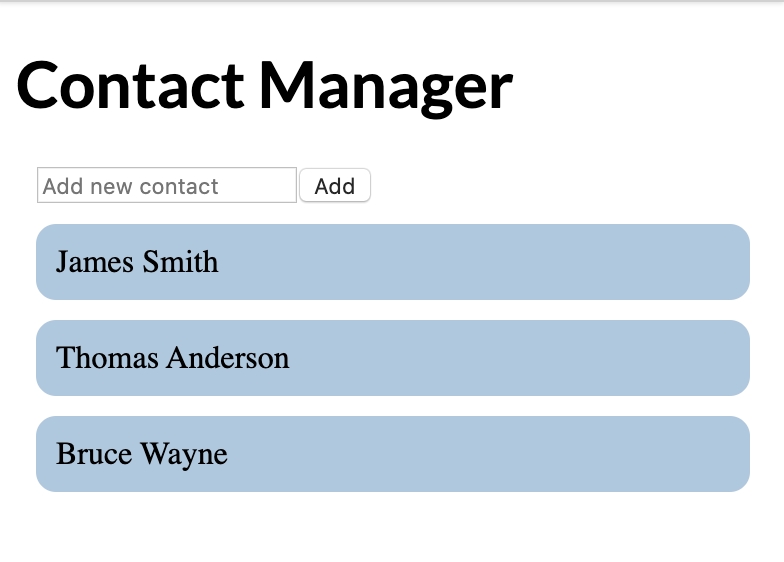
**The user object contains an "id" and a "name" property. Given an array of user objects called "arr", create a list of names, where each element's key is represented by the id.**

const users = arr.

((user) => <li key={user.}>

{.name} </li>);

**Contact Manager**

Now, that we know how to create components, pass them data using props and manage their data using state, let's start building our **Contact Manager**.  
  
Our Contact Manager will allow to view the list of contacts and add new ones to the list.By looking at the mockup, it makes sense to have two components:  
**AddPersonForm**: a form with the text field and Add button.  
**PeopleList**: a list of contacts.  
  
Let's create these components.  
**AddPersonForm** uses state to manage the value of the text field:

function AddPersonForm() {   
const [ person, setPerson ] = useState("");   
  
function handleChange(e) {   
setPerson(e.target.value);   
}   
  
function handleSubmit(e) {   
e.preventDefault();   
}   
return (   
<form onSubmit={handleSubmit}>   
<input type="text"   
placeholder="Add new contact"   
onChange={handleChange}   
value={person} />   
<button type="submit">Add</button>   
</form>   
);   
}

**JSX**

For now, we just prevent the default behavior when the form is submitted.  
  
**PeopleList** received an array representing the contacts and renders a list on the page:

function PeopleList(props) {   
const arr = props.data;   
const listItems = arr.map((val, index) =>   
<li key={index}>{val}</li>   
);   
return <ul>{listItems}</ul>;   
}

**JSX**

Now we can render our components on the page and include some initial data:

const contacts = ["James Smith", "Thomas Anderson", "Bruce Wayne"];   
  
const el = (   
<div>   
<AddPersonForm />   
<PeopleList data={contacts} />   
</div>   
);

**JSX**

[**Try it on StackBlitz**](https://stackblitz.com/edit/react-contact-manager-1?file)  
  
Tap **Try it on StackBlitz**to see the output.

Adding a new contact does not work, as we have not built the logic in the **handleSubmit** function yet.

**Contact Manager**

**Given an array of posts, create div elements for each post object.**

const posts = arr.((post) =>

< key={post.id}>

{post.text}

</>

);

**Sharing State**

Right now, our **AddPersonForm** independently keeps its state. How can we add a new contact to our **PeopleList** then, when the form is submitted?  
  
To accomplish that, we need to share the state between the components. We can do that by **lifting the state up** to a parent component. This means that the parent component will hold the data that needs to be shared between the components. In our case, that is the contacts list.  
  
Let's create a parent component called **ContactManager**, which includes the **AddPersonForm** and **PeopleList** as child components and holds the contacts list in its state:

function ContactManager(props) {   
const [contacts, setContacts] = useState(props.data);   
  
return (   
<div>   
<AddPersonForm />   
<PeopleList data={contacts} />   
</div>   
);   
}

**JSX**

[**Try it on StackBlitz**](https://stackblitz.com/edit/react-contact-manager-2?file)  
  
The ContactManager component receives the initial contacts list using props, saves it in its state.  
Then it passes down the contacts list to its child component.

Data can be passed from the parent to the child, but not from the child to the parent. React uses what is called **unidirectional data flow**, in other words, data only flows downward, so to speak.

**Sharing State**

**How can data flow in React?**

From parent component to child component

**Adding a Contact**

Now, we can create an **addPerson()** function to our ContactManager component to add a new person to our contacts state array:

function ContactManager(props) {   
const [contacts, setContacts] = useState(props.data);   
  
**function addPerson(name) {   
setContacts([...contacts, name]);   
}**  
...   
}

**JSX**

But how are we going to call this function from our child AddPersonForm component, where the data for the new person is stored?  
Just like we passed down data using props, React allows us to pass down function references!

function ContactManager(props) {   
const [contacts, setContacts] = useState(props.data);   
  
function addPerson(name) {   
setContacts([...contacts, name]);   
}   
  
return (   
<div>   
<AddPersonForm **handleSubmit={addPerson}** />   
<PeopleList data={contacts} />   
</div>   
);   
}

**JSX**

Similar to passing the contacts list to our PeopleList component, we passed down the **addPerson**() function to our AddPersonForm using a prop called **handleSubmit**.  
  
Now, our PeopleList can call the **handleSubmit** function that it received when the form is submitted, to add a new person to the list:

function AddPersonForm(props) {   
const [ person, setPerson ] = useState('');   
  
function handleChange(e) {   
setPerson(e.target.value);   
}   
  
**function handleSubmit(e) {   
props.handleSubmit(person);   
setPerson('');   
e.preventDefault();   
}**  
return (   
<form onSubmit={handleSubmit}>   
<input type="text"   
placeholder="Add new contact"   
onChange={handleChange}   
value={person} />   
<button type="submit">Add</button>   
</form>   
);   
}

**JSX**

[**Try it on StackBlitz**](https://stackblitz.com/edit/react-contact-manager-3?file)

We also clear the value of the text field using **setPerson('')** after adding a new person.

**Adding a Contact: The parent component uses the following code to pass down its "calculate()" function: <Child doStuff={calculate} /> How can the child component call that function:** props.doStuff()

Résumé

Un élément important à retenir de cette leçon est que les props peuvent être utilisés pour transmettre non seulement les state, mais aussi les fonctions, qui peuvent manipuler le state.

De cette façon, nous pouvons stocker state de l'application dans le parent et permettre à ses composants enfants d'utiliser et de manipuler le state.

Maintenant, lorsque notre application est entièrement fonctionnelle, nous pouvons ajouter des styles CSS et une vérification pour empêcher la création de contacts vides.

Essayez-le sur StackBlitz

Explorez le projet, essayez de modifier le code et ajoutez des attributs à nos contacts, tels que l'âge, l'emplacement, le numéro de téléphone.

Personnalisez votre liste de contacts et partagez votre création dans les commentaires ci-dessous !

**Summary: Which of the following is used to pass down information from parent component to child component?** Props

**Module 2 Quiz**

**12 Comments**

**The pattern that allows developers to break different functionalities into smaller pieces is called what?**

Separation of Concerns

**Fill in the blanks to complete the class component:**

class HelloWorld React.Component {

() {

return <div>Hello world.</div>;

} }

**useEffect can combine which of the following lifecycle methods?**

componentDidUpdate

componentWillUnmount

componentDidMount

**Drag and drop to create a valid controlled input field.**

function Compose(props) {

const [text, setText] = useState(props.text);

function handleChange(e) {

setText(e.target.value);

}

return <input type="text"

value={text}

onChange={handleChange} />;

}

**Fill in the blanks to create a functional component Search, which sets the initial value of the text field in the state using the received props.**

Search(props) {

const [val, setVal] = useState(.q);



<div>

<input type="text" value={val} />

<button>Search</button>

</div>;

}

**State Management**

Dans le module précédent, nous avons créé une application Contact Manager, qui stocke le state dans le composant parent et le transmet aux composants enfants correspondants.

Dans le monde réel avec des applications à plus grande échelle, cela peut devenir beaucoup plus complexe. Nous devrons peut-être transmettre les données de state à plusieurs niveaux pour accéder au composant souhaité.

Le fait de devoir transmettre des données à plusieurs niveaux de composants imbriqués rendrait difficile la compréhension de la cause d'un changement de state, car plusieurs composants potentiels pourraient modifier le state.

Cela entraînerait également beaucoup de code redondant, ce qui rendrait difficile la maintenance et le débogage du code.

Appuyez sur Continuer pour voir comment nous pouvons surmonter ces défis.

**State Management**

**How would you pass down data from the parent component to its child?**

Using props