ECOLE COMMUNAUTAIRE DE L'ENSEIGNEMENT SUPERIEUR

FORMATEUR : LEVI GOTENI

ECES

REACT JS

I. Introduction à React JS

1. Qu'est-ce que React ?
2. Principes de base de React (Virtual DOM, composants, etc.)
3. Pourquoi utiliser React ?
4. Avantages de React par rapport à d'autres frameworks ou bibliothèques
5. Cas d'utilisation courants de React
6. Historique de React
7. Origine et évolution de React
8. Versions majeures et principales fonctionnalités introduites dans chaque version

II. INSTALLATION DE L'ENVIRONNEMENT DE DEVELOPPEMENT

- Node.js et npm
- Installation de Node.js et npm
- Explication de npm et de son rôle dans le développement React

1- Création d'un projet React

- Utilisation de Create React App pour créer un nouveau projet
- Options de configuration disponibles lors de la création d'un projet

2- Structure d'un projet React

- Vue d'ensemble des fichiers et répertoires générés par Create React App
- Explication des principaux fichiers (index.html, index.js, etc.)

III. COMPOSANTS REACT

1- Qu'est-ce qu'un composant ?

- Définition d'un composant dans le contexte de React
- Importance des composants dans le développement React

2- Création de composants

- Syntaxe pour créer des composants fonctionnels et des composants de classe
- Exemples de création de différents types de composants

3- **Les composants fonctionnels vs. les composants de classe**
   - Avantages et inconvénients des deux approches
   - Recommandations pour choisir entre les deux types de composants
4- **Utilisation des props**
   - Explication des props et de leur utilisation pour passer des données entre les composants
   - Exemples d'utilisation de props dans différents scénarios

VI. ÉTAT ET CYCLE DE VIE DES COMPOSANTS

1- **Gestion de l'état dans React**
   - Concept d'état dans React et son importance
   - Méthodes pour gérer l'état dans les composants de classe et les composants fonctionnels avec des Hooks
2- **Cycle de vie d'un composant de classe**
   - Présentation des différentes phases du cycle de vie d'un composant de classe
   - Utilisation des méthodes de cycle de vie pour exécuter du code à des moments spécifiques

3- **Hooks et cycle de vie des composants fonctionnels**
   - Introduction aux Hooks et à leur rôle dans la gestion du cycle de vie des composants fonctionnels
   - Utilisation des principaux Hooks (useState, useEffect, etc.)

V. RENDU CONDITIONNEL ET GESTION DES EVENEMENTS

1- **Rendu conditionnel avec JavaScript**
   - Utilisation de JavaScript pour conditionner l'affichage de contenu dans les composants React
   - Exemples de rendu conditionnel pour différents scénarios

2- Gestion des événements dans React

- Syntaxe pour ajouter des gestionnaires d'événements aux éléments JSX
- Exemples de gestion d'événements courants (clic, changement, soumission de formulaire, etc.)

3- Utilisation des méthodes de classe vs. des fonctions fléchées pour la gestion des événements

- Comparaison des deux approches pour la liaison des événements dans les composants de classe et les composants fonctionnels
- Recommandations pour choisir la meilleure approche en fonction du contexte

VI. LISTES ET CLES

1- Rendu de listes

- Utilisation de la méthode map pour afficher dynamiquement des listes de données dans React
- Exemples de rendu de listes simples et complexes

2- Utilisation des clés pour optimiser le rendu des listes

- Explication de l'importance des clés dans le rendu des listes et leur impact sur les performances
- Bonnes pratiques pour le choix et l'utilisation des clés dans les listes React

VII. FORMULAIRES DANS REACT

1- Contrôle des composants de formulaire

- Utilisation de l'état pour contrôler les éléments de formulaire dans React
- Exemples de création de formulaires contrôlés

2- Gestion des événements de formulaire

- Capturer les événements de soumission et de changement dans les formulaires React
- Validation des données saisies par l'utilisateur avant la soumission

3- Validation de formulaire

- Méthodes pour valider les données de formulaire dans React
- Utilisation de bibliothèques tierces pour simplifier la validation des formulaires

VIII. COMMUNICATION ENTRE COMPOSANTS

1- **Transmission des données entre composants parent et enfants**
   - Utilisation des props pour transmettre des données des composants parents aux composants enfants
   - Exemples de communication unidirectionnelle des données
2- **Utilisation des callbacks pour la communication entre enfants et parent**
   - Mécanismes pour permettre aux composants enfants de communiquer avec leurs composants parents
   - Utilisation de callbacks pour transmettre des données et des événements
3- **Utilisation du contexte pour la communication entre composants éloigné**
   - Introduction au contexte React et son rôle dans la communication entre composants éloignés
   - Exemples d'utilisation du contexte pour partager des données entre différents composants

IX. INTRODUCTION AUX OUTILS ET BIBLIOTHEQUES COMPLEMENTAIRES

1- **React Router pour la navigation**
   - Présentation de React Router et de son rôle dans la navigation côté client dans les applications React
   - Configuration de routes et navigation entre différentes vues

2- **Axios pour les requêtes HTTP**
   - Utilisation d'Axios pour effectuer des requêtes HTTP dans les applications React
   - Exemples de requêtes GET, POST, PUT et DELETE avec Axios
3- **Styled Components pour la stylisation**
   - Introduction à Styled Components et son approche basée sur les composants pour la stylisation dans React
   - Création de composants stylisés et utilisation de fonctionnalités avancées telles que les thèmes et les animations

X. DEPLOIEMENT D'UNE APPLICATION REACT

Préparation de l'application pour la production
   - Optimisation des performances de l'application pour la production
   - Configuration des variables d'environnement et des paramètres de construction

1- **Déploiement sur différents services d'hébergement**

- Guide étape par étape pour le déploiement d'une application React sur des plateformes populaires telles que Netlify, Vercel, Heroku, etc.
- Configuration des options de déploiement et gestion des mises à jour de l

1. **Qu'est-ce que React ?**

   React est une bibliothèque JavaScript open-source utilisée pour construire des interfaces utilisateur interactives et dynamiques. Elle a été développée par Facebook et est largement utilisée dans l'industrie pour la création d'applications web modernes.

2. **Les principaux concepts de React sont les suivants :**

   - Composants : Les composants sont les blocs de construction fondamentaux de toute application React. Ils permettent de diviser l'interface utilisateur en morceaux réutilisables et autonomes, ce qui facilite la gestion et la maintenance du code.
   - Virtual DOM : React utilise un concept appelé Virtual DOM pour améliorer les performances en minimisant les manipulations du DOM réel. Le Virtual DOM est une représentation virtuelle légère de l'interface utilisateur qui est synchronisée avec le DOM réel de manière efficace.
   - Unidirectional Data Flow : React suit un modèle de flux de données unidirectionnel, ce qui signifie que les données circulent dans une seule direction à travers les composants de l'application. Cela rend le code plus prévisible et plus facile à comprendre.

POURQUOI UTILISER REACT ?

React présente plusieurs avantages qui en font un choix populaire pour le développement d'applications web :

- Performance : Grâce à son Virtual DOM et à son rendu efficace, React offre d'excellentes performances, même pour les applications complexes.

- Réutilisabilité du code : En divisant l'interface utilisateur en composants réutilisables, React favorise la réutilisabilité du code, ce qui permet de gagner du temps et de réduire la duplication.
- Facilité de maintenance : La modularité des composants et le flux de données unidirectionnel rendent le code plus facile à comprendre, à tester et à maintenir.

HISTORIQUE DE REACT

React a été initialement développé par Facebook en 2011 pour répondre aux besoins croissants de performance et de maintenabilité de leur interface utilisateur. Il a été open-source en 2013, permettant ainsi à la communauté des développeurs de contribuer à son développement.

Depuis lors, React a connu plusieurs versions majeures, introduisant de nouvelles fonctionnalités et améliorations à chaque étape. Certaines des versions les plus importantes incluent React 16 avec l'introduction des Hooks en 2018, et React 17 avec des améliorations de stabilité et de compatibilité en 2020.

II. INSTALLATION DE L'ENVIRONNEMENT DE DEVELOPPEMENT

- **Node.js et npm**

Avant de commencer à travailler avec React, vous devez installer Node.js et npm (Node Package Manager). Node.js est un environnement d'exécution JavaScript côté serveur, tandis que npm est un gestionnaire de packages qui vous permet d'installer, de partager et de gérer des dépendances pour vos projets JavaScript.

Pour installer Node.js et npm, suivez les étapes suivantes :

1. Rendez-vous sur le site officiel de Node.js à l'adresse nodejs.org.

2. Téléchargez la version recommandée de Node.js pour votre système d'exploitation (Windows, macOS, Linux).

3. Suivez les instructions d'installation fournies par le programme d'installation de Node.js.

4. Une fois l'installation terminée, vérifiez que Node.js et npm sont correctement installés en ouvrant votre terminal (ou invite de commande) et en exécutant les commandes suivantes :

- node -v
- npm -v

Cela affichera les versions de Node.js et npm installées sur votre système.

- **Création d'un projet React**

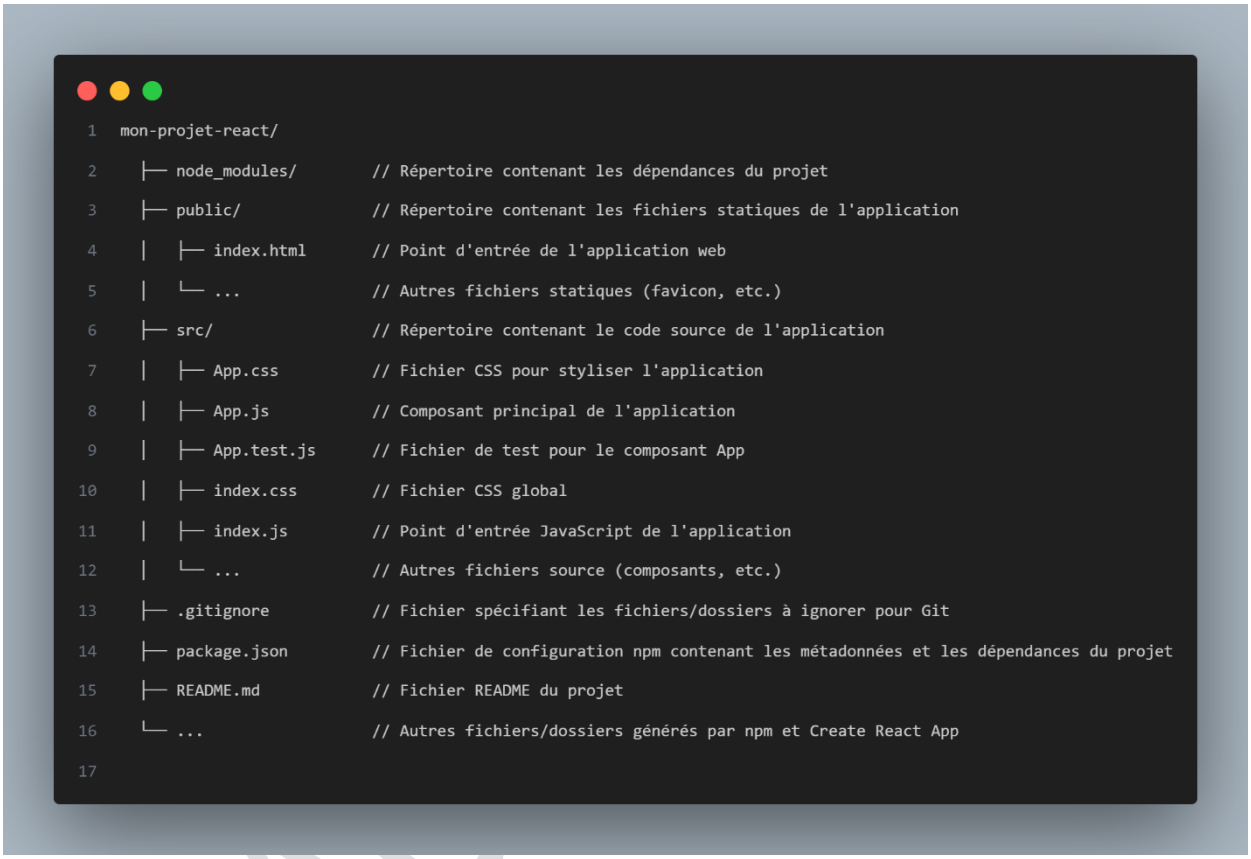
Une fois que Node.js et npm sont installés, vous pouvez créer un nouveau projet React en utilisant Create React App, un outil de démarrage officiel de React qui configure automatiquement votre projet avec les outils et les configurations recommandés.

Pour créer un nouveau projet React avec Create React App, ouvrez votre terminal et exécutez la commande suivante : **npx create-react-app mon-projet-react**

Cela créera un nouveau répertoire appelé « mon-projet-react » et installera tous les packages nécessaires pour votre projet React. Une fois l'installation terminée, accédez au répertoire de votre projet en utilisant la commande « cd mon-projet-react ».

- **Structure d'un projet React**

Une fois que vous êtes dans le répertoire de votre projet React, vous verrez une structure de dossier initiale générée par Create React App. Voici un aperçu de cette structure :



```
mon-projet-react/
  ├── node_modules/        // Répertoire contenant les dépendances du projet
  ├── public/              // Répertoire contenant les fichiers statiques de l'application
  │   ├── index.html       // Point d'entrée de l'application web
  │   └── ...              // Autres fichiers statiques (favicon, etc.)
  ├── src/                 // Répertoire contenant le code source de l'application
  │   ├── App.css          // Fichier CSS pour styliser l'application
  │   ├── App.js           // Composant principal de l'application
  │   ├── App.test.js      // Fichier de test pour le composant App
  │   ├── index.css        // Fichier CSS global
  │   ├── index.js         // Point d'entrée JavaScript de l'application
  │   └── ...              // Autres fichiers source (composants, etc.)
  ├── .gitignore           // Fichier spécifiant les fichiers/dossiers à ignorer pour Git
  ├── package.json         // Fichier de configuration npm contenant les métadonnées et les dépendances du projet
  ├── README.md            // Fichier README du projet
  └── ...                  // Autres fichiers/dossiers générés par npm et Create React App
```

La structure de votre projet React comprend principalement les répertoires « node_modules », « public » et « src », ainsi que quelques fichiers de configuration et de démarrage.

III. COMPOSANTS REACT

QU'EST-CE QU'UN COMPOSANT ?

Les composants sont les blocs de construction fondamentaux de toute application React. Ils permettent de diviser l'interface utilisateur en morceaux réutilisables et autonomes, ce qui facilite la gestion et la maintenance du code. En React, tout est un composant, que ce soit une petite boîte comme un bouton ou une grande boîte comme l'ensemble de l'application.

CREATION DE COMPOSANTS

En React, il existe deux types principaux de composants : les composants fonctionnels et les composants de classe. Nous allons nous concentrer sur les composants fonctionnels dans ce cours.

Un composant fonctionnel est une simple fonction JavaScript qui accepte des propriétés (props) en entrée et renvoie un élément React. Voici comment créer un composant fonctionnel de base :

```
import React from 'react';

// Définition d'un composant fonctionnel nommé MyComponent
function MyComponent() {
  return <div>Hello World!</div>;
}

export default MyComponent;

```

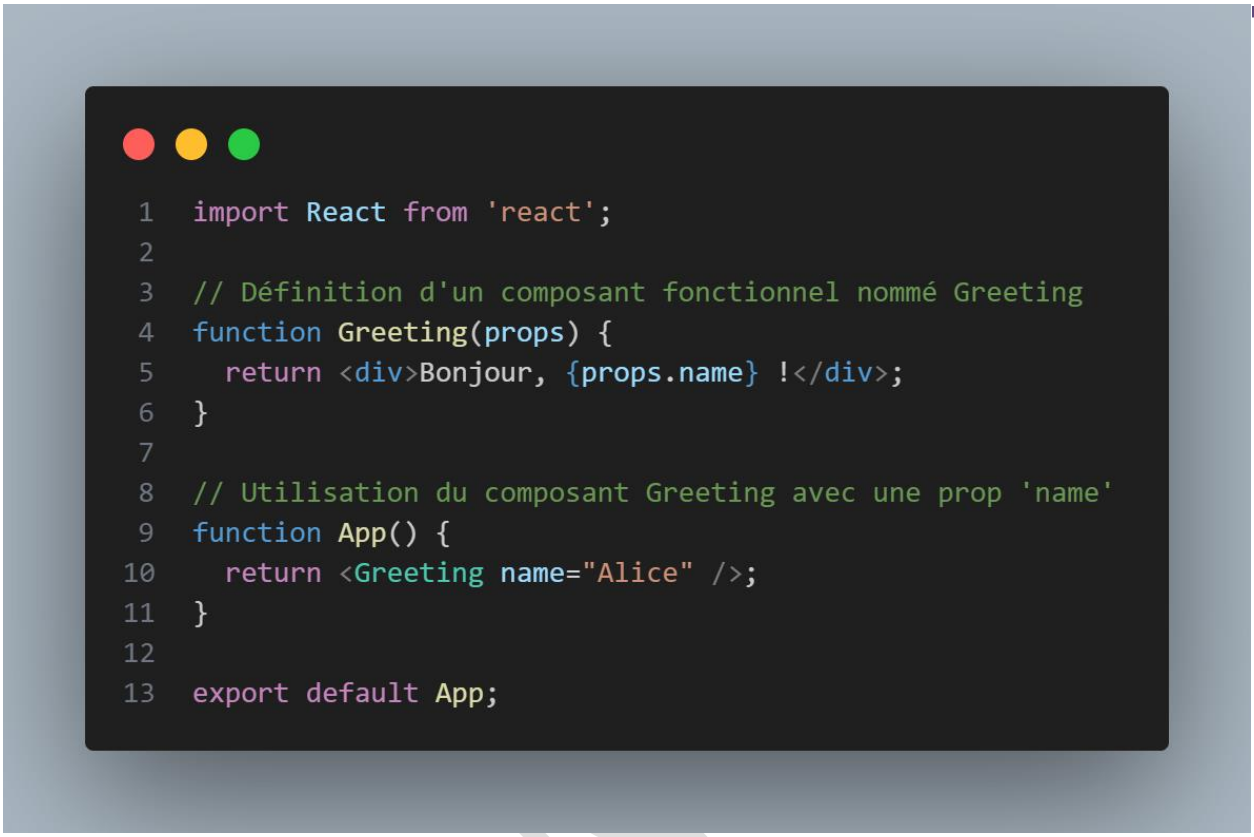
Dans cet exemple, nous avons créé un composant fonctionnel nommé « MyComponent » qui retourne un élément « » avec le texte « Hello World ! ».

LES COMPOSANTS FONCTIONNELS VS LES COMPOSANTS DE CLASSE

Les composants fonctionnels sont des fonctions JavaScript simples, tandis que les composants de classe sont des classes JavaScript qui étendent la classe de base « React.Component ». Historiquement, les composants de classe étaient la méthode principale pour créer des composants React, mais depuis l'introduction des Hooks dans React 16.8, les composants fonctionnels sont devenus plus populaires en raison de leur syntaxe plus concise et de leur meilleure prise en charge des fonctionnalités avancées.

UTILISATION DES PROPS

Les props (propriétés) sont des données passées de composant parent à composant enfant. Ils sont utilisés pour transmettre des données immuables de haut en bas dans la hiérarchie des composants. Voici comment utiliser les props dans un composant fonctionnel :



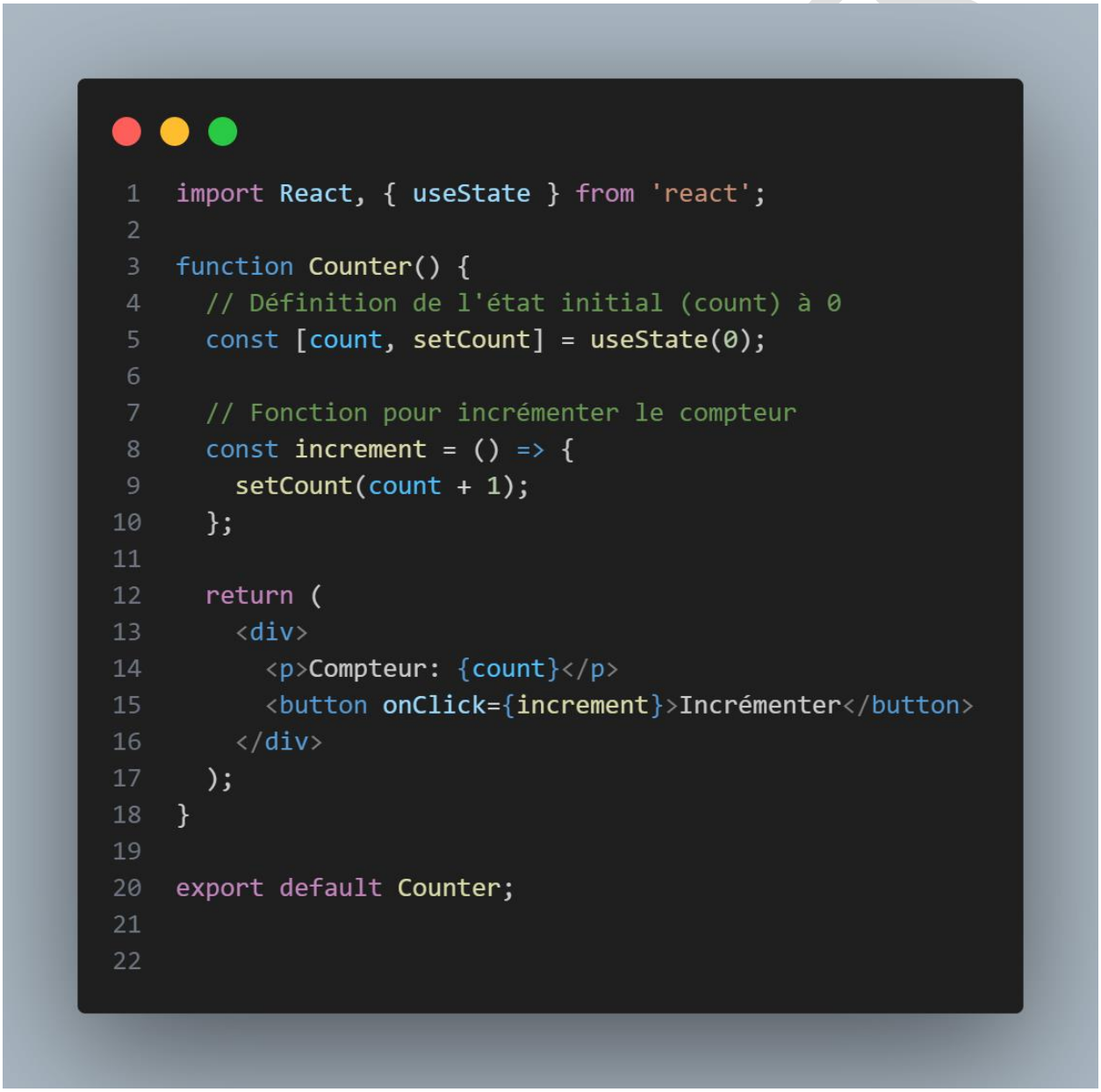
```react
import React from 'react';

// Définition d'un composant fonctionnel nommé Greeting
function Greeting(props) {
  return <div>Bonjour, {props.name} !</div>;
}

// Utilisation du composant Greeting avec une prop 'name'
function App() {
  return <Greeting name="Alice" />;
}

export default App;
```

Dans cet exemple, nous avons créé un composant fonctionnel « Greeting » qui utilise une prop « name » pour afficher un message de salutation personnalisé. Ensuite, nous avons utilisé ce composant « Greeting » dans le composant « App » en lui passant la prop « name » avec la valeur « Alice ».

Ce chapitre met en lumière les composants React, en mettant particulièrement l'accent sur les composants fonctionnels. Nous avons vu comment créer un composant fonctionnel, la différence entre les composants fonctionnels et les composants de classe, ainsi que l'utilisation des props pour transmettre des données entre les composants.

III. ÉTAT ET CYCLE DE VIE DES COMPOSANTS

L'état (state) est un concept fondamental en React qui permet aux composants de gérer et de maintenir des données qui changent au fil du temps. Contrairement aux props, l'état est interne à un composant et peut être modifié par le composant lui-même. Dans les composants fonctionnels, l'état est géré à l'aide de Hooks, en particulier avec le Hook « useState ».



```jsx
import React, { useState } from 'react';

function Counter() {
  // Définition de l'état initial (count) à 0
  const [count, setCount] = useState(0);

  // Fonction pour incrémenter le compteur
  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Compteur: {count}</p>
      <button onClick={increment}>Incrémenter</button>
    </div>
  );
}

export default Counter;
```

Dans cet exemple, nous avons créé un composant fonctionnel `Counter` qui utilise le Hook « useState » pour déclarer une variable d'état « count » avec une valeur initiale de 0. Nous avons également défini une fonction « increment » qui met à jour l'état du compteur lorsqu'un bouton est cliqué.

CYCLE DE VIE DES COMPOSANTS FONCTIONNELS

Avant l'introduction des Hooks dans React, les composants fonctionnels ne disposaient pas de cycle de vie. Cependant, avec l'introduction du Hook « useEffect », les composants fonctionnels peuvent désormais gérer les effets secondaires et simuler le cycle de vie des composants de classe.

Le Hook « useEffect » permet d'exécuter du code au moment du montage (lorsque le composant est affiché à l'écran), de la mise à jour (lorsque les props ou l'état changent) et du démontage (lorsque le composant est retiré de l'écran).

Voici un exemple d'utilisation du Hook « useEffect » pour simuler le cycle de vie « componentDidMount » dans un composant fonctionnel :

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

function Timer() {
  const [seconds, setSeconds] = useState(0);

  useEffect(() => {
    const intervalId = setInterval(() => {
      setSeconds(prevSeconds => prevSeconds + 1);
    }, 1000);

    // Nettoyage de l'intervalle lors du démontage du composant
    return () => clearInterval(intervalId);
  }, []); // Le tableau vide indique que l'effet ne dépend d'aucune variable et ne doit être exécuté qu'une seule fois

  return <div>Temps écoulé: {seconds} secondes</div>;
}

export default Timer;
```

Dans cet exemple, nous avons créé un composant fonctionnel « Timer » qui utilise le Hook « useEffect » pour démarrer un minuteur qui incrémente le nombre de secondes chaque seconde. Nous utilisons également la fonction de nettoyage retournée par « useEffect » pour arrêter l'intervalle lorsque le composant est démonté.

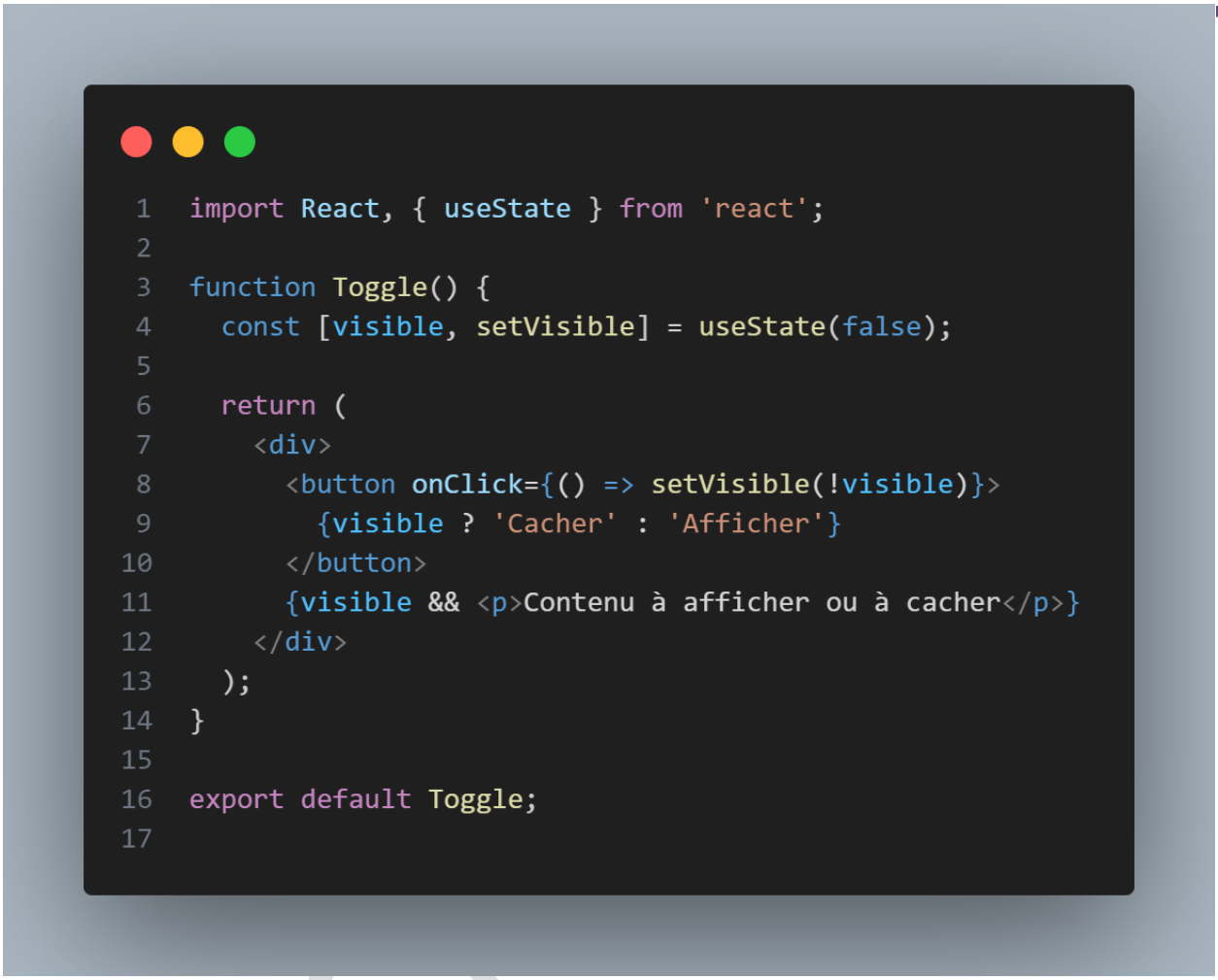
Ce chapitre met en avant la gestion de l'état et du cycle de vie des composants fonctionnels dans React, en utilisant les Hooks « useState » et « useEffect ».

V. RENDU CONDITIONNEL ET GESTION DES EVENEMENTS

RENDU CONDITIONNEL AVEC JAVASCRIPT

Le rendu conditionnel est un moyen de rendre certains éléments ou composants uniquement dans certaines conditions. En JavaScript, vous pouvez utiliser des structures conditionnelles telles que « if », « else », ou l'opérateur ternaire « condition ? true : false » pour rendre dynamiquement du contenu.

Voici un exemple de rendu conditionnel dans un composant fonctionnel :



```jsx
import React, { useState } from 'react';

function Toggle() {
  const [visible, setVisible] = useState(false);

  return (
    <div>
      <button onClick={() => setVisible(!visible)}>
        {visible ? 'Cacher' : 'Afficher'}
      </button>
      {visible && <p>Contenu à afficher ou à cacher</p>}
    </div>
  );
}

export default Toggle;

```

Dans cet exemple, nous avons un bouton qui, lorsqu'il est cliqué, modifie l'état « visible » du composant. En fonction de la valeur de « visible », le contenu à l'intérieur du « » sera affiché ou masqué.

GESTION DES EVENEMENTS DANS REACT

En React, la gestion des événements est similaire à la gestion des événements en JavaScript pur, mais avec quelques différences syntaxiques. Vous pouvez ajouter des gestionnaires d'événements aux éléments JSX en utilisant des attributs d'événement tels que « onClick », « onChange », « onSubmit », etc.

Voici un exemple de gestion des événements dans un composant fonctionnel :

```jsx
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Compteur: {count}</p>
      <button onClick={increment}>Incrémenter</button>
    </div>
  );
}

export default Counter;
```

Dans cet exemple, nous avons un bouton qui, lorsqu'il est cliqué, appelle la fonction « increment » pour augmenter le compteur.

UTILISATION DES METHODES DE CLASSE VS. DES FONCTIONS FLECHEES POUR LA GESTION DES EVENEMENTS

En React, il existe deux façons principales de définir des gestionnaires d'événements : en utilisant des méthodes de classe ou des fonctions fléchées. Les fonctions fléchées sont

généralement préférées car elles lient automatiquement le contexte, évitant ainsi la nécessité de lier explicitement « this ». Cependant, si vous avez besoin d'accéder à « this » dans votre méthode, vous devrez utiliser une méthode de classe ou lier explicitement « this » dans votre fonction fléchée.

Ce chapitre explore le rendu conditionnel et la gestion des événements dans les composants fonctionnels React. Nous avons examiné comment effectuer le rendu conditionnel en utilisant des expressions JavaScript et comment gérer les événements en utilisant des attributs d'événement JSX.

VI. LISTES ET CLES

RENDU DE LISTES

En React, vous pouvez rendre des listes dynamiques en utilisant la méthode « map() » sur un tableau de données pour créer une liste d'éléments JSX. Cela vous permet de générer des éléments de liste basés sur les données que vous avez, ce qui est particulièrement utile lorsque vous avez une liste dynamique à afficher.

Voici un exemple de rendu de liste dans un composant fonctionnel :

```jsx
import React from 'react';

function List() {
    const fruits = ['Pomme', 'Banane', 'Orange'];

    return (
        <ul>
            {fruits.map((fruit, index) => (
                <li key={index}>{fruit}</li>
            ))}
        </ul>
    );
}

export default List;
```

Dans cet exemple, nous avons une liste de fruits stockée dans un tableau « fruits ». Nous utilisons la méthode « map() » pour parcourir chaque élément du tableau et générer un élément « » pour chaque fruit. Nous utilisons également l'index de chaque élément comme clé (key) pour aider React à identifier chaque élément de manière unique.

UTILISATION DES CLES POUR OPTIMISER LE RENDU DES LISTES

Les clés (keys) sont des attributs spéciaux utilisés par React pour aider à identifier de manière unique chaque élément rendu dans une liste. Les clés aident React à identifier plus efficacement les changements lors de la mise à jour de la liste, ce qui améliore les performances et évite les rendus inutiles.

Il est important de noter que les clés doivent être uniques au sein de la liste et stables dans le temps (c'est-à-dire qu'elles ne doivent pas changer entre les rendus). Les identifiants uniques provenant des données de votre application sont généralement la meilleure option pour les clés.

Voici un exemple de rendu de liste avec des clés :

```jsx
import React from 'react';

function List() {
    const fruits = [
        { id: 1, name: 'Pomme' },
        { id: 2, name: 'Banane' },
        { id: 3, name: 'Orange' }
    ];

    return (
        <ul>
            {fruits.map(fruit => (
                <li key={fruit.id}>{fruit.name}</li>
            ))}
        </ul>
    );
}

export default List;
```

Dans cet exemple, chaque objet fruit a une propriété « id » unique qui est utilisée comme clé pour chaque élément de la liste. Cela garantit que chaque élément de la liste est identifié de manière unique, ce qui est crucial pour la performance et la stabilité du rendu.

Ce chapitre a couvert le rendu de listes dynamiques dans les composants fonctionnels React en utilisant la méthode « map() », ainsi que l'importance des clés pour optimiser le rendu des listes.

VII. FORMULAIRES DANS REACT

CONTROLE DES COMPOSANTS DE FORMULAIRE

En React, les formulaires fonctionnent de manière similaire aux formulaires HTML traditionnels, mais avec un peu de syntaxe supplémentaire pour gérer l'état des éléments de formulaire. Pour contrôler les éléments de formulaire, vous pouvez utiliser l'état (useState) pour suivre les valeurs des champs de formulaire et mettre à jour ces valeurs en réponse aux saisies de l'utilisateur.

Voici un exemple de formulaire contrôlé dans un composant fonctionnel :

```jsx
import React, { useState } from 'react';

function LoginForm() {
    const [username, setUsername] = useState('');
    const [password, setPassword] = useState('');

    const handleUsernameChange = (event) => {
        setUsername(event.target.value);
    };

    const handlePasswordChange = (event) => {
        setPassword(event.target.value);
    };

    const handleSubmit = (event) => {
        event.preventDefault();
        // Logique pour soumettre les données du formulaire
    };

    return (
        <form onSubmit={handleSubmit}>
            <label>
                Nom d'utilisateur :
                <input type="text" value={username} onChange={handleUsernameChange} />
            </label>
            <label>
                Mot de passe :
                <input type="password" value={password} onChange={handlePasswordChange} />
            </label>
            <button type="submit">Se connecter</button>
        </form>
    );
}

export default LoginForm;
```

Dans cet exemple, nous utilisons l'état (useState) pour suivre les valeurs des champs de formulaire « username » et « password ». Nous utilisons également les fonctions de gestion « handleUsernameChange » et « handlePasswordChange » pour mettre à jour l'état en réponse aux changements de saisie de l'utilisateur.

GESTION DES EVENEMENTS DE FORMULAIRE

En React, vous pouvez utiliser l'événement « onSubmit » sur le formulaire pour gérer la soumission du formulaire. Vous pouvez également utiliser les événements « onChange » sur les champs de formulaire pour détecter les changements de saisie de l'utilisateur en temps réel.

Dans l'exemple précédent, nous avons utilisé « onSubmit » pour appeler la fonction « handleSubmit » lors de la soumission du formulaire. La fonction « handleSubmit » peut ensuite gérer la soumission des données du formulaire, effectuer des validations, ou effectuer d'autres actions nécessaires.

VALIDATION DE FORMULAIRE

La validation de formulaire est une étape importante dans le processus de traitement des données du formulaire. En React, vous pouvez effectuer des validations de formulaire en utilisant l'état pour suivre l'état de la validation et en affichant des messages d'erreur en fonction de cet état.

Voici un exemple simplifié de validation de formulaire dans React :

```jsx
import React, { useState } from 'react';

function RegistrationForm() {
    const [email, setEmail] = useState('');
    const [password, setPassword] = useState('');
    const [error, setError] = useState('');

    const handleSubmit = (event) => {
        event.preventDefault();
        if (!email || !password) {
            setError('Veuillez remplir tous les champs.');
        } else {
            // Logique pour soumettre les données du formulaire
        }
    };

    return (
        <form onSubmit={handleSubmit}>
            <label>
                Email :
                <input type="email" value={email} onChange={(e) => setEmail(e.target.value)} />
            </label>
            <label>
                Mot de passe :
                <input type="password" value={password} onChange={(e) => setPassword(e.target.value)} />
            </label>
            <button type="submit">S'inscrire</button>
            {error && <p style={{ color: 'red' }}>{error}</p>}
        </form>
    );
}

export default RegistrationForm;
```

Dans cet exemple, nous vérifions si les champs « email » et « password » sont vides lors de la soumission du formulaire. Si l'un des champs est vide, nous affichons un message d'erreur approprié.

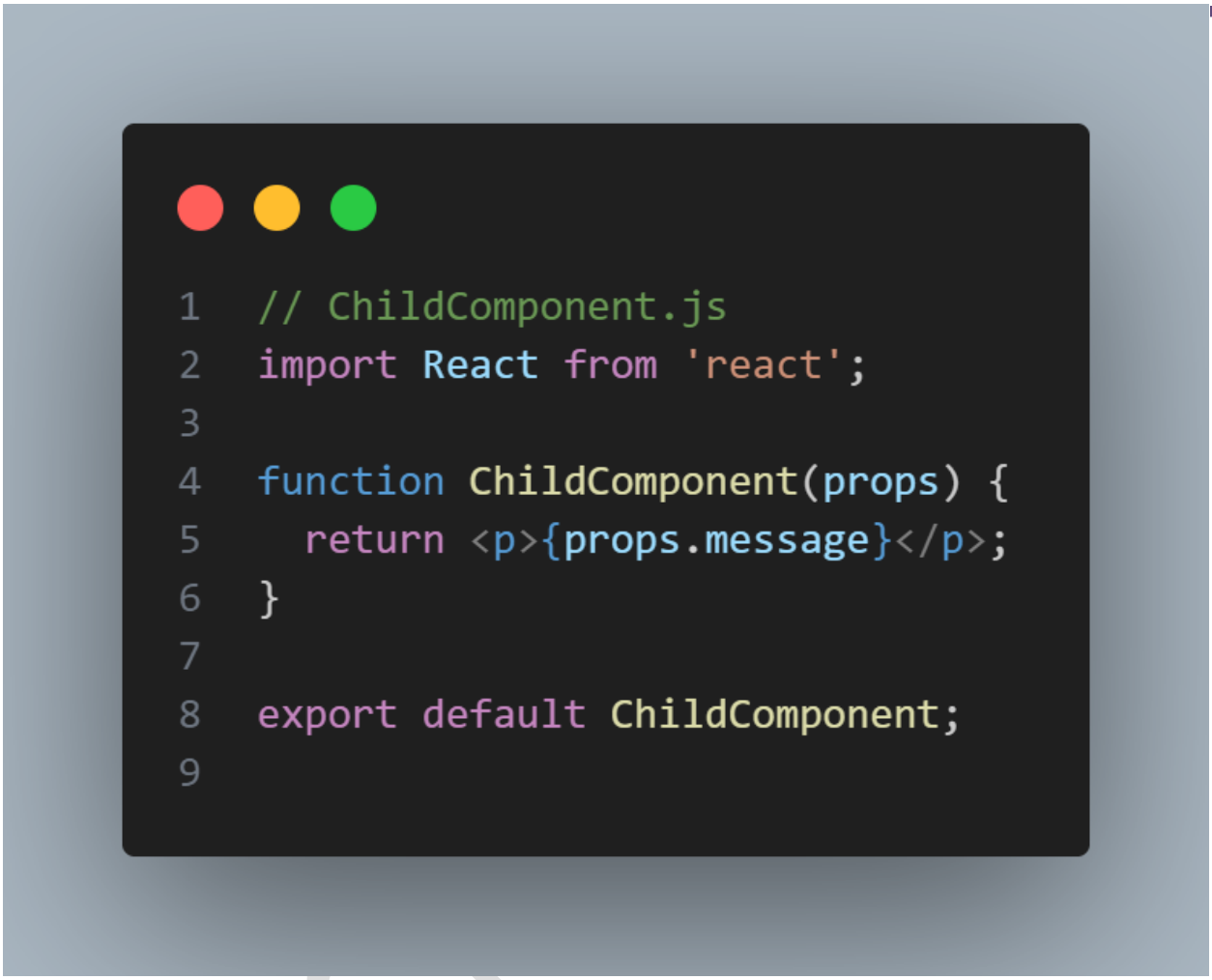
Ce chapitre couvre la création et la gestion de formulaires dans les composants fonctionnels React, y compris le contrôle des éléments de formulaire, la gestion des événements de formulaire et la validation de formulaire.

TRANSMISSION DES DONNEES ENTRE COMPOSANTS PARENT ET ENFANTS

En React, la communication entre composants se fait généralement en passant des données de composants parents à des composants enfants via les props. Les composants enfants peuvent recevoir des données sous forme de props et les utiliser pour afficher du contenu dynamique.

Voici un exemple de transmission de données entre un composant parent et un composant enfant :

```jsx
import React from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const message = "Bonjour de la part du parent !";

  return <ChildComponent message={message} />;
}

export default ParentComponent;
```



```
// ChildComponent.js
import React from 'react';

function ChildComponent(props) {
  return <p>{props.message}</p>;
}

export default ChildComponent;

```

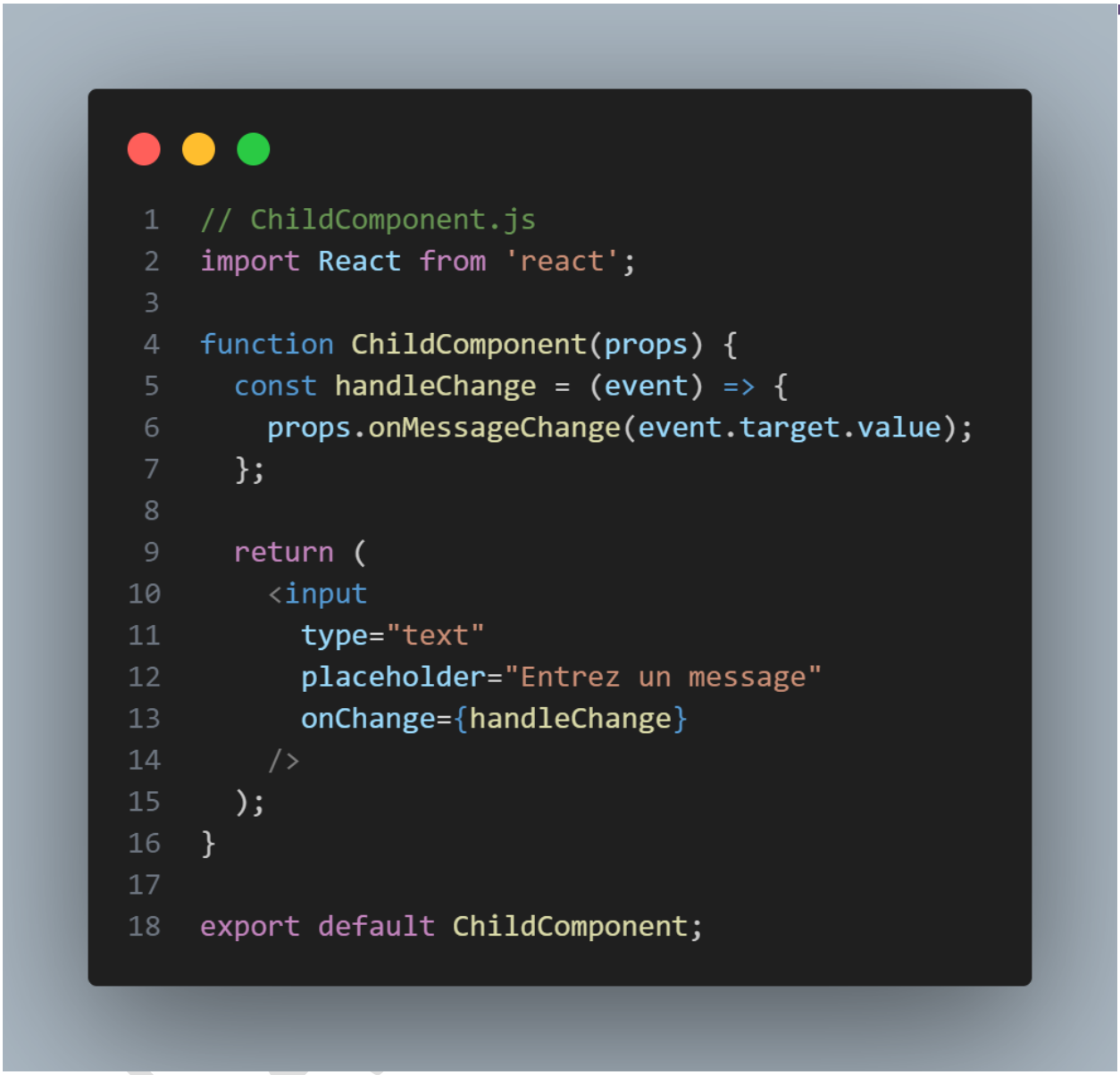
Dans cet exemple, le composant « ParentComponent » passe une prop « message » au composant « ChildComponent ». Le composant « ChildComponent » reçoit cette prop et l'affiche dans son rendu.

UTILISATION DES CALLBACKS POUR LA COMMUNICATION ENTRE ENFANTS ET PARENT

Pour permettre aux composants enfants de communiquer avec leur parent, vous pouvez passer des callbacks en tant que props aux composants enfants. Les composants enfants peuvent ensuite appeler ces callbacks pour transmettre des données ou des événements à leur parent.

Voici un exemple d'utilisation de callbacks pour la communication entre un composant enfant et son parent :

```jsx
// ParentComponent.js
import React, { useState } from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const [message, setMessage] = useState('');

  const handleMessageChange = (newMessage) => {
    setMessage(newMessage);
  };

  return (
    <div>
      <p>Message reçu du composant enfant : {message}</p>
      <ChildComponent onMessageChange={handleMessageChange} />
    </div>
  );
}

export default ParentComponent;
```



```jsx
// ChildComponent.js
import React from 'react';

function ChildComponent(props) {
  const handleChange = (event) => {
    props.onMessageChange(event.target.value);
  };

  return (
    <input
      type="text"
      placeholder="Entrez un message"
      onChange={handleChange}
    />
  );
}

export default ChildComponent;
```

Dans cet exemple, le composant « ChildComponent » appelle la fonction « onMessageChange » fournie par son parent chaque fois que le contenu de l'input change. Le parent écoute cet événement et met à jour son état avec la nouvelle valeur du message.

UTILISATION DU CONTEXTE POUR LA COMMUNICATION ENTRE COMPOSANTS ELOIGNES

Pour la communication entre composants éloignés ou pour éviter de passer des props à travers plusieurs niveaux de composants, vous pouvez utiliser le contexte (Context API) de React. Le contexte permet de partager des données entre des composants sans avoir besoin de passer explicitement des props à travers chaque niveau de l'arborescence des composants.

Voici un exemple d'utilisation du contexte pour la communication entre composants éloignés :

```jsx
// ThemeContext.js
import { createContext, useState } from 'react';

export const ThemeContext = createContext();

export function ThemeProvider({ children }) {
  const [theme, setTheme] = useState('light');

  return (
    <ThemeContext.Provider value={{ theme, setTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}
```

```jsx
// ChildComponent.js
import React, { useContext } from 'react';
import { ThemeContext } from './ThemeContext';

function ChildComponent() {
  const { theme, setTheme } = useContext(ThemeContext);

  const toggleTheme = () => {
    setTheme(theme === 'light' ? 'dark' : 'light');
  };

  return (
    <div>
      <p>Thème actuel : {theme}</p>
      <button onClick={toggleTheme}>Changer de thème</button>
    </div>
  );
}

export default ChildComponent;
```

Dans cet exemple, le composant « ChildComponent » utilise le Hook « useContext » pour accéder au contexte « ThemeContext » et lire le thème actuel ainsi que la fonction pour changer de thème.

Ce chapitre couvre différents moyens de communication entre composants dans React, y compris la transmission de données via les props, l'utilisation de callbacks pour la communication entre enfants et parent, et l'utilisation du contexte pour la communication entre composants éloignés.

REACT ROUTER POUR LA NAVIGATION

React Router est une bibliothèque populaire qui permet de gérer la navigation dans une application React. Elle offre des fonctionnalités pour définir des routes et pour rendre des composants en fonction de l'URL actuelle. Avec React Router, vous pouvez créer des applications à pages multiples et gérer facilement la navigation entre les différentes vues de votre application.

Voici un exemple simple de configuration de React Router dans une application React :

```jsx
// App.js
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Home from './Home';
import About from './About';
import Contact from './Contact';

function App() {
  return (
    <Router>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route path="/contact" component={Contact} />
      </Switch>
    </Router>
  );
}

export default App;
```

Dans cet exemple, nous utilisons « BrowserRouter » comme conteneur racine pour notre application. Nous définissons ensuite plusieurs routes à l'aide de « Route », en spécifiant le chemin de l'URL et le composant à rendre lorsque l'URL correspond à ce chemin.

AXIOS POUR LES REQUETES HTTP

Axios est une bibliothèque JavaScript populaire pour effectuer des requêtes HTTP depuis le navigateur ou Node.js. Elle offre une syntaxe simple et expressive pour effectuer des requêtes AJAX et gérer les réponses de manière asynchrone. Axios prend en charge les promesses et fournit des fonctionnalités telles que l'interception de requêtes et la gestion des erreurs.

Voici un exemple d'utilisation d'Axios pour effectuer une requête GET dans une application React :

```
// App.js
import React from 'react';
import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';
import Home from './Home';
import About from './About';
import Contact from './Contact';

function App() {
  return (
    <Router>
      <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
        <Route path="/contact" component={Contact} />
      </Switch>
    </Router>
  );
}

export default App;
```

Dans cet exemple, nous utilisons Axios pour effectuer une requête GET vers une URL distante. Nous utilisons ensuite « useState » pour stocker les données de réponse et « useEffect » pour déclencher la requête lors du montage du composant.

STYLED COMPONENTS POUR LA STYLISATION

Styled Components est une bibliothèque qui permet de styliser les composants React en utilisant des composants JavaScript. Elle offre une approche basée sur les composants pour écrire des styles CSS en tant que composants React, ce qui permet de créer des styles réutilisables et encapsulés pour vos composants. Styled Components prend également en charge les thèmes, les animations et les styles dynamiques basés sur les props.

Voici un exemple d'utilisation de Styled Components pour styliser un bouton dans une application React :

```jsx
import React from 'react';
import styled from 'styled-components';

const Button = styled.button`
  background-color: ${props => props.primary ? 'blue' : 'white'};
  color: ${props => props.primary ? 'white' : 'black'};
  font-size: 16px;
  padding: 10px 20px;
  border: 2px solid blue;
  border-radius: 5px;
`;

function MyComponent() {
    return (
        <div>
            <Button primary>Cliquez ici</Button>
            <Button>Autre bouton</Button>
        </div>
    );
}

export default MyComponent;
```

Dans cet exemple, nous utilisons « styled.button » pour créer un composant de bouton avec des styles CSS spécifiés en tant que propriétés du composant.

Ce chapitre a présenté quelques-uns des outils et bibliothèques complémentaires couramment utilisés avec React, notamment React Router pour la navigation, Axios pour les requêtes HTTP et Styled Components pour la stylisation.