COURS : REACT JS

Pr. Mohamed LACHGAR, lachgar.m@gmail.com 1ère partie

Contents

1 Introduction à React

1.1 Présentation de React

React est une bibliothèque JavaScript open source créée par Facebook en 2011. Elle est utilisée pour développer des interfaces utilisateur (UI) interactives et dynamiques pour les applications web et mobiles.

React utilise une approche basée sur des composants, où chaque composant représente une partie de l'interface utilisateur et est réutilisable dans toute l'application. Les composants sont des éléments autonomes qui peuvent être combinés pour créer des interfaces plus complexes.

React utilise également une approche déclarative pour la création d'interfaces utilisateur. Cela signifie que les développeurs déclarent simplement ce que l'interface utilisateur doit faire, et React gère les mises à jour en temps réel en fonction des changements de l'état des composants.

React est également très flexible et peut être intégré à d'autres technologies front-end, telles que React Native pour le développement d'applications mobiles, ou encore à des bibliothèques et frameworks de back-end, tels que Node.js ou Spring Boot.

Enfin, React possède une grande communauté de développeurs actifs et une abondance de ressources en ligne pour l'apprentissage et le développement. Cela en fait une technologie populaire et largement utilisée pour la création d'interfaces utilisateur modernes et réactives pour les applications web et mobiles.

1.2 Installation de l'environnement de développement

L'installation de l'environnement de développement pour React dépend du système d'exploitation que vous utilisez. Voici les étapes générales pour installer l'environnement de développement pour React :

1. **Installer Node.js**
   Node.js est une plateforme JavaScript qui permet d'exécuter du code JavaScript côté serveur. React utilise Node.js pour l'exécution de ses outils et dépendances. Vous pouvez télécharger la dernière version stable de Node.js à partir de leur site officiel : https://nodejs.org/

2. **Vérifier l'installation de NPM**
   NPM (Node Package Manager) est un gestionnaire de paquets pour les bibliothèques et les modules Node.js. C'est une dépendance importante pour installer des packages pour React et autres bibliothèques JavaScript.

   Une fois que Node.js est installé, vous pouvez vérifier si NPM est également installé en ouvrant une fenêtre de terminal et en entrant la commande suivante :

```
1 npm -v
```

3. **Installer un éditeur de code**
   Un éditeur de code est nécessaire pour écrire du code React. Il existe plusieurs éditeurs de code disponibles, tels que Visual Studio Code, Sublime Text, Atom, etc. Choisissez celui qui convient le mieux à vos besoins et téléchargez-le à partir de leur site officiel.

4. **Créer un nouveau projet React**
   Pour créer un nouveau projet React, vous pouvez utiliser l'outil de ligne de commande **create-react-app**. Pour l'installer, ouvrez une fenêtre de terminal et exécutez la commande suivante :

```
npm install -g create-react-app
```

   Ensuite, pour créer un nouveau projet React, exécutez la commande suivante :

```
create-react-app mon-projet
```

   Cela va créer un nouveau projet React avec toutes les dépendances et les fichiers nécessaires.

5. **Démarrer le serveur de développement**
   Pour démarrer le serveur de développement et visualiser votre projet React dans un navigateur, accédez au dossier de votre projet à partir de la ligne de commande et exécutez la commande suivante :

```
npm start
```

   Cela va lancer le serveur de développement et ouvrir une page web dans votre navigateur par défaut. Vous pouvez maintenant commencer à écrire du code React et visualiser les résultats en temps réel dans votre navigateur.

2 Composants React

2.1 Création de composants

Les composants sont un élément central de React. Les composants sont des éléments autonomes qui encapsulent une partie de l'interface utilisateur et peuvent être combinés pour créer des interfaces utilisateur plus complexes. Chaque composant a son propre état (state) et ses propres propriétés (props).

Il existe deux types de composants dans React :

1. **Les composants fonctionnels**
   Les composants fonctionnels sont des fonctions JavaScript qui prennent en entrée des

propriétés (props) et renvoient du JSX (JavaScript XML) pour décrire l'interface utilisateur. Ils sont simples à écrire et à comprendre, et peuvent être utilisés pour des composants simples qui ne nécessitent pas de gestion d'état complexe.

Voici un exemple de composant fonctionnel qui affiche un message de bienvenue :

```
import React from 'react';

function Welcome(props) {
  return <h1>Bienvenue, {props.name}!</h1>;
}

export default Welcome;
```

Dans cet exemple, le composant fonctionnel **Welcome** prend en entrée une propriété **name** et renvoie un élément **h1** contenant le message de bienvenue avec le nom donné.

2. **Les composants de classe**

   Les composants de classe sont des classes JavaScript qui étendent la classe **React.Component**. Ils ont leur propre état (state) et peuvent être utilisés pour des composants plus complexes qui nécessitent une gestion d'état plus avancée.

   Voici un exemple de composant de classe qui affiche un message de bienvenue et permet à l'utilisateur de cliquer sur un bouton pour modifier le message :

```
import React from 'react';

class Welcome extends React.Component {
  constructor(props) {
    super(props);
    this.state = { message: 'Bienvenue!' };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({ message: 'Bonjour!' });
  }

  render() {
    return (
      <div>
        <h1>{this.state.message}, {this.props.name}!</h1>
        <button onClick={this.handleClick}>Cliquez ici</button>
      </div>
    );
  }
}
```

```
23
24 export default Welcome;
```

Dans cet exemple, le composant de classe **Welcome** a son propre état **message** et une méthode **handleClick** pour modifier cet état lorsqu'un bouton est cliqué. La méthode **render** retourne l'interface utilisateur, qui inclut le message de bienvenue avec le nom donné, ainsi qu'un bouton pour modifier le message.

C'est ainsi que vous pouvez créer des composants fonctionnels et de classe dans React, qui encapsulent une partie de l'interface utilisateur et peuvent être combinés pour créer des interfaces utilisateur plus complexes.

2.2 Utilisation de propriétés

Les propriétés (ou "props" en abrégé) sont des arguments passés à un composant React, qui peuvent être utilisés pour personnaliser l'affichage de ce composant. Les propriétés sont passées aux composants sous la forme d'objets JavaScript, et sont accessibles dans les composants sous la forme de variables.

Voici un exemple d'utilisation de propriétés dans un composant fonctionnel :

```
1 import React from 'react';
2
3 function Welcome(props) {
4   return <h1>Bienvenue, {props.name}!</h1>;
5 }
6
7 export default Welcome;
```

Dans cet exemple, le composant fonctionnel **Welcome** prend en entrée une propriété **name**, qui est utilisée pour personnaliser le message de bienvenue affiché. Lorsque le composant est utilisé dans d'autres parties de l'application, la propriété **name** peut être passée comme suit :

```
1 <Welcome name="Mohamed" />
```

Dans cet exemple, le composant **Welcome** est utilisé avec une propriété **name** de valeur **"Mohamed"**. Le composant affichera donc le message "Bienvenue, John!".

Il est important de noter que les propriétés sont immuables, c'est-à-dire qu'elles ne peuvent pas être modifiées à l'intérieur du composant. Si vous devez modifier des données dans un composant, vous devez plutôt utiliser l'état (state) du composant.

En utilisant des propriétés, vous pouvez personnaliser l'affichage de vos composants React en leur passant des données dynamiques. Cela vous permet de créer des composants réutilisables et adaptables à différents contextes.

2.3 État et cycle de vie des composants

L'état (ou "state" en abrégé) est un concept important en React, qui permet aux composants de maintenir leur propre état interne et de modifier leur apparence en fonction de cet état. L'état est utilisé pour stocker des données qui peuvent changer au cours du temps, comme l'état d'une checkbox, le contenu d'un champ de saisie, etc.

Les composants de classe sont les seuls à pouvoir avoir un état dans React. Pour définir un état initial pour un composant de classe, vous pouvez utiliser le constructeur de la classe :

```
1  import React from 'react';
2
3  class Counter extends React.Component {
4    constructor(props) {
5      super(props);
6      this.state = { count: 0 };
7    }
8
9    render() {
10     return <div>Le compteur est à : {this.state.count}</div>;
11   }
12 }
13
14 export default Counter;
```

Dans cet exemple, le composant de classe **Counter** a un état initial de **count: 0** , qui est défini dans le constructeur. L'état est ensuite utilisé dans la méthode **render** pour afficher le compteur.

Lorsqu'un composant a un état, il peut également avoir un cycle de vie, qui est une série d'étapes qui se produisent à mesure que le composant est créé, mis à jour et détruit. Les méthodes du cycle de vie permettent de contrôler le comportement d'un composant à chaque étape de son cycle de vie.

Voici les méthodes du cycle de vie les plus couramment utilisées dans les composants de classe :

- **componentDidMount()**: Cette méthode est appelée une fois que le composant est monté dans le DOM. Elle est souvent utilisée pour initialiser des données à partir d'une source externe, comme un serveur ou une API.
- **componentDidUpdate()**: Cette méthode est appelée chaque fois que le composant est mis à jour. Elle est souvent utilisée pour effectuer des actions après que l'état ou les propriétés du composant ont été modifiés.
- **componentWillUnmount()**: Cette méthode est appelée juste avant que le composant soit démonté et retiré du DOM. Elle est souvent utilisée pour nettoyer les ressources associées au composant, comme des écouteurs d'événements.

Il est important de comprendre comment fonctionne le cycle de vie des composants pour pouvoir contrôler leur comportement et leur apparence. En utilisant l'état et le cycle de vie des composants, vous pouvez créer des interfaces utilisateur dynamiques et réactives en React.

3 Gestion des événements

3.1 Gestion d'événements avec React

En React, les événements sont gérés de manière similaire aux événements en JavaScript standard, mais avec une syntaxe légèrement différente. Les événements peuvent être gérés à la fois dans les composants de classe et les composants fonctionnels.

Voici un exemple de gestion d'événements dans un composant fonctionnel :

```
1  import React from 'react';
2
3  function Button(props) {
4    function handleClick() {
5      console.log('Le bouton a été cliqué.');
6    }
7
8    return <button onClick={handleClick}>Cliquer ici</button>;
9  }
10
11 export default Button;
```

Dans cet exemple, le composant fonctionnel **Button** affiche un bouton avec le texte "Cliquer ici". Lorsque le bouton est cliqué, la fonction **handleClick** est appelée, qui affiche un message dans la console.

Notez que la fonction **handleClick** est définie à l'intérieur du composant **Button**, et est ensuite passée à l'attribut **onClick** du bouton. Lorsque le bouton est cliqué, React appelle la fonction **handleClick** définie dans le composant **Button**.

Vous pouvez également utiliser des méthodes de classe pour gérer les événements dans les composants de classe :

```
1  import React from 'react';
2
3  class Button extends React.Component {
4    handleClick() {
5      console.log('Le bouton a été cliqué.');
6    }
7
8    render() {
9      return <button onClick={this.handleClick}>Cliquer ici</button>;
10   }
```

```
}

export default Button;
```

Dans cet exemple, la méthode **handleClick** est définie dans la classe **Button**, et est ensuite passée à l'attribut **onClick** du bouton. Lorsque le bouton est cliqué, React appelle la méthode **handleClick** définie dans la classe **Button**.

Il est important de noter que la fonction ou la méthode de gestion d'événements doit être liée au composant qui gère l'événement. Cela peut être fait en utilisant la méthode **bind**, en définissant la fonction à l'intérieur du constructeur, ou en utilisant une fonction fléchée pour définir la méthode dans la classe.

En utilisant la gestion d'événements, vous pouvez créer des interactions utilisateur dynamiques et réactives en React.

3.2 Création de formulaires

Les formulaires sont une partie importante de nombreuses applications Web, et React fournit un support intégré pour la création de formulaires.

Voici un exemple de formulaire simple dans React :

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

function ContactForm() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const [message, setMessage] = useState('');

  function handleSubmit(event) {
    event.preventDefault();
    console.log('Nom:', name);
    console.log('Email:', email);
    console.log('Message:', message);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>
        Nom:
        <input type="text" value={name} onChange={e => setName(e.target.↩
            value)} />
      </label>
      <label>
        Email:
        <input type="email" value={email} onChange={e => setEmail(e.target↩
            .value)} />
```

```
      </label>
      <label>
        Message:
        <textarea value={message} onChange={e => setMessage(e.target.value↩
            )} />
      </label>
      <button type="submit">Envoyer</button>
    </form>
  );
}

export default ContactForm;
```

Dans cet exemple, nous avons créé un composant fonctionnel **ContactForm** qui affiche un formulaire de contact avec des champs pour le nom, l'e-mail et le message. Nous avons utilisé le hook d'état **useState** pour gérer l'état du formulaire.

Lorsque le formulaire est soumis, la fonction **handleSubmit** est appelée, qui empêche le comportement par défaut du formulaire (recharger la page) et affiche les valeurs des champs dans la console.

Notez que chaque champ de formulaire a un **value** et un **onChange** attribut. Le **value** attribut est lié à l'état du formulaire, tandis que l'attribut **onChange** est utilisé pour mettre à jour l'état lorsque l'utilisateur saisit du texte.

En utilisant des formulaires dans React, vous pouvez facilement créer des interfaces utilisateur interactives pour collecter des données à partir de l'utilisateur.

3.3 Utilisation de formulaires contrôlés

Les formulaires contrôlés sont une technique utilisée dans React pour gérer les champs de formulaire. Dans un formulaire contrôlé, la valeur de chaque champ de formulaire est gérée par React, plutôt que par le navigateur.

Voici un exemple de formulaire contrôlé dans React :

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

function ContactForm() {
  const [formValues, setFormValues] = useState({
    name: '',
    email: '',
    message: ''
  });

  function handleChange(event) {
    const { name, value } = event.target;
    setFormValues(prevValues => ({
```

```
13        ...prevValues,
14        [name]: value
15      }));
16    }
17
18    function handleSubmit(event) {
19      event.preventDefault();
20      console.log('Nom:', formValues.name);
21      console.log('Email:', formValues.email);
22      console.log('Message:', formValues.message);
23    }
24
25    return (
26      <form onSubmit={handleSubmit}>
27        <label>
28          Nom:
29          <input type="text" name="name" value={formValues.name} onChange={↩
              handleChange} />
30        </label>
31        <label>
32          Email:
33          <input type="email" name="email" value={formValues.email} onChange↩
              ={handleChange} />
34        </label>
35        <label>
36          Message:
37          <textarea name="message" value={formValues.message} onChange={↩
              handleChange} />
38        </label>
39        <button type="submit">Envoyer</button>
40      </form>
41    );
42  }
43
44  export default ContactForm;
```

Dans cet exemple, nous avons utilisé le hook d'état **useState** pour stocker les valeurs de formulaire dans un objet **formValues**. Lorsque l'utilisateur saisit du texte dans un champ de formulaire, la fonction **handleChange** est appelée, qui met à jour l'état **formValues**.

Notez que chaque champ de formulaire a un **name**, un **value** et un **onChange** attribut. Le **name** attribut est utilisé pour identifier le champ de formulaire, tandis que l'attribut **value** est lié à l'état **formValues**. Lorsque l'utilisateur saisit du texte, l'attribut **onChange** est utilisé pour mettre à jour l'état **formValues**.

En utilisant des formulaires contrôlés dans React, vous pouvez facilement gérer l'état des champs de formulaire et effectuer des validations sur les entrées de l'utilisateur avant la soumis-

sion. Cela peut améliorer l'expérience utilisateur en fournissant des commentaires instantanés sur les entrées invalides.

3.4 Exercice : Formulaire de connexion

Objectif

Créer un formulaire de connexion avec deux champs : email et mot de passe.

Instructions

1. Créer un nouveau composant **LoginForm** qui rend un formulaire contenant deux champs : email et mot de passe.
2. Stocker les valeurs des champs de formulaire dans l'état du composant en utilisant le hook d'état **useState**.
3. Ajouter une fonction **handleChange** qui met à jour l'état lorsqu'un utilisateur saisit du texte dans les champs de formulaire.
4. Ajouter une fonction **handleSubmit** qui est appelée lorsque l'utilisateur soumet le formulaire. Dans cette fonction, afficher les valeurs des champs de formulaire dans la console à l'aide de **console.log()**.
5. Lié chaque champ de formulaire à l'état en utilisant les attributs **name**, **value** et **onChange**.
6. Ajouter un bouton de soumission de formulaire.

Exemple de code

```
1  import React, { useState } from 'react';
2
3  function LoginForm() {
4    const [formValues, setFormValues] = useState({
5      email: '',
6      password: ''
7    });
8
9    function handleChange(event) {
10     const { name, value } = event.target;
11     setFormValues(prevValues => ({
12       ...prevValues,
13       [name]: value
14     }));
15   }
16
```

```
17    function handleSubmit(event) {
18      event.preventDefault();
19      console.log('Email:', formValues.email);
20      console.log('Password:', formValues.password);
21    }
22
23    return (
24      <form onSubmit={handleSubmit}>
25        <label>
26          Email:
27          <input type="email" name="email" value={formValues.email} onChange↩
                ={handleChange} />
28        </label>
29        <label>
30          Mot de passe:
31          <input type="password" name="password" value={formValues.password}↩
                onChange={handleChange} />
32        </label>
33        <button type="submit">Se connecter</button>
34      </form>
35    );
36  }
37
38  export default LoginForm;
```

Dans cet exemple, nous avons créé un nouveau composant **LoginForm** qui contient un formulaire de connexion. Nous avons utilisé le hook d'état **useState** pour stocker les valeurs des champs de formulaire. Nous avons également ajouté les fonctions **handleChange** et **handleSubmit** pour gérer les changements de saisie de l'utilisateur et la soumission du formulaire.

En liant chaque champ de formulaire à l'état du composant, nous avons créé un formulaire contrôlé qui gère les entrées de l'utilisateur de manière efficace et valide les saisies avant la soumission.

4 Routage

4.1 Utilisation de React Router

React Router est une bibliothèque qui permet de gérer la navigation entre les pages de l'application web avec React. Voici les étapes à suivre pour utiliser React Router dans votre application :

1. Installation de React Router : Pour utiliser React Router, vous devez l'installer à l'aide de npm. Vous pouvez le faire en exécutant la commande suivante dans le terminal :

```
1  npm install react-router-dom
```

2. Importation des composants de React Router : Vous devez maintenant importer les composants nécessaires de React Router dans votre application. Pour la plupart des applications web, vous pouvez simplement importer **BrowserRouter**, **Routes**, **Route** et **Link** depuis la bibliothèque **react-router-dom**.

```
import { BrowserRouter as Router, Routes, Route, Link} from "react-router-dom";
```

3. Configuration des routes : Vous pouvez maintenant configurer les routes de votre application en utilisant le composant **BrowserRouter**. Vous devez envelopper votre application dans le composant **BrowserRouter** pour activer le routage.

```
 <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
</BrowserRouter>
```

Dans cet exemple, nous avons défini trois routes pour les pages d'accueil, à propos et de contact. La route exacte pour la page d'accueil est **/**, tandis que les routes pour les pages à propos et de contact sont respectivement **/about** et **/contact**. Nous avons également spécifié les composants à utiliser pour chaque route.

4. Navigation entre les pages : Vous pouvez maintenant naviguer entre les pages en utilisant le composant Link de React Router. Le composant **Link** est utilisé pour créer des liens vers d'autres pages de l'application web.

```
<nav>
  <ul>
    <li>
      <Link to="/">Accueil</Link>
    </li>
    <li>
      <Link to="/about">   propos</Link>
    </li>
    <li>
      <Link to="/contact">Contact</Link>
    </li>
  </ul>
</nav>
```

Dans cet exemple, nous avons créé un menu de navigation avec des liens vers les pages d'accueil, à propos et de contact. Nous avons utilisé le composant **Link** pour créer ces liens. Lorsque l'utilisateur clique sur l'un de ces liens, React Router gère la navigation vers la page appropriée.

Voici un exemple complet de l'utilisation de React Router dans une application React :

```
1  import { BrowserRouter, Route, Link } from 'react-router-dom';
2  import Home from './components/Home';
3  import About from './components/About';
4  import Contact from './components/Contact';
5
6  function App() {
7    return (
8      <BrowserRouter>
9        <nav>
10         <ul>
11           <li>
12             <Link to="/">Accueil</Link>
13           </li>
14           <li>
15             <Link to="/about">   propos</Link>
16           </li>
17           <li>
18             <Link to="/contact">Contact</Link>
19           </li>
20         </ul>
21       </nav>
22       <Route path="/" exact>
23         <Home />
24       </Route>
25       <Route path="/about">
26         <About />
27       </Route>
28       <Route path="/contact">
29         <Contact />
30       </Route>
31     </BrowserRouter>
32   );
33 }
34
35 export default App;
```

5 Fonctions utiles

Les fonctions **map**, **filter** et **reduce** sont des fonctions de haut niveau en JavaScript qui permettent de manipuler des tableaux de données de manière simple et efficace. Elles sont très utiles dans le développement d'applications web pour transformer, filtrer et réduire des données. Dans ce cours, nous allons examiner ces fonctions en détail et voir comment elles peuvent être utilisées pour traiter des données dans JavaScript.

5.1 La fonction map()

La fonction **map** est utilisée pour transformer les éléments d'un tableau en appliquant une fonction à chaque élément. La fonction prend en entrée un tableau et retourne un nouveau tableau avec les éléments transformés. Voici la syntaxe de la fonction **map** :

```
1 const newArray = oldArray.map(callback)
```

La fonction **map** prend une fonction de rappel (callback) en entrée qui est appliquée à chaque élément du tableau. Cette fonction peut avoir jusqu'à trois arguments : l'élément actuel, l'index de l'élément actuel et le tableau sur lequel la fonction **map** est appelée.

Voici un exemple simple pour illustrer l'utilisation de la fonction **map** :

```
1 const numbers = [1, 2, 3, 4];
2 const doubledNumbers = numbers.map((number) => number * 2);
3 console.log(doubledNumbers); // [2, 4, 6, 8]
```

Dans cet exemple, la fonction **map** prend le tableau **numbers** en entrée et retourne un nouveau tableau **doubledNumbers** dans lequel chaque élément est multiplié par 2.

5.2 La fonction filter()

La fonction **filter** est utilisée pour filtrer les éléments d'un tableau en appliquant une fonction de test à chaque élément. La fonction prend en entrée un tableau et retourne un nouveau tableau avec les éléments filtrés. Voici la syntaxe de la fonction **filter** :

```
1 const newArray = oldArray.filter(callback)
```

La fonction **filter** prend une fonction de rappel (callback) en entrée qui est appliquée à chaque élément du tableau. Cette fonction doit retourner **true** ou **false** pour chaque élément du tableau. Les éléments pour lesquels la fonction retourne true sont inclus dans le nouveau tableau retourné, et les éléments pour lesquels la fonction retourne **false** sont exclus.

Voici un exemple simple pour illustrer l'utilisation de la fonction **filter** :

```
1 const numbers = [1, 2, 3, 4];
```

```
const filteredNumbers = numbers.filter((number) => number % 2 === 0);
console.log(filteredNumbers); // [2, 4]
```

Dans cet exemple, la fonction **filter** prend le tableau **numbers** en entrée et retourne un nouveau tableau **filteredNumbers** qui ne contient que les nombres pairs.

5.3 La fonction reduce()

La fonction **reduce** est utilisée pour réduire les valeurs d'un tableau en une seule valeur. Elle prend en paramètre deux arguments: une fonction de rappel et une valeur initiale (optionnelle).

La fonction de rappel prend deux arguments: l'accumulateur (qui est la valeur résultante après chaque itération) et l'élément en cours de traitement. La fonction doit retourner une valeur qui sera l'accumulateur de la prochaine itération.

Voici un exemple simple qui utilise la fonction reduce pour calculer la somme des éléments d'un tableau:

```
const nombres = [1, 2, 3, 4, 5];
const somme = nombres.reduce((accumulateur, element) => accumulateur + element, 0);

console.log(somme); // Affiche 15
```

Dans cet exemple, la fonction reduce prend une fonction de rappel qui ajoute l'accumulateur et l'élément en cours. L'accumulateur est initialisé à 0 avec la valeur optionnelle passée comme deuxième argument.

La fonction reduce est également souvent utilisée pour transformer un tableau en un objet. Par exemple, si nous avons un tableau d'objets avec un identifiant unique pour chaque objet, nous pouvons utiliser la fonction reduce pour créer un objet où les clés sont les identifiants et les valeurs sont les objets correspondants. Voici un exemple:

```
const utilisateurs = [
  { id: 1, nom: 'Ahmed', age: 25 },
  { id: 2, nom: 'Fatima', age: 30 },
  { id: 3, nom: 'Omar', age: 20 }
];

const utilisateursParId = utilisateurs.reduce((obj, utilisateur) => {
  obj[utilisateur.id] = utilisateur;
  return obj;
}, {});

console.log(utilisateursParId[2]); // Affiche { id: 2, nom: 'Fatima', age: 30 }
```

Dans cet exemple, nous avons créé un objet vide avec comme valeur initiale de l'accumulateur. Dans la fonction de rappel, nous avons ajouté une propriété à l'objet avec la clé correspondant à l'identifiant de l'utilisateur et la valeur correspondant à l'objet utilisateur. La fonction de rappel retourne l'objet qui sera l'accumulateur de la prochaine itération.

5.4 Autres fonctions

Les fonctions **map**, **filter** et **reduce** sont des outils puissants pour travailler avec des tableaux en JavaScript. Elles peuvent vous aider à écrire un code plus élégant, plus expressif et plus efficace. En comprenant ces fonctions et en sachant quand les utiliser, vous pouvez simplifier vos projets et gagner du temps lors du développement.

Il existe d'autres fonctions de tableau utiles en JavaScript, telles que **find**, **every** et **some**, qui peuvent également vous aider à travailler avec des tableaux de manière plus efficace. En explorant ces fonctions et en pratiquant leur utilisation, vous pouvez devenir un programmeur JavaScript plus efficace et plus productif.

5.5 TP : NodeJs

L'objectif de ce TP est de manipuler des tableaux en utilisant les fonctions map, filter et reduce de JavaScript. Nous allons travailler avec un tableau de prénoms marocains.

Étape 1: Mise en place du projet

1. Créer un nouveau dossier TP-JavaScript
2. Ouvrir un terminal dans ce dossier
3. Exécuter la commande **npm init** et répondre aux questions pour initialiser le projet avec un fichier **package.json**.

Étape 2: Création du fichier index.js

1. Créer un nouveau fichier **index.js** à la racine du projet.
2. Ouvrir le fichier **index.js** dans un éditeur de code.

Étape 3: Définir le tableau de prénoms

1. Définir un tableau **prenoms** contenant une liste de prénoms marocains.

```
1 const prenoms = ["Fatima", "Mohamed", "Amina", "Ahmed", "Samira", "↩
    Hassan", "Nadia", "Youssef"];
```

Étape 4: Utiliser la fonction map

1. Utiliser la fonction **map** pour créer un nouveau tableau contenant les prénoms en majuscules.

```
const prenomsMajuscules = prenoms.map((prenom) => prenom.toUpperCase());
console.log(prenomsMajuscules); // ["FATIMA", "MOHAMED", "AMINA", "AHMED", "SAMIRA", "HASSAN", "NADIA", "YOUSSEF"]
```

2. Utiliser la fonction **map** pour créer un nouveau tableau contenant la longueur de chaque prénom.

```
const longueurs = prenoms.map((prenom) => prenom.length);
console.log(longueurs); // [6, 7, 5, 5, 6, 6, 5, 7]
```

Étape 5: Utiliser la fonction filter

1. Utiliser la fonction **filter** pour créer un nouveau tableau contenant les prénoms qui ont une longueur supérieure à 5.

```
const prenomsLongs = prenoms.filter((prenom) => prenom.length > 5);
console.log(prenomsLongs); // ["Fatima", "Mohamed", "Samira", "Hassan", "Youssef"]
```

2. Utiliser la fonction **filter** pour créer un nouveau tableau contenant les prénoms qui commencent par la lettre "A".

```
const prenomsCommencantParA = prenoms.filter((prenom) => prenom.startsWith("A"));
console.log(prenomsCommencantParA); // ["Amina", "Ahmed"]
```

Étape 6: Utiliser la fonction reduce

1. Utiliser la fonction **reduce** pour calculer la somme des longueurs de tous les prénoms.

```
const sommeLongueurs = prenoms.reduce((total, prenom) => total + prenom.length, 0);
console.log(sommeLongueurs); // 47
```

2. Utiliser la fonction reduce pour trouver le prénom le plus long.

```
const prenoms = ['Fatima', 'Mohamed', 'Amina', 'Ali', 'Omar', 'Yassine'];

const longestPrenom = prenoms.reduce((longest, current) => {
  if (current.length > longest.length) {
    return current;
  } else {
    return longest;
  }
}, '');

console.log(longestPrenom); // affiche 'Mohamed'
```

Dans cet exemple, on initialise l'accumulateur à une chaîne vide ". À chaque itération, la fonction de réduction compare la longueur du prénom en cours de traitement avec celle du prénom stocké dans l'accumulateur, et retourne le prénom le plus long. À la fin de la réduction, l'accumulateur contient le prénom le plus long trouvé dans le tableau.

Etape 7 : Lancer le projet

Pour lancer un projet Node.js, vous pouvez suivre ces étapes générales :

1. Ouvrir votre **terminal/commande** ligne et naviguez jusqu'au répertoire racine de votre projet.
2. Installer toutes les dépendances du projet avec la commande **npm install**.
3. Lancer le serveur avec la commande **node server.js** ou **npm start**.
4. Dans votre navigateur, accéder à l'URL **http://localhost:<port>** pour voir l'application.

Noter que vous devrez remplacer <port> par le numéro de port sur lequel votre serveur écoute (par défaut c'est 3000). Si le serveur est lancé sur un port différent, vous devrez utiliser ce numéro de port à la place.

6 Gestion des requêtes HTTP avec Axios

6.1 Présentation d'Axios

Axios est une bibliothèque JavaScript utilisée pour effectuer des requêtes HTTP depuis une application React. Elle permet de récupérer des données à partir d'une API ou d'un serveur distant et de les utiliser dans une application. Axios prend en charge les requêtes HTTP avec les méthodes GET, POST, PUT, DELETE, etc., ainsi que l'envoi de données en format JSON.

Pour utiliser Axios dans une application React, il est d'abord nécessaire de l'installer en utilisant le gestionnaire de paquets npm ou yarn. Une fois installé, Axios peut être importé dans le code source de l'application et utilisé pour effectuer des requêtes HTTP vers une API ou un serveur.

Par exemple, pour effectuer une requête GET vers une API qui renvoie des données en format JSON, le code suivant peut être utilisé :

```
1  import axios from 'axios';
2
3  axios.get('https://api.example.com/data')
4    .then(response => {
5      // traiter les données de réponse
6      console.log(response.data);
7    })
8    .catch(error => {
9      // gérer les erreurs
10     console.log(error);
11   });
```

Cette requête enverra une requête GET à l'URL spécifiée et récupérera les données de réponse en format JSON. Le **.then()** permet de traiter la réponse en cas de succès, tandis que le **.catch()** permet de gérer les erreurs en cas d'échec de la requête.

Axios est donc une bibliothèque utile pour effectuer des requêtes HTTP depuis une application React et permet de récupérer et d'utiliser des données à partir d'une API ou d'un serveur distant.

6.2 Installation d'Axios

6.3 Réalisation de requêtes HTTP avec Axios

Pour installer Axios dans une application React, vous pouvez utiliser le gestionnaire de paquets npm ou yarn. Voici les étapes pour installer Axios en utilisant npm :

1. Ouvrir un terminal et accédez au répertoire racine de votre application React.
2. Taper la commande suivante pour installer Axios :

```
npm install axios
```

3. Attendre que la commande soit exécutée. Une fois terminée, Axios sera installé et prêt à être utilisé dans votre application.

Une fois installé, vous pouvez importer Axios dans votre code source et l'utiliser pour effectuer des requêtes HTTP vers une API ou un serveur distant. Voici un exemple :

```
import axios from 'axios';

axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.log(error);
  });
```

Dans cet exemple, Axios est importé dans le code source, puis utilisé pour effectuer une requête GET vers l'URL spécifiée. La réponse est traitée dans la fonction de rappel **.then()** si la requête réussit, ou dans la fonction de rappel **.catch()** si la requête échoue.

6.4 Utilisation d'Axios pour gérer les erreurs de requêtes

Lorsque vous effectuez une requête avec Axios, il est important de gérer les erreurs qui peuvent se produire, telles qu'une erreur de réseau ou une erreur de syntaxe dans la requête. Pour gérer les erreurs de requête avec Axios, vous pouvez utiliser la méthode **.catch()**.

Voici un exemple qui montre comment gérer les erreurs de requête avec Axios :

```
import axios from 'axios';

axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.log(error);
  });
```

Dans cet exemple, la méthode **.get()** est utilisée pour envoyer une requête GET à l'URL spécifiée. Si la requête réussit, la réponse sera traitée dans la fonction de rappel **.then()**. Si la requête échoue pour une raison quelconque, elle sera traitée dans la fonction de rappel **.catch()**. L'objet error dans la fonction de rappel **.catch()** contient des informations sur l'erreur qui s'est produite.

Vous pouvez également personnaliser le traitement des erreurs en fonction de l'erreur spécifique. Par exemple, pour gérer une erreur de réseau, vous pouvez afficher un message d'erreur à l'utilisateur, comme ceci :

```
axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    if (error.response) {
      // La requête a été effectuée et le serveur a renvoyé une réponse avec un code d erreur
      console.log(error.response.data);
      console.log(error.response.status);
      console.log(error.response.headers);
    } else if (error.request) {
      // La requête a été effectuée mais le serveur n a pas renvoyé de réponse
      console.log(error.request);
    } else {
      // Une erreur s est produite lors de la configuration de la requête
      console.log('Erreur', error.message);
    }
    console.log(error.config);
  });
```

Dans cet exemple, l'objet **error** est vérifié pour voir s'il contient une réponse ou une demande. Si une réponse est renvoyée avec un code d'erreur, les détails de la réponse sont affichés. Si la requête a été effectuée mais qu'aucune réponse n'a été renvoyée, le contenu de la demande est affiché. Si une erreur s'est produite lors de la configuration de la requête, un message d'erreur est affiché.

7 Utilisation de React JS avec Axios

7.1 Utilisation d'Axios pour récupérer des données dans un composant React

Pour récupérer des données dans un composant React à l'aide d'Axios, vous pouvez utiliser la méthode componentDidMount() pour effectuer une requête HTTP à une API ou à un serveur distant dès que le composant est monté. Voici un exemple :

```
import React, { Component } from 'react';
import axios from 'axios';

class MyComponent extends Component {
  state = {
```

```
    data: null
  }

  componentDidMount() {
    axios.get('https://api.example.com/data')
      .then(response => {
        this.setState({ data: response.data });
      })
      .catch(error => {
        console.log(error);
      });
  }

  render() {
    const { data } = this.state;

    return (
      <div>
        {data && (
          <ul>
            {data.map(item => (
              <li key={item.id}>{item.name}</li>
            ))}
          </ul>
        )}
      </div>
    );
  }
}

export default MyComponent;
```

Dans cet exemple, la méthode **componentDidMount()** est utilisée pour effectuer une requête GET à l'URL spécifiée dès que le composant est monté. Si la requête réussit, les données sont stockées dans l'état du composant à l'aide de la méthode **setState()**. Si la requête échoue pour une raison quelconque, l'erreur est affichée dans la console.

Les données récupérées peuvent ensuite être affichées dans le rendu du composant en utilisant la syntaxe JSX. Dans cet exemple, si les données sont présentes dans l'état du composant, elles sont affichées dans une liste **<ul>** avec chaque élément affiché dans un **<li>**. Notez que la vérification **data** est utilisée pour s'assurer que data n'est pas nul avant de tenter de l'afficher.

Cette méthode est un moyen courant d'obtenir des données à partir d'une API ou d'un serveur distant dans un composant React en utilisant Axios. Cependant, il existe d'autres façons de récupérer des données avec Axios, comme l'utilisation de **async/await** ou la création de fonctions d'assistance personnalisées.

7.2 componentDidMount() vs useEffect

componentDidMount() et **useEffect** sont deux méthodes utilisées pour effectuer des opérations liées aux effets secondaires dans React.

componentDidMount() est une méthode du cycle de vie des composants dans les composants de classe React qui est appelée une fois lorsque le composant est monté sur le DOM. Elle est couramment utilisée pour effectuer des tâches d'initialisation, telles que la mise en place d'écouteurs d'événements ou la récupération de données à partir d'une API.

D'autre part, **useEffect** est un crochet **(hook)** dans les composants fonctionnels React qui permet d'effectuer des effets secondaires dans les composants fonctionnels. Il peut être utilisé pour remplacer **componentDidMount**, mais il est plus polyvalent car il peut gérer plusieurs scénarios, tels que les mises à jour de l'état du composant ou les modifications des props.

Voici un exemple d'utilisation de **componentDidMount** dans un composant de classe pour récupérer des données :

```
class MonComposant extends React.Component {
  componentDidMount() {
    fetch('/api/data')
      .then(response => response.json())
      .then(data => {
        this.setState({ data });
      });
  }

  render() {
    return <div>{this.state.data}</div>;
  }
}
```

Et voici un exemple d'utilisation de **useEffect** dans un composant fonctionnel pour obtenir le même résultat :

```
function MonComposant() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('/api/data')
      .then(response => response.json())
      .then(data => {
        setData(data);
      });
  }, []);

  return <div>{data}</div>;
}
```

Notez que dans l'exemple de **useEffect**, nous utilisons le crochet **useState** pour gérer l'état des données, et nous passons un tableau de dépendances vide en tant que deuxième argument à **useEffect**. Cela garantit que l'effet n'est exécuté qu'une fois lorsque le composant est monté, similaire à **componentDidMount**.

7.3 Intégration de la réponse d'une requête HTTP dans l'état d'un composant

L'intégration de la réponse d'une requête HTTP dans l'état d'un composant React à l'aide d'Axios est un processus simple qui peut être accompli en utilisant la méthode **setState()**. Voici un exemple de code qui illustre comment intégrer la réponse d'une requête HTTP dans l'état d'un composant React :

```
1  import React, { Component } from 'react';
2  import axios from 'axios';
3
4  class MyComponent extends Component {
5    state = {
6      data: null
7    }
8
9    componentDidMount() {
10     axios.get('https://api.example.com/data')
11       .then(response => {
12         this.setState({ data: response.data });
13       })
14       .catch(error => {
15         console.log(error);
16       });
17   }
18
19   render() {
20     const { data } = this.state;
21
22     return (
23       <div>
24         {data && (
25           <ul>
26             {data.map(item => (
27               <li key={item.id}>{item.name}</li>
28             ))}
29           </ul>
30         )}
31       </div>
32     );
```

```
33    }
34  }
35
36  export default MyComponent;
```

Dans cet exemple, la méthode **setState()** est utilisée pour mettre à jour l'état du composant avec les données récupérées dans la réponse. Lorsque l'état est mis à jour, le composant est automatiquement re-rendu avec les nouvelles données.

Notez que la mise à jour de l'état avec la méthode **setState()** est asynchrone, ce qui signifie que les modifications peuvent ne pas être reflétées immédiatement. Si vous avez besoin d'exécuter du code après la mise à jour de l'état, vous pouvez utiliser une fonction de rappel en tant que deuxième argument de la méthode **setState()** :

```
1  this.setState({ data: response.data }, () => {
2    console.log('L\'état a été mis à jour !');
3  });
```

Dans cet exemple, la fonction de rappel est exécutée après la mise à jour de l'état et affiche un message dans la console.

7.4 Utilisation d'Axios pour envoyer des données dans une requête HTTP

Axios est une bibliothèque JavaScript populaire pour effectuer des requêtes HTTP à partir de navigateurs ou de serveurs Node.js. Elle fournit une interface simple et élégante pour effectuer des requêtes HTTP, gérer les erreurs et transformer les données de réponse.

Pour envoyer des données dans une requête HTTP avec Axios, vous pouvez utiliser la méthode **axios.post(url, data, config)**. Cette méthode envoie une requête POST à l'URL spécifiée avec les données fournies dans l'objet data. Vous pouvez également fournir des options supplémentaires dans l'objet config.

Voici un exemple d'utilisation d'Axios pour envoyer des données dans une requête POST :

```
1  import axios from 'axios';
2
3  const postData = {
4    name: 'Mohamed LACHGAR',
5    email: 'lachgar.m@gmail.com'
6  };
7
8  axios.post('/api/user', postData)
9    .then(response => {
10     console.log('Data sent successfully:', response.data);
11   })
12   .catch(error => {
13     console.error('Error sending data:', error);
```

```
14   });
```

Dans cet exemple, nous importons Axios et créons un objet **postData** qui contient les données que nous voulons envoyer. Nous utilisons ensuite la méthode **axios.post** pour envoyer les données à l'URL **/api/user**. La méthode **post** prend deux arguments : l'URL de destination et les données à envoyer.

Ensuite, nous chaînons un appel à la méthode **then** pour gérer la réponse de la requête. Dans ce cas, nous affichons simplement les données renvoyées par le serveur dans la console. Nous ajoutons également un appel à la méthode **catch** pour gérer les erreurs potentielles.

Il est important de noter que dans cet exemple, nous utilisons l'API Promise d'Axios pour gérer les réponses asynchrones de la requête. Nous pouvons également utiliser la syntaxe **async/await** pour effectuer la même requête :

```
1  import axios from 'axios';
2
3  const postData = {
4    name: 'Mohamed LACHGAR',
5    email: 'lachgar.m@gmail.com'
6  };
7
8  async function sendPostData() {
9    try {
10     const response = await axios.post('/api/user', postData);
11     console.log('Data sent successfully:', response.data);
12   } catch (error) {
13     console.error('Error sending data:', error);
14   }
15 }
16
17 sendPostData();
```

Dans cet exemple, nous définissons une fonction **sendPostData** qui utilise la syntaxe **async/await** pour envoyer les données et gérer la réponse de la requête. La fonction est ensuite appelée en dehors de la définition.

TP : Gestion des villes et zones

Objectif : Le but de ce TP est de créer une application Web qui permettra de gérer les villes et les zones d'une ville. Les utilisateurs pourront ajouter, modifier et supprimer des villes et des zones, ainsi que visualiser la liste des villes et des zones existantes.

Technologies utilisées : React JS pour la partie front-end et Spring Boot pour la partie back-end.

Fonctionnalités de l'application :

- Afficher la liste des villes existantes
- Ajouter une nouvelle ville avec son nom
- Modifier le nom d'une ville existante
- Supprimer une ville existante
- Afficher la liste des zones d'une ville donnée
- Ajouter une nouvelle zone à une ville donnée avec son nom
- Modifier le nom d'une zone existante
- Supprimer une zone existante