Introduction à React.js

Objectifs du TP

Ce TP va vous permettre de découvrir React.js en créant une application de gestion de tâches (Todo List). À travers ce projet concret, vous allez comprendre :

- Les concepts fondamentaux de React
- La création d'une interface utilisateur dynamique
- La gestion de l'état (state) dans une application React
- Les interactions entre composants
- La manipulation des événements utilisateur

Prérequis

Avant de commencer, assurez-vous d'avoir :

- Des connaissances de base en HTML, CSS et JavaScript
- Node.js installé sur votre machine (version LTS recommandée)
- Un éditeur de code (nous recommandons VS Code avec l'extension React)
- Git pour cloner le dépôt de la maquette

💡 Pour vérifier que Node.js est bien installé, ouvrez un terminal et tapez :

```
node --version
npm --version
```

Les deux commandes devraient afficher un numéro de version.

Découverte de la maquette

Présentation

La maquette que nous allons reproduire est une Todo List interactive permettant de :

- Afficher une liste de tâches

- Ajouter de nouvelles tâches
- Marquer des tâches comme terminées
- Supprimer des tâches
- Voir le nombre de tâches terminées/total

Récupération du code

1. Clonez le dépôt contenant la maquette :

```
git clone https://github.com/Formacitron/formation-decouverte-react.js-maquette
cd formation-decouverte-react.js-maquette
```

1. Ouvrez le fichier `index.html` dans votre navigateur
2. Examinez le code HTML et CSS fourni - nous allons le transformer en composants React

💡 Notez la structure HTML de la maquette, en particulier :

- L'en-tête avec le logo et le compteur
- Le formulaire d'ajout de tâches
- La liste des tâches avec leurs états (terminé/non terminé)

Création du projet React

Introduction à Vite

Pour notre projet, nous allons utiliser Vite, un outil de build moderne qui offre :

- Des temps de démarrage ultra-rapides
- Un rechargement à chaud (HMR) quasi instantané
- Une configuration simple et intuitive
- Une meilleure performance en développement

Initialisation avec Vite

1. Ouvrez un terminal dans le dossier où vous souhaitez créer votre projet
2. Créez un nouveau projet React avec Vite :

```
npm create vite@latest tdtodo -- --template react
```

💡 Cette commande :

- Crée un nouveau dossier `tdtodo`
- Configure un projet React minimal
- Met en place l'environnement Vite
- Une fois la création terminée, installez les dépendances et démarrez l'application :

```
cd tdtodo
npm install
npm run dev
```

Votre navigateur devrait s'ouvrir automatiquement sur http://localhost:5173

Structure du projet créé

Explorez les fichiers générés par Vite :

- `package.json` : Configuration du projet et dépendances
- `index.html` : Point d'entrée HTML (à la racine du projet)
- `src/` : Code source de l'application
    - `App.jsx` : Composant principal
    - `main.jsx` : Point d'entrée JavaScript
    - `App.css` : Styles du composant App
- `vite.config.js` : Configuration de Vite

💡 Différences notables avec Create React App :

- `index.html` est à la racine du projet
- Les fichiers React utilisent l'extension `.jsx`
- La configuration est plus légère et explicite
- Le serveur de développement est plus rapide

Comprendre la structure de base

Le fichier `main.jsx` est le point d'entrée de l'application :

```jsx
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
import './index.css'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
)
```

Le fichier `App.jsx` contient notre composant principal :

```jsx
function App() {
  return (
    <>
      <h1>Hello Vite + React</h1>
    </>
  )
}

export default App
```

Préparation des ressources

Copie des assets

Nous allons d'abord intégrer les ressources de la maquette dans notre projet React :

1. Dans le dossier `srcs/assets` de votre projet :
    - Remplacez `react.svg` par celui de la maquette
    - Copiez le fichier `logo.svg` fourni
2. Videz le contenu du fichier `src/index.css`
3. Remplacez le contenu de `src/App.css` par celui du fichier `style.css` de la maquette
    - Modifiez le chemin de l'image du logo React: `background-image: url('assets/react.svg');`

Utilisation de `@fortawesome/react-fontawesome`

Font Awesome est une bibliothèque d'icones. Cette approche utilise les composants React officiels de Font Awesome :

1. Installation des paquets nécessaires : Avec un terminal, dans le dossier de votre projet, lancez la commande suivante:

```
npm install @fortawesome/react-fontawesome @fortawesome/fontawesome-svg-core @forta
```

1. Dans votre fichier `main.jsx`, ajoutez la configuration (juste au dessue de la partie 'createRoot...') :

```
import { library } from '@fortawesome/fontawesome-svg-core'
import { faSquare, faCheckSquare } from '@fortawesome/free-regular-svg-icons'
import { faTasks, faTrash } from '@fortawesome/free-solid-svg-icons'

library.add(faSquare, faCheckSquare, faTasks, faTrash)
```

Notez que nous avons listé 4 icônes (faSquare, faCheckSquare, faTasks, faTrash). Il sera nécessaire de revenir en ajouter si nous souhaitons en utiliser d'autres.

💡 Font Awesome nous fournira des icônes pour :

- Les cases à cocher (tâches terminées/non terminées)
- Le bouton de suppression
- L'indicateur de progression

Vérification

À ce stade, votre application devrait toujours s'exécuter, mais avec le logo React d'origine remplacé par le nôtre. Si vous rencontrez des erreurs, vérifiez que :

- Les fichiers ont été copiés aux bons emplacements
- Les noms de fichiers correspondent exactement
- L'installation de Font Awesome s'est bien déroulée

La prochaine étape consistera à transformer notre maquette HTML en composants React.

Introduction aux composants React

Qu'est-ce qu'un composant ?

Un composant React est un bloc de code réutilisable qui :

- Encapsule une partie de l'interface utilisateur
- Peut recevoir des données (props)
- Peut avoir son propre état interne
- Peut être composé avec d'autres composants

💡 Pensez aux composants comme à des briques LEGO : chaque pièce est indépendante mais peut être assemblée avec d'autres pour créer quelque chose de plus grand.

Types de composants React

Il existe deux façons principales d'écrire des composants en React :

Composants fonctionnels (recommandé)

Un composant fonctionnel est, une fonction qui retourne du JSX. Et le JSX est une sorte d'HTML amélioré, dans lequel nous pouvons ajouter de la logique. Observez bien comment ce code est écrit.

```
function MonComposant() {
  return <div>Hello World</div>;
}
```

Composants classe (héritage)

```
class MonComposant extends React.Component {
  render() {
    return <div>Hello World</div>;
  }
}
```

💡 Nous utiliserons exclusivement des composants fonctionnels dans ce TP, car ils sont plus simples à comprendre et représentent la méthode moderne, et recommandée, de développement React.

Transformation de la maquette en React

Analyse de la structure

Avant de commencer à coder, identifions les composants logiques de notre application :

- `App` : Composant racine
  - `Header` : En-tête avec logo et compteur
  - `Form` : Formulaire d'ajout de tâches
  - `List` : Liste des tâches
    - `Item` : Une tâche individuelle (optionnel)
  - `Footer` : Pied de page

Le fichier App.jsx

1. Nettoyez d'abord le fichier `App.jsx` existant afin d'obtenir ceci:

```
import './App.css'
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome'

function App() {
  return (
    <>
      {/* Notre code viendra ici */}
    </>
  )
}

export default App
```

1. Ajoutez ensuite le code HTML de la maquette (adapté pour JSX): Modifiez uniquement la fonction App.

```
function App() {

  return (
    <>
      <header>
        <img src="/logo.svg" alt="" />
        <h1>TO DO LIST</h1>

        <div>
```

```
          <FontAwesomeIcon icon="fas fa-tasks" />
          <span>2 / 3</span>
        </div>
      </header>

      <main>
        <form>
          <input type="text" placeholder="Ajouter une tâche" />
          <button>Ajouter</button>
        </form>

        <ul>
          <li>
            <FontAwesomeIcon icon="far fa-square" size='2x' />
            <span>Trouver un job</span>
            <button><FontAwesomeIcon icon="fas fa-trash" /></button>
          </li>
          <li className="done">
            <FontAwesomeIcon icon="far fa-check-square" size='2x' />
            <span>Apprendre React.JS</span>
            <button><FontAwesomeIcon icon="fas fa-trash" /></button>
          </li>
          <li className="done">
            <FontAwesomeIcon icon="far fa-check-square" size='2x' />
            <span>Découvrir Formacitron</span>
            <button><FontAwesomeIcon icon="fas fa-trash" /></button>
          </li>
        </ul>
      </main>

      <footer>
        Copyright &copy; 2021-2025 Formacitron
      </footer>
    </>
  )
}
```

💡 Notez les différences entre HTML et JSX :

- `class` devient `className`
- Les balises sans contenu doivent être auto-fermantes (ex: `<img />` au lieu de `<img>`)
- Les attributs utilisent la casse camelCase (ex: on`Click` au lieu de on`click`)
- Les icônes sont remplacées par des composants React ex:`<FontAwesomeIcon icon="fas fa-tasks" />`

Création des composants

1. Créez un dossier `src/components` pour organiser vos composants :

```
mkdir src/components
```

1. Créez le composant Header (`src/components/Header.jsx`) :

```
import logo from '../assets/logo.svg'
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome'

function Header() {
  return (
    <header>
      <img src={logo} alt="" />
      <h1>TO DO LIST</h1>

      <div>
        <FontAwesomeIcon icon="fas fa-tasks" />
        <span>2 / 3</span>
      </div>
    </header>
  );
}

export default Header;
```

1. Créez le composant Form (`src/components/Form.jsx`) :

```
function Form() {
  return (
        <form>
          <input type="text" placeholder="Ajouter une tâche" />
```

```
          <button>Ajouter</button>
        </form>
  )
}

export default Form
```

1. Créez le composant List (`src/components/List.jsx`) :

```
import { FontAwesomeIcon } from '@fortawesome/react-fontawesome'

function List() {
  return (
    <ul>
      <li>
        <FontAwesomeIcon icon="far fa-square" size='2x' />
        <span>Trouver un job</span>
        <button><FontAwesomeIcon icon="fas fa-trash" /></button>
      </li>
      <li className="done">
        <FontAwesomeIcon icon="far fa-check-square" size='2x' />
        <span>Apprendre React.JS</span>
        <button><FontAwesomeIcon icon="fas fa-trash" /></button>
      </li>
      <li className="done">
        <FontAwesomeIcon icon="far fa-check-square" size='2x' />
        <span>Découvrir Formacitron</span>
        <button><FontAwesomeIcon icon="fas fa-trash" /></button>
      </li>
    </ul>
  )
}

export default List
```

1. Créez le composant Footer (`src/components/Footer.jsx`) :

```
function Footer() {
  return (
    <footer>
      Copyright &copy; 2021-2025 Formacitron
    </footer>
  );
}

export default Footer;
```

Assemblage des composants

Mettez à jour `App.jsx` pour utiliser vos nouveaux composants :

```
import './App.css'
import Header from './components/Header'
import React from 'react'
import Form from './components/Form'
import List from './components/List'
import Footer from './components/Footer'

function App() {

  return (
    <>
      <Header></Header>

      <main>
        <Form></Form>
        <List></List>
      </main>

      <Footer></Footer>
    </>
  )
}
```

```
export default App
```

💡 Points importants :

- Chaque composant est dans son propre fichier
- Les noms des composants commencent par une majuscule
- Les composants sont exportés par défaut
- L'import des composants utilise des chemins relatifs

Test et vérification

Votre application devrait maintenant :

1. S'afficher exactement comme la maquette
2. Être organisée en composants distincts
3. Être prête pour l'ajout de fonctionnalités

En cas de problèmes :

- Vérifiez que tous les imports sont corrects
- Assurez-vous que les chemins vers les images sont valides
- Confirmez que les styles CSS sont bien chargés
- Vérifiez la console du navigateur pour d'éventuelles erreurs

Introduction à l'état (state) en React

Qu'est-ce que l'état ?

L'état (state) en React est :

- Un objet qui contient des données qui peuvent changer au fil du temps
- Spécifique à un composant
- Capable de déclencher automatiquement le re-rendu du composant quand il change
- Géré avec le Hook `useState` dans les composants fonctionnels

💡 L'état est la mémoire de votre composant. Quand l'état change, React met automatiquement à jour l'interface utilisateur.

Le Hook useState

`useState` est une fonction fournie par React qui :

1. Crée une variable d'état
2. Fournit une fonction pour la mettre à jour
3. Déclenche le re-rendu quand la valeur change

Syntaxe de base :

```
const [valeur, setValeur] = useState(valeurInitiale);
```

Exemple concret :

```
const [compteur, setCompteur] = useState(0);
// compteur : la valeur actuelle
// setCompteur : fonction pour modifier le compteur
// 0 : valeur initiale
```

Ajout de l'état à notre application

État des tâches dans App

Commençons par ajouter l'état qui contiendra nos tâches dans le composant App :

```
import './App.css'
import Header from './components/Header'
import { useState } from 'react'
import Form from './components/Form'
import List from './components/List'
import Footer from './components/Footer'

function App() {
  // Création de l'état todos avec quelques tâches initiales
  const [todos, setTodos] = useState([
    { id: 1, description: 'Trouver un job', done: false },
    { id: 2, description: 'Apprendre React.JS', done: true },
    { id: 3, description: 'Découvrir Formacitron', done: true }
  ]);

  return (
```

```
    <>
      <Header></Header>

      <main>
        <Form></Form>
        <List></List>
      </main>

      <Footer></Footer>
    </>
  )
}

export default App
```

💡 Notes importantes :

- Chaque tâche a un identifiant unique (`id`)
- Le state est initialisé avec un tableau d'objets
- Chaque tâche a une description et un état (done)

Passage des props aux composants

Les props (propriétés) permettent de passer des données d'un composant parent à un composant enfant.

1. Mise à jour de `App.jsx` pour passer les todos au composant List: Attention: l'exemple montre la partie à modifier. Ne changez pas le reste du code !

```
function App() {
  const [todos, setTodos] = useState([
    { id: 1, description: 'Trouver un job', done: false },
    { id: 2, description: 'Apprendre React.JS', done: true },
    { id: 3, description: 'Découvrir Formacitron', done: true }
  ]);

  return (
    <>
      <Header />
      <main>
```

```
        <Form />
        <List todos={todos} />
      </main>
      <Footer />
    </>
  )
}
```

1. Mise à jour du composant List pour utiliser les todos :

```
function List({ todos }) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id} className={todo.done ? 'done' : ''}>
          <FontAwesomeIcon icon={`far fa-2x ${todo.done ? 'fa-check-square' : 'fa-s
          <span>{todo.description}</span>
          <button><FontAwesomeIcon icon="fas fa-trash" /></button>
        </li>
      ))}
    </ul>
  )
}
```

💡 Points clés :

- `todos={todos}` passe l'état au composant List
- `map` permet de transformer chaque tâche en élément JSX
- `key` est nécessaire pour aider React à gérer efficacement les listes
- L'opérateur ternaire (`?:`) permet de gérer l'affichage conditionnel

Ajout du compteur dans le Header

1. Calculons le nombre de tâches terminées dans App :

```
function App() {
  const [todos, setTodos] = useState([/*...*/]);
```

```
  // Calcul des statistiques
  const totalTodos = todos.length;
  const completedTodos = todos.filter(todo => todo.done).length;

  return (
    <>
      <Header completed={completedTodos} total={totalTodos} />
      <main>
        <Form />
        <List todos={todos} />
      </main>
      <Footer />
    </>
  )
}
```

1. Mise à jour du composant Header :

```
function Header({ completed, total }) {
  return (
    <header>
      <img src={logo} alt="" />
      <h1>TO DO LIST</h1>

      <div>
        <FontAwesomeIcon icon="fas fa-tasks" />
        <span>{completed} / {total}</span>
      </div>
    </header>
  );
}
```

Ajout de l'interactivité

Fonction pour ajouter une tâche

Dans `App.jsx`, ajoutons une fonction pour créer de nouvelles tâches, et passons la au composant Form. Lorsqu'on met à jour un state, il est nécessaire de lui fournir un nouvel objet.

```
function App() {
  const [todos, setTodos] = useState([/*...*/]);

  const addTodo = (description) => {
    // Création d'un nouvel ID (simplifié pour l'exemple)
    const newId = todos.length ? Math.max(...todos.map(t => t.id)) + 1 : 1;

    // Création de la nouvelle tâche
    const newTodo = {
      id: newId,
      description,
      done: false
    };

    // Mise à jour de l'état avec la nouvelle tâche
    setTodos([...todos, newTodo]);
  };

  return (
    <>
      <Header completed={completedTodos} total={totalTodos} />
      <main>
        <Form onAdd={addTodo} />
        <List todos={todos} />
      </main>
      <Footer />
    </>
  )
}
```

Mise à jour du formulaire

Modifions le composant Form pour gérer la saisie et l'ajout de tâches :

```jsx
import { useState } from 'react'

function Form({ onAdd }) {
  const [input, setInput] = useState('');

  const handleSubmit = (e) => {
    e.preventDefault(); // Empêche le rechargement de la page

    if (input.trim()) {
      onAdd(input.trim());
      setInput(''); // Vide le champ après ajout
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        placeholder="Ajouter une tâche"
        value={input}
        onChange={(e) => setInput(e.target.value)}
      />
      <button>Ajouter</button>
    </form>
  )
}
```

💡 Points importants :

- `useState` gère la valeur du champ de saisie
- `onChange` met à jour l'état à chaque modification du champ
- `onSubmit` gère la soumission du formulaire
- La fonction `trim()` élimine les espaces inutiles
- Le champ est vidé après l'ajout d'une tâche

Test de l'ajout de tâches

À ce stade, vous devriez pouvoir :

1. Saisir une description dans le champ texte
2. Cliquer sur “Ajouter” ou appuyer sur Entrée
3. Voir la nouvelle tâche apparaître dans la liste
4. Voir le compteur total se mettre à jour

Finalisation de l’application

Gestion des tâches terminées

Pour gérer l’état des tâches (terminées/non terminées), ajoutons une fonction `toggleTodo` dans `App.jsx` :

```
function App() {
  const [todos, setTodos] = useState([/*...*/]);

  // Le code précédent est ici...

  const toggleTodo = (id) => {
    // Copie du tableau de todos
    const updatedTodos = todos.map(todo => {
      // Si on trouve la tâche, on inverse son état
      if (todo.id === id) {
        return { ...todo, done: !todo.done };
      }
      return todo;
    });

    setTodos(updatedTodos);
  };

  return (
    <>
      <Header completed={completedTodos} total={totalTodos} />
      <main>
        <Form onAdd={addTodo} />
        <List todos={todos} onToggle={toggleTodo} />
      </main>
      <Footer />
```

```
    </>
  )
}
```

💡 Points clés :

- La fonction reçoit l'ID de la tâche à modifier
- `map` crée un nouveau tableau sans modifier l'original
- L'opérateur spread (`...`) copie toutes les propriétés de la tâche
- L'état done est inversé seulement pour la tâche ciblée

Mise à jour du composant `List` pour utiliser cette fonction :

```
function List({ todos, onToggle }) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id} className={todo.done ? 'done' : ''}>
          <FontAwesomeIcon icon={`far fa-2x ${todo.done ? 'fa-check-square' : 'fa-s
            onClick={() => onToggle(todo.id)}
        />
          <span>{todo.description}</span>
          <button><FontAwesomeIcon icon="fas fa-trash" /></button>
        </li>
      ))}
    </ul>
  )
}
```

Suppression des tâches

Ajoutons maintenant la possibilité de supprimer des tâches. Dans `App.jsx` :

```
function App() {
  const [todos, setTodos] = useState([/*...*/]);

  // Le code précédent est ici...
```

```
  const deleteTodo = (id) => {
    // Filtre les todos pour garder tous sauf celui à supprimer
    const updatedTodos = todos.filter(todo => todo.id !== id);
    setTodos(updatedTodos);
  };

  return (
    <>
      <Header completed={completedTodos} total={totalTodos} />
      <main>
        <Form onAdd={addTodo} />
        <List
          todos={todos}
          onToggle={toggleTodo}
          onDelete={deleteTodo}
        />
      </main>
      <Footer />
    </>
  )
}
```

Mise à jour du composant `List` :

```
function List({ todos, onToggle, onDelete }) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id} className={todo.done ? 'done' : ''}>
          <FontAwesomeIcon icon={`far fa-2x ${todo.done ? 'fa-check-square' : 'fa-s
            onClick={() => onToggle(todo.id)}
          />
          <span>{todo.description}</span>
          <button onClick={() => onDelete(todo.id)}>
            <FontAwesomeIcon icon="fas fa-trash" />
          </button>
        </li>
      ))}
```

```
    </ul>
  )
}
```

Bonus : Persistance avec localStorage

Introduction à localStorage

`localStorage` est une API web qui permet de :

- Stocker des données dans le navigateur
- Conserver ces données même après fermeture du navigateur
- Stocker uniquement des chaînes de caractères

💡 Parfait pour sauvegarder les todos entre les sessions !

Ajout de la persistance

Pour implémenter la persistance, nous allons utiliser le Hook `useEffect`. `useEffect` est un hook React qui permet d'exécuter du code à des moments précis dans la "vie" d'un composant. Le cycle de vie d'un composant, c'est comme son histoire : il naît (apparaît sur la page), vit (peut être mis à jour) et meurt (disparaît de la page). useEffect nous permet d'agir à chacun de ces moments.

```
import { useState, useEffect } from 'react' // Mettre à jour l'import

function App() {
  // Modifier le useState initial
  // Chargement initial depuis localStorage
  const [todos, setTodos] = useState(() => {
    const savedTodos = localStorage.getItem('todos');
    return savedTodos ? JSON.parse(savedTodos) : [
      { id: 1, description: 'Trouver un job', done: false },
      { id: 2, description: 'Apprendre React.JS', done: true },
      { id: 3, description: 'Découvrir Formacitron', done: true }
    ];
  });

  // Sauvegarde dans localStorage à chaque modification
```

```
  useEffect(() => {
    localStorage.setItem('todos', JSON.stringify(todos));
  }, [todos]);

  // ... reste du code ...
}
```

💡 Points importants :

- `useState` accepte une fonction d'initialisation
- `JSON.stringify` convertit l'objet en chaîne pour le stockage
- `JSON.parse` reconvertit la chaîne en objet
- `useEffect` s'exécute à chaque modification de `todos`

Points d'amélioration possibles

Voici quelques suggestions pour améliorer l'application :

Gestion des erreurs

- Validation des entrées utilisateur
- Gestion des cas limites (liste vide, etc.)

Conclusion

Concepts appris

- Composants React et leur cycle de vie
- Gestion de l'état avec `useState`
- Communication parent-enfant via les props
- Gestion des événements utilisateur
- Persistance des données avec `localStorage`
- Effets secondaires avec `useEffect`

Ressources pour aller plus loin

- Documentation officielle React : https://react.dev/

- React DevTools pour le débogage
- TypeScript pour un typage statique

💡 Conseil final : La meilleure façon d'apprendre React est de pratiquer ! N'hésitez pas à modifier et étendre cette application pour explorer d'autres fonctionnalités.