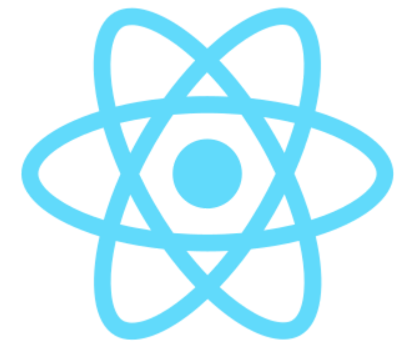
React



Deuxième partie

Rappels

React est une **bibliothèque JavaScript** pour le développement d'applications web dynamiques.

- On déclare des **composants**, responsables de **rendre** un fragment de HTML.
- On utilise la **syntaxe JSX**, qui facilite l'écriture de HTML en JS et l'utilisation d'autres composants.
- Une application React est constituée d'un **arbre de composants**.
- Les composants peuvent prendre des **valeurs en entrée** via les ***props***.
- Les composants peuvent maintenir un **état interne** grâce au ***state***.
- Les **modifications de *props* ou *state*** entraînent le **re-rendu des composants concernés**.
- Les **changements de *state* se propagent via les *props*** dans l'arbre **de haut en bas**, ce qui garantit la cohérence de l'application à tout moment.
- La **modification effective du DOM** est gérée toute seule par React via le ***Virtual DOM***.
- On peut déclarer les composants sous forme de **classes** ou de **fonctions**.

Cycle de vie des composants classes

La classe `React.Component` propose trois méthodes principales à surcharger :

- `componentDidMount()`
  - Appelée juste **après le premier *render*** : le composant a été ajouté au DOM
  - Usage : appels d'API, événements globaux, timers, etc.
- `componentDidUpdate(prevProps, prevState)`
  - Appelée juste **après chaque *render* suivant**, avec les *props*/le *state* précédents
  - Usage : appels d'API, mises-à-jour impératives, etc.
  - ⚠️ Toujours donner une condition aux modifications de state et aux appels coûteux
- `componentWillUnmount()`
  - Appelée juste **avant le retrait** du composant du DOM
  - Usage : nettoyage des actions effectuées dans `componentDidMount`

Cycle de vie, exemple

```
class Clock extends Component {
  constructor(props) {
    super(props);
    this.state = { date: new Date() };
  }

  componentDidMount() {
    // initialisation d'un timer
    this.timerID = setInterval(
      () => { this.setState({ date: new Date() }); },
      1000
    );
  }

  componentDidUpdate() {
    // action impérative après chaque render
    document.title = '🕓 ' + this.state.date.toLocaleTimeString();
  }

  componentWillUnmount() {
    // nettoyage du timer
    clearInterval(this.timerID);
  }

  render() {
    return <div>Il est {this.state.date.toLocaleTimeString()}.</div>
  }
}
```

Cycle de vie des composants fonctionnels : les *hooks*

Dans leur forme la plus simple, les composants fonctionnels sont des **fonctions pures** (props => html) :

- la sortie dépend uniquement des entrées (*props*)
- pas de modification d'état externe (effet de bord)

Mais dans certains cas, on veut :

- interagir avec un **état local**. *Ex : garder l'état ouvert/fermé d'un panneau dépliant.*
- interagir avec un **état global**. *Ex : connaître l'utilisateur connecté.*
- avoir des **effets extérieurs**. *Ex : changer le titre de la page, faire une requête à une API, etc.*
- accéder aux **éléments DOM** rendus. *Ex : obtenir la taille à l'écran d'un élément.*

Pour gérer ces cas **dans les composants fonctionnels**, React introduit la notion de ***hooks*** :

- fonctions spécialisées exécutées à chaque rendu
- permettent de "s'accrocher" à un comportement hors du rendu pur

`useState`

Permet de gérer une valeur de *state*.

```
const [stateValue, setStateValue] = useState(initialValue);
```

- retourne un tableau comprenant :
  - la **valeur courante** de l'état pour ce cycle de rendu (`stateValue`)
  - une **fonction de mise à jour** (`setStateValue`)
- prend en entrée la **valeur initiale** (`initialValue`), qui sera utilisée lors du premier cycle

Par convention, on nomme `valeur` / `setValeur` :

- `const [nbClicks, setNbClicks] = useState(0);`
- `const [date, setDate] = useState(new Date());`
- ...

On peut utiliser plusieurs `useState` dans un même composant pour diviser le *state* en unités.

Exemple `useState`

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

const StateExample = () => {
  const [nbClicks, setNbClicks] = useState(0);
  const [detailsOpen, setDetailsOpen] = useState(false);

  const toggleDetails = () => {
    setDetailsOpen(!detailsOpen);
  }

  const incNbClicks = () => {
    setNbClicks(nbClicks + 1);
  }

  return (
    <div>
      <button onClick={toggleDetails}>{detailsOpen ? '▲ Masquer' : '▼ Voir'}</button>
      {detailsOpen && (
        <div>
          <p>Nombre de clics : {nbClicks}</p>
          <button type="button" onClick={incNbClicks}>Click!</button>
        </div>
      )}
    </div>
  );
}
```

`useEffect`

Hook générique de gestion d'effet de bords.

```
useEffect(callback);
```

- Prend en argument une **fonction callback**
- Le callback va être appelé après chaque `render`.

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

const Counter = () => {
  const [nbClicks, setNbClicks] = useState(0);

  useEffect(() => { console.log(nbClicks); });

  return (
    <button type="button" onClick={() => setNbClicks(nbClicks + 1)}>Click!</button>
  );
}
```

`useEffect` avec dépendances

⚠️ En pratique, on veut souvent contrôler finement à quel moment un effet se déclenche ou non.

Ex : ré-interroger l'API seulement si le nom d'utilisateur change.

```
useEffect(callback, dependencies);
```

- `useEffect` prend en second argument un **tableau de dépendances**.
- Le callback va être appelé seulement **si les valeurs du tableau ont changé**.

```
useEffect(() => {
  // faire quelque chose seulement si user ou language a changé
  // par rapport au render précédent
}, [user, language]);
```

```
useEffect(() => {
  // faire quelque chose seulement au premier rendu
}, [])
```

Exemple `useEffect` : appel d'API

```
import { useState, useEffect } from 'react';
import axios from 'axios'; // bibliothèque d'appels HTTP qu'on va utiliser en TME

const SongLyrics = ({ artist, song }) => {
  const [lyrics, setLyrics] = useState(null);

  useEffect(() => {
    axios.get(`https://api.lyrics.ovh/v1/${artist}/${song}`)
      .then(result => {
        setLyrics(result.lyrics)
      })
      .catch(error => {
        setLyrics(null);
      });
  }, [artist, song]); // on "surveille" ces props pour refaire l'appel en cas de changement

  return (
    <div>
      <h2>{song} (by {artist})</h2>
      <pre>{lyrics || 'Not found'}</pre>
    </div>
  )
}
```

`useEffect` vs. `componentDidMount`/`DidUpdate`/`WillUnmount`

Meilleure proximité de l'initialisation et du nettoyage de ressources

```
componentDidMount() {
  this.timerID = setInterval(
    () => { this.setState({ date: new Date() }); },
    1000
  );
}

componentWillUnmount() {
  clearInterval(this.timerID);
}
```

```
useEffect(() => {
  const timerID = setInterval(
    () => { setDate(new Date()); },
    1000
  );

  return () => { // fonction de nettoyage
    clearInterval(timerID);
  }
}, []); // premier rendu seulement
```

Pas de distinction artificielle premier rendu/rendus ultérieurs

```
componentDidMount() {
  fetchUserData(this.props.userName);
}

componentDidUpdate(prevProps, prevState) {
  if (this.props.userName !== prevProps.userName) {
    fetchUserData(this.props.userName);
  }
}
```

```
useEffect(() => {
  fetchUserData(userName)
}, [userName]);
```

`useContext`

Hook de connexion à un **contexte** (état global qui ne passe pas par les *props*).

Si le composant est enfant d'un fournisseur du contexte `MyContext` :

```
const value = useContext(MyContext);
```

- Prend en entrée le type de contexte
- Retourne sa valeur courante

Mécanisme souvent utilisé pour les données transverses à toute une application :

- utilisateur connecté
- langue de l'interface
- thème d'interface
- ...

Exemple `useContext`

```
const LanguageContext = React.createContext({ lang: 'en' });

function App() {
  return <LanguageContext.Provider value={{ lang: 'fr' }}>
    <Header />
  </LanguageContext.Provider>;
}

function Header() {
  return <header>
    <Greeting />
  </header>;
}

function Greeting() {
  const { lang } = useContext(LanguageContext);
  return <div>
    {lang === 'fr' ? 'Bonjour !' : 'Hello!' }
  </div>;
}
```

`useRef`

Hook de stockage de valeur persistante entre cycles de rendu.

```
const myRef = useRef(null);
```

- Prend en entrée la valeur initiale
- la valeur est accessible en lecture et écriture via `current` : `myRef.current`

Assimilable à une propriété d'instance dans une classe (`this.foobar`)

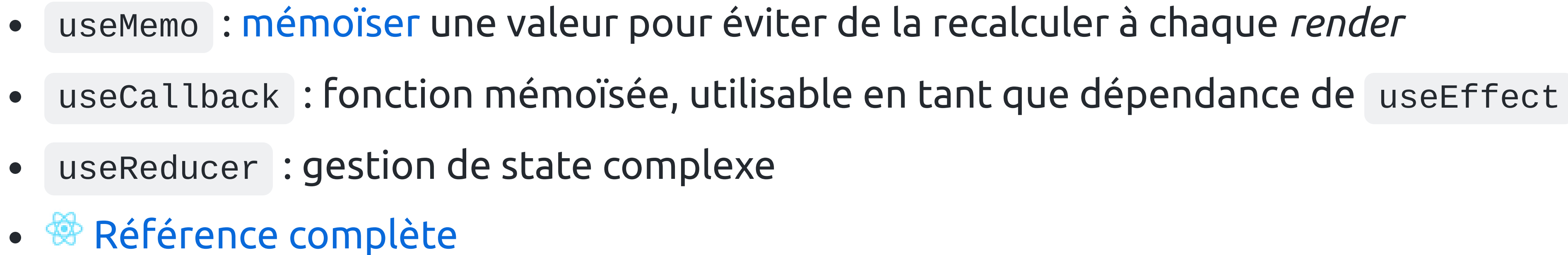
Principal usage : passé à l'attribut `ref` pour obtenir une référence directe vers un élément DOM rendu

```
const RefToDOM = () => {
  const domElt = useRef(null);

  return <div ref={domElt}>...</div>;
}
```

Exemple `useRef`

```
import { useEffect, useRef } from 'react'

const Measure = () => {
  const domElt = useRef(null);

  useEffect(() => {
    const onResize = () => {
      // domElt.current contient une référence vers le vrai nœud DOM de la div
      const width = domElt.current.clientWidth;
      console.log(width);
    }

    window.addEventListener('resize', onResize)

    return () => {
      window.removeEventListener('resize', onResize)
    }
  }, [])

  return <div ref={domElt} style={{ border: '1px solid', height: 50 }}></div>
}
```

Autres hooks



- `useMemo` : mémoïser une valeur pour éviter de la recalculer à chaque *render*
- `useCallback` : fonction mémoïsée, utilisable en tant que dépendance de `useEffect`
- `useReducer` : gestion de state complexe
- Référence complète

On peut aussi **définir ses propres hooks**, par composition des autres (on les nommera aussi `use*`).

```
function useCurrentUser() {
  const [user, setUser] = useState(null)
  useEffect(() => {
    // code complexe avec appel réseau, lecture de cookies, authentification, contexte, etc.
    // setUser(…)
  }, []);
  return user;
}
// Dans un composant : const currentUser = useCurrentUser();
```

Mécanisme très utilisé dans les bibliothèques tierces pour React.

Formulaires

> ⚠️ Rappel : en HTML, les champs de formulaires sont *stateful* : ils ont un état interne.

Deux approches en React :

Non contrôlé

- Utilisation de l'état interne de l'élément HTML
- Lecture de la valeur quand c'est nécessaire
- Nécessite une référence explicite sur l'élément DOM (via `useRef` par exemple)

Contrôlé

- Gestion de la valeur en tant que *state* React
  "Source unique de vérité"
- Mise à jour via le cycle de rendu de React
- Plus lourd (*state* + événements)
- Permet des opérations à la volée
  Ex. : formatage ou validation

https://fr.reactjs.org/docs/forms.html

Formulaire non contrôlé

```
const onlyNumbers = (str) => str.split('').every((char) => Number.isInteger(parseInt(char)))

const UncontrolledForm = () => {
  const cardNumberRef = useRef(null)
  const [hasInputError, setHasInputError] = useState(false)

  const handleSubmit = (e) => {
    e.preventDefault();
    // lecture de l'état interne de l'élément DOM pointé par cardNumberRef
    const cardNumber = cardNumberRef.current.value;
    setHasInputError(!onlyNumbers(cardNumber))
  }

  return (
    <form onSubmit={handleSubmit}>
      <label htmlFor="cardNumber">Numéro de carte : </label>
      <input id="cardNumber" ref={cardNumberRef} />
      {hasInputError && <div>Entrée incorrecte !</div>}
    </form>
  )
}
```

Formulaire contrôlé

```
const groupBy4 = (str) => (str.match(/.{1,4}/g) || []).join(' ')
const onlyNumbers = (str) => str.split('').every((char) => Number.isInteger(parseInt(char)))

const ControlledForm = () => {
  const [cardNumber, setCardNumber] = useState('')
  const [hasInputError, setHasInputError] = useState(false)

  const handleChange = (e) => {
    const cleanValue = e.target.value.replace(/ /g, '')
    setCardNumber(groupBy4(cleanValue.substring(0, 16)))

    const isInvalid = !onlyNumbers(cleanValue)
    setHasInputError(isInvalid)
  }

  return (
    <div>
      <label htmlFor="cardNumber">Numéro de carte : </label>
      {/* Un champ contrôlé a les attributs value et onChange renseignés */}
      <input id="cardNumber" onChange={handleChange} value={cardNumber} />
      {hasInputError && <div>Entrée incorrecte !</div>}
    </div>
  )
}
```

React côté serveur

On peut rendre les mêmes composants React côté serveur, en Node.js :

```
import ReactDOMServer from 'react-dom/server';

// pas d'accrochage à un DOM
const htmlString = ReactDOMServer.renderToStaticMarkup(`<MyComponent />`);
```

Le rendu sera **statique**, car **limité au premier cycle de *render*** (valeurs de *state* initiales).

Pour une SPA, on peut faire le **premier rendu côté serveur**…
… et **laisser React "reprendre la main" côté client** dès qu'il est chargé.

On parle alors de **SSR** (Server Side Rendering).

Deux raisons principales :

- Expérience utilisateur : page HTML complète avant chargement du JS par le navigateur.
- SEO (Search Engine Optimization) : page HTML complète pour les moteurs de recherche.

Autour de React

Validation de types

On peut utiliser le package prop-types :

- documentation des *props* d'un composant
- affichage de warnings à l'exécution si props non conformes

```
import PropTypes from 'prop-types';

const Header = ({ name, friends, nbNotifications }) => {
  return <header>…</header>
}

Header.propTypes = {
  userName : PropTypes.string.isRequired,
  friends: PropTypes.arrayOf(PropTypes.string),
  nbNotifications: PropTypes.number
}

Header.defaultProps = {
  friends: []
}
```

On peut aussi utiliser directement TypeScript pour du typage fort.

Redux

Gestionnaire d'état pour les applications web (pas forcément React)

Souvent utilisé dans les applications React complexes : https://react-redux.js.org/

Principes :

- **store** : state global, souvent à la racine de l'application
- **actions** : opérations possibles sur ce store (ex : ADD_FRIEND, REMOVE_MESSAGE, etc.)
- **reducers** : fonctions de calcul du nouvel état du *store* en fonction d'une action
- Les composants peuvent se **connecter** au store pour en recevoir des parties en *props*.
- Les composants peuvent **envoyer** des actions au store (*dispatch*).

En React moderne, on peut implémenter une majorité de ce modèle avec les contextes et les *hooks* (`useContext`, `useReducer`)...

React Router

https://v5.reactrouter.com/web Router déclaratif pour les SPA React

Principe : gérer la structure d'URLs et les différentes vues de l'application en React pur

```
export default function App() {
  return (
    <Router>
      <div>
        <nav>
          <Link to="/">Home</Link> · <Link to="/about">About</Link> · <Link to="/users">Users</Link>
        </nav>
        <Switch>
          <Route path="/about">
            <About />
          </Route>
          <Route path="/users">
            <Users />
          </Route>
          <Route path="/">
            <Home />
          </Route>
        </Switch>
      </div>
    </Router>
  );
}
```

React Native

https://reactnative.dev/

Framework de développement d'applications natives, également maintenu par Facebook.

- Même modèle conceptuel que React pour le web : *props/state* et propagation
- Même syntaxe JSX
- On appelle des composants visuels natifs au lieu des éléments HTML
- Principe de couche d'abstraction du rendu, similaire au VirtualDOM

Permet de cibler Android et iOS avec un même code JS de plus haut niveau.