React

Introduction

JSX est un langage transpilé en javascript (par Babel).
Il faut exporter les composants

```
function Component() { ... }
export default Component;
```

On peut facilement initialiser une application react

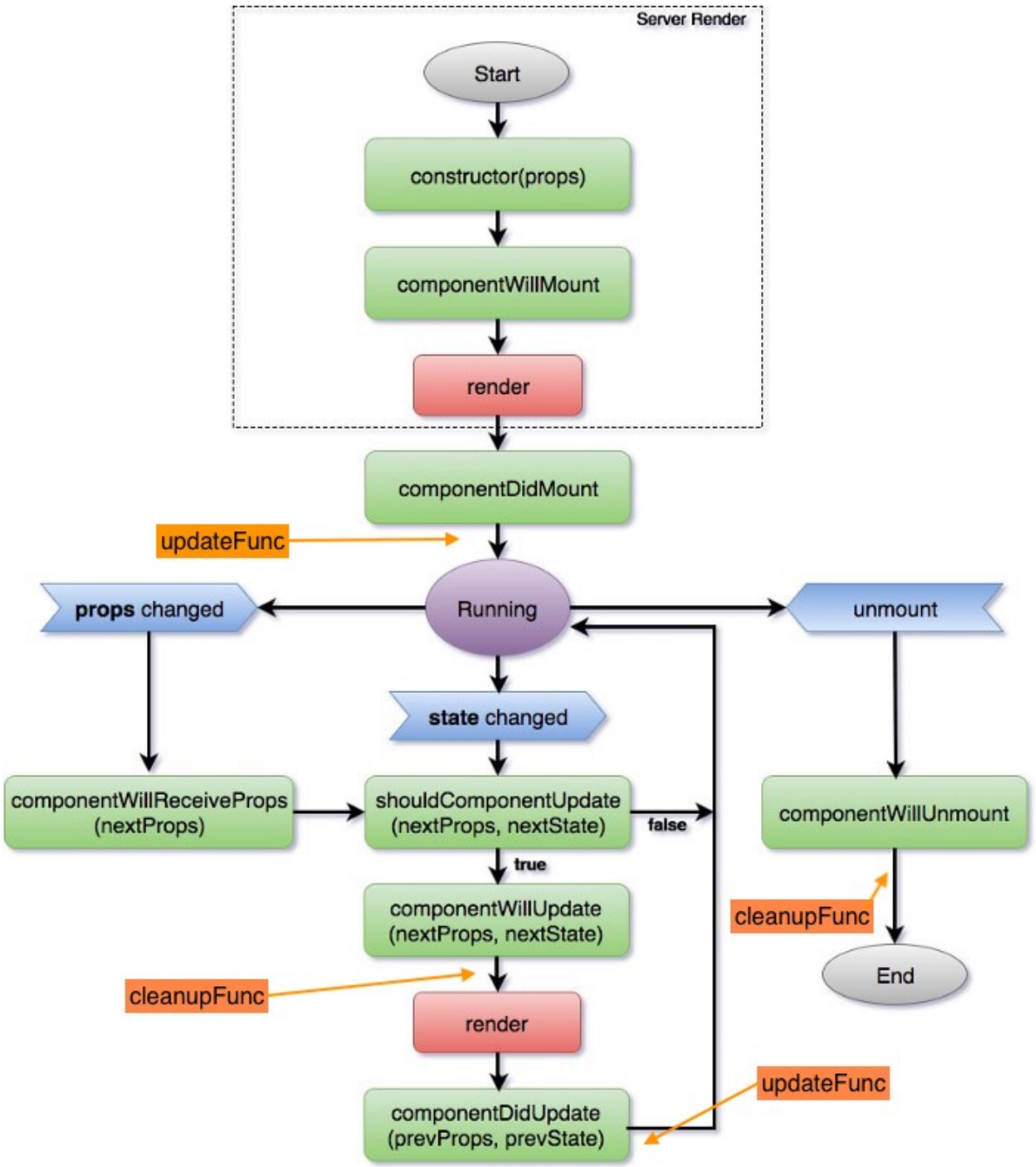
```
npx create-react-app my-app
// npm install react-scripts@latest
```

Render

C'est la fonction la plus importante, elle dit comment dessiner le composant (Nb: un composant fonction est implicitement directement la fonction `render`)

Si elle retourne `null`, alors le composant sera complètement invisible met fera malgré tout les calculs dont il a besoin.

Lifecycle



Nb: les blocs en verts sont des functions qu'on peut surcharger dans le composant

Composant

Classe

```
import React from 'react';

class Welcome extends React.Component {
  // On va devoir transformer les props en state:
  // les props ne seront plus disponibles dans la fonction `render`
  constructor(props) {
    super(props);
    // Don't call this.setState() here!
    this.state = { counter: 0 }; // si on fait this.setState on va tout casser
    this.handleClick = this.handleClick.bind(this);
  }
  // Function custom, Nb: elle n'est pas `bind` au composant, il faut le faire
manuellement dans le constructeur
  handleClick() {
      // Do something
  }

  // Manage un des états
  componentDidUpdate(prevProps, prevState, snapshot) {
    // is invoked immediately after updating occurs
  }

  // Seule fonction obligatoire
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

- + verbeux pour les cas simples
- + simple pour les cas complexes
  e.g. si on veut fetch des données quand le composant est créé.

Fonctions

```
import { useState, useEffect /*...*/ } from "react";

function Welcome({name, value}) {  // Nb: utilisation de la destructuration
    // Cette fonction est comme `render` avec un accès direct aux props du
constructeur

    // Arrow function: Cela évite de devoir faire les `bind`
    const handleClick = () => {
      // Do something
    }
    return return <button onClick={handleClick}>Hello, { name }</button>;

}
```

- + simple dans la majorité des cas et souvent mieux optimisée

- Il faut utiliser des hook `useXXX` pour compenser la gestion des lifecycles des composants classe

useXXX: Fonction vs Classes

useState

Permet d'avoir un état interne qui persiste entre chaque re-render. Ca compense le `setState`

```
function Welcome() {
    const [counter, setCounter] = useState(0) // prend la valeur initiale en
paramètre
    const incrementCounter = () => {
      setCounter(counter + 1);
    };
    return <button onClick={}>{counter}</div>;

}
```

Nb: Il ne faut **JAMAIS** modifier directement la valeur (ici `counter`). Il faut toujours:

- (idéalement) travailler sur une copie
- (obligatoire) finir la modification par assigner la valeur avec le setter (ici `setCounter`)

Quand on utilise `setCounter`, le composant entier est redessiné.
Toute valeur qui n'est **PAS** un `useState` (ou `useCallback` ou `useRef`) sera perdue

useRef

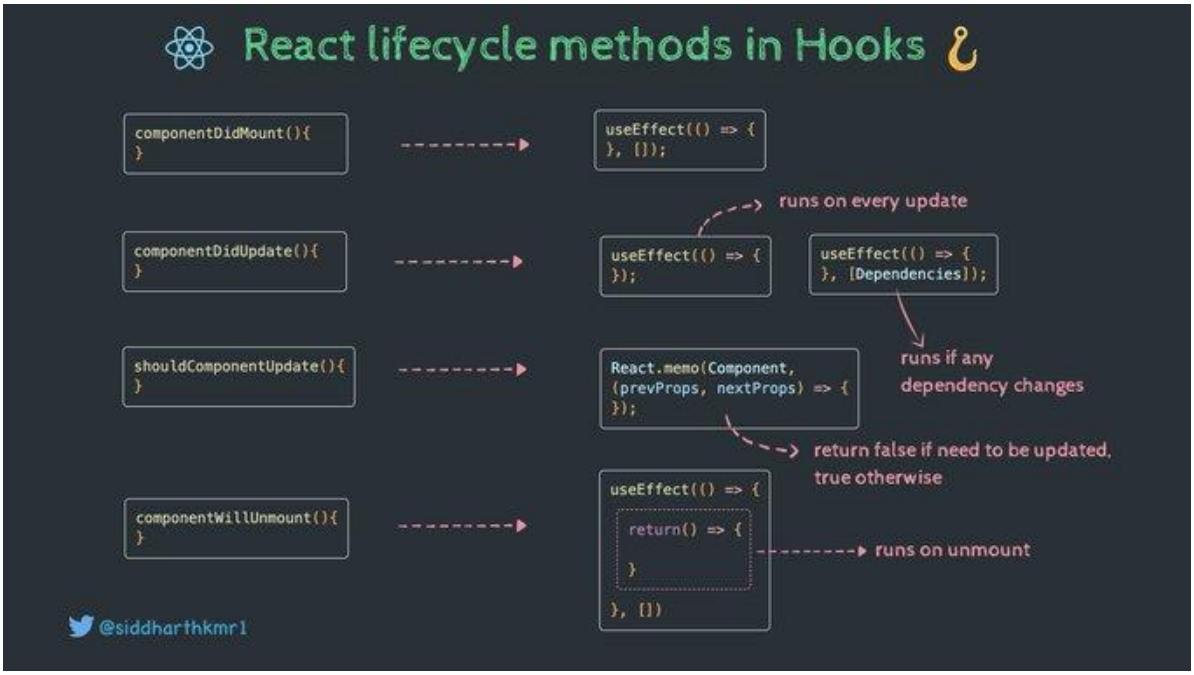
Comme `useState`, sauf que:

- Change la valeur ne va pas redessiner le composant
- Il n'y a donc de getter/setter mais juste 1 valeur

```
const counter = useRef(0);
counter.current += 1;
```

useEffect (utile pour fetch des données)

Cela compense `componentDidMount`, `componentDidUpdate` et `componentWillUnmount`




```
function Welcome({name}) {
    useEffect(
        // 1er paramètre: fonction a executer avant de mount le composant et
après chaque update
        () => {
            // Do something after mount/update
            music.play()

            // On peut retourner une fonction, elle sera exécutée avant de
détruire le composant
            return () => {
                music.stop()
            }
        },
        [name], // 2e paramètre: les triggers
                // (i.e. valeur qui, si modifiée, redéclenche l'appel à la
fonction du 1er paramètre)
    )
    return <div>{name}</div>;

}
```

useReducer (peu utilisé)

Comme `useState` sauf que la valeur passée au setter n'est pas directement assignée mais passe d'abord dans une fonction

```
const [name, setName] = useState("");
const onNameChange = (newName) => {
    setName("User: " + newName);
}
```

ici, on veut toujours que `name` commence par `User:`, donc on devrait à chaque fois recopier le code ci-dessus.

```
const [name, setName] = useReducer(
    (newName) => {return `"User: ${newName}`}, // Cette fonction est appelé à chaque appel de setName
    "" // Comme useState: valeur par défaut
);
const onNameChange = (newName) => {
    setName(newName);
}
```

useCallback (peu utilisé: optimisation)

C'est juste pour optimiser les fonctions

```
function Welcome({name, value}) {
    // Cette fonction est refaite à chaque fois que le composant est re-dessiné
    const handleClick = () => {console.log(value)}
    return return <button onClick={handleClick}>Hello, { name }</button>;

}
```

Pour optimiser

```
function Welcome({name, value}) {
    const handleClick = useCallback(
        () => {console.log(value)},
        // 2e paramètre (optionel): trigger pour recalculer la fonction
        [value] // Cette fonction doit être recalculée si value change
    )
    return <button onClick={handleClick}>Hello, { name }</button>;

}
```

useMemo (peu utilisé: optimisation)

Quand une valeur est lourde à recalculer on veut la stocker et la recalculer que au besoin

```
const heavyToCompute = useMemo(
    () => {
        /* Heavy computation*/
        // return computedValue;
    },
    [trigger1, trigger2] // Valeur qui, si changée, doivent re-déclencher le calcul
)
```

Nb: On ne peut pas directement assigner la variable `heavyToCompute`

createRef

Comme `useRef` mais la référence est re-créée à chaque fois.
Le but est par exemple de référencer un élément du jsx

```
function Welcome({name, value}) {
    let myInput = createRef();
    const printInputValue = () => {
        /*
            `myInput` est juste un wrapper sur la reference
            pour obtenir la valeur interne il faut utiliser `.current`
            ici, la valeur interne est un objet html `HTMLInputElement`,
              Si on veut sa valeur, on doit utiliser `.value`
        */
        console.log(myInput.current.value)
    }
    return <div>
        <button onClick={printInputValue}>print input value</button>
        <input ref={myInput}></input>
    </div>;

}
```

Nb: la valeur référencée ici est un HTMLInputElement, si on veut accéder à la valeur actuelle de l'input il faut utiliser l'attribute `.value`

Routeur

Le composant `<Routeur>` doit englober les autres composants du routeur (E.g. `<Link>`, `<Switch>`, ...) pour que ces derniers marches.
On met donc généralement le composant racine de l'application

```
import React from "react";
import {
  BrowserRouter as Router,
  Switch,
  Route,
  Link,
  useRouteMatch,
  useParams
} from "react-router-dom";

// Composant racine
export default function App() {
  return (
    <Router>
      {/* Contenu réel du composant */}
      <h1> Welcome </h1>
    </Router>
  );
}
```

On peut afficher des composants différents en fonction de la route active

```
function topics() {
  return <>
    <Switch>
      {/* Si un paramètre est fourni, on va appeler le composant <Topic> */}
      <Route path={`${match.path}/:topicId`}>
        <Topic />
      </Route>
      {/* Sinon on afficher le composant que l'on veut */}
      <Route path={match.path}>
        <h3>Please select a topic.</h3>
      </Route>
    </Switch>
  </>
}

// Le paramètre de route sont fournis avec useParams
function Topic() {
  let { topicId } = useParams();
  return <h3>Requested topic ID: {topicId}</h3>;
}
```

useParams

Permet d'accéder aux paramètres de la route.
=> son utilisation est donc très **contextuel**, le composant ne pourra **PLUS** être utilisé sans le routeur

useRouteMatch

permet de créer des matchs de route

```
let match = useRouteMatch();
return <Link to={`${match.url}/components`}>Components</Link>
```

Nb: `<Link>` correspond a une balise `<a>`

Parent - Enfant "Lifting States Up" (Alternative au routeur ~)

On peut gérer un état sur le composant parent et alterner entre les enfants

```
function ParentComponent() {
    const [value, setValue] = useState(true);
    if(value) {
        return <ChildA setValue={setValue}/>;
    }
    return <ChildB setValue={setValue}/>;
}
```

Nb: il faut éviter les `if` pour le rendu (lisibilité et bug) et favoriser le conditionnel dans le rendu

```
function ParentComponent({value}) {
    const [value, setValue] = useState(true);
    return { cond && <ChildA  setValue={setValue}/> || <ChildB setValue=
{setValue}/> };
}
```

Nb: la props `children` donne accès aux composants enfants

Context

C'est un moyen de partager des variables entre plusieurs composants

Déclaration

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

// On déclare un context
export const UserContext = createContext();

// On déclare un composant `UserProvider` qui contient un contexte
export const UserProvider = ({ children }) => {
  const [user, setUser] = useState({ name: 'John' });
  const setName = name => setUser({ ...user, name });
  return (
    {/* On utilise le contexte UserContext et on lui passe des valeurs */}
    <UserContext.Provider value={[user, setName]}>
      {children}
    </UserContext.Provider>
  );
}
```

Utilisation

Initialisation

Il faut le déclarer une fois pour que les composants enfants puissent y accèder

```
return (
    <UserContext>
        {/* Mettre ici les enfants devant partager le context */}
    </UserContext>
)
```

On va souvent l'initialiser autour de toute l'application

```
import { UserProvider } from './context/UserContext';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <UserProvider>
    <App />
  </UserProvider>
);
```

Utilisation

Pour qu'un composant accède au contexte, il faut utiliser `useContext` (peu importe qu'on ait un composant de type classe ou fonction)

```
import { UserContext } from './context/UserContext';
function User() {
  const value = useContext(UserContext); // c'est ce qu'on a définit dans
`value=` sur le contexte
  // On peut utiliser la destructuration directement
  // const [user, setName] = useContext(UserContext);

  return ... ;
}
```