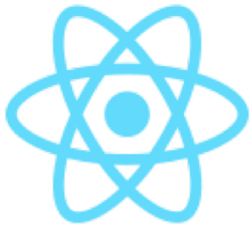


https://reactjs.org/

J.-M. Pecatte

Dept MMI – IUT Castres

jean-marie.pecatte@iut-tlse3.fr

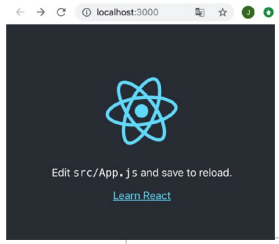
React

- React (aussi appelé React.js ou ReactJS) est une bibliothèque JavaScript libre développée par Facebook depuis 2013. https://reactjs.org/
- Son but principal est de faciliter la création d'application web monopage, via la création de composants dépendant d'un état et générant une page (ou portion) HTML à chaque changement d'état.
- La bibliothèque est utilisée notamment par **Netflix**, **Yahoo**, **Airbnb**, **Sony**, **Atlassian** ainsi que par les équipes de Facebook

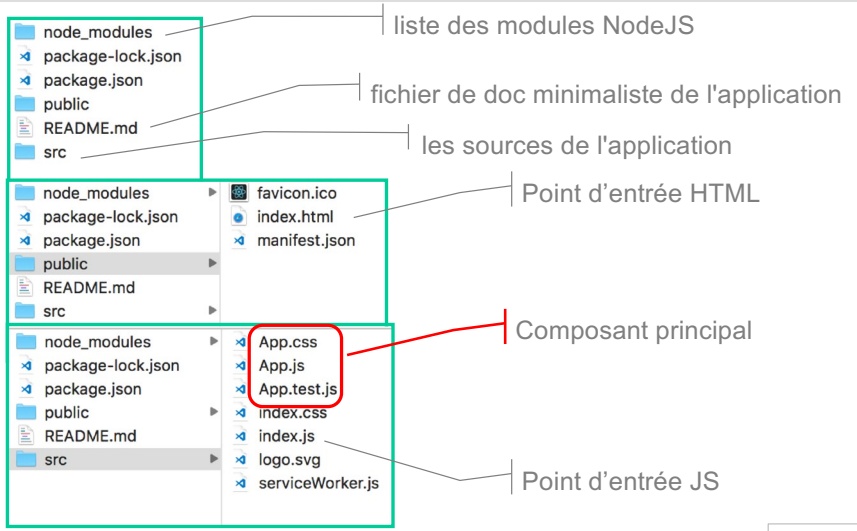
https://fr.wikipedia.org/wiki/React_(JavaScript)

React : installation

- Installer NodeJS et NPM
- Installer React : npm install -g create-react-app
- Créer une première application :
  npx create-react-app my-app
- Se placer dans le dossier : cd my-app
- Pour exécuter cette première application : npm start
  l'application est accessible dans un navigateur via un serveur web local



React : structure des dossiers



Archi de base

index.html

```
<body>
<div id="root"></div>
</body>
```

Afficher le composant <App /> dans l'élément d'id root

index.js

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

ReactDOM.render(<App />, document.getElementById('root'));
```

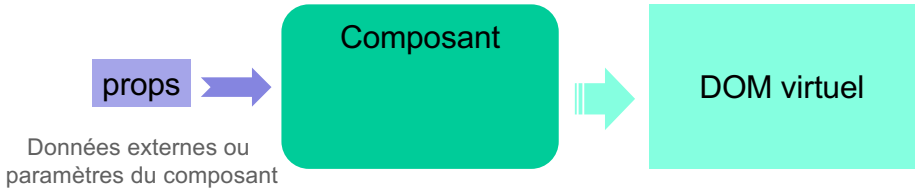
app.js

```
import React from 'react';
function App() {
  return (
    <p>Bonjour</p>
  );
}
export default App;
```

le composant <App />

Les composants

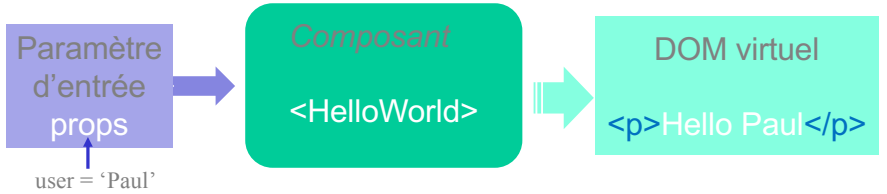
Composant de base (sans état)



Données externes ou paramètres du composant

React détecte les changements des props et met à jour le DOM virtuel

Composant de base : exemple




```
<HelloWorld user='Paul' />   =>  <p> Hello Paul </p>
<HelloWorld user='Jean' />   =>  <p> Hello Jean </p>
```

Le composant HelloWorld accepte un attribut (paramètre) nommé user qui est le prénom de la personne à saluer
Un composant React a un seul point d'entrée : l'objet nommé props ;
React lui affectera automatiquement une propriété user ;
→ props.user contiendra la valeur donnée en entrée

2 types de composant

- Composant fonctionnel *(function component) :*
  - composant déclaré comme une fonction Javascript
  - cette fonction a un seul paramètre, l'objet props
  - elle renvoie une vue à afficher en retour.

- Composant de type classe *(class component)* :
  - composant déclaré comme une classe (class base)
  - plus riche que les composants fonctionnels
  - la méthode render() est obligatoire, elle définit la vue à afficher.

Composant fonctionnel

- Il est décrit par une fonction pure (dont le résultat ne dépend que de ses paramètres)
- Le (seul) paramètre est appelée props ; il est immuable (non modifiable) ; il permet de récupérer les attributs.
- Le résultat *(le rendu)* est au format JSX

```
import React from 'react';
function HelloWorldF(props) {
    // props est de la forme { user : XXX }
    let user = props.user;
    return <p>Hello {user}</p>
};
export default HelloWorldF;
```

- Utilisation du composant avec un paramètre

```
<HelloWorldF user='Paul' />
```


Composant de type classe

- Il est décrit par une classe héritée de la classe Component
- Le constructeur a un paramètre appelé props qui permet d'ajouter à la classe une propriété du même nom
- La méthode render() obligatoire sert à définir le résultat *(le rendu)* au format JSX

```
import React from "react";
class HelloWorldC extends React.Component {
  render() {
    let user = this.props.user; // this.props pour accéder à la propriété
    return <p>Hello {user}</p>;
  }
}
export default HelloWorldC;
```

La classe doit implémenter au minimum la méthode **render()** qui définit le rendu du composant

- Utilisation du composant

```
<HelloWorldC user='Paul' />
```


La props children

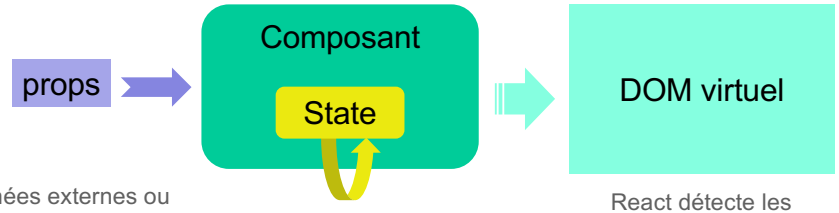
- Une propriété particulière est la props.children ; elle est aussi immuable (non modifiable) ; elle permet de récupérer le « contenu » du composant (ce qu'il y a entre les balises ouvrantes et fermantes)
- Le résultat *(le rendu)* est au format JSX

```
import React from 'react';
function HelloNomPrenom(props) {
    return <p>Bonjour {props.prenom} {props.children}</p>
};
export default HelloNomPrenom;
```

- Utilisation du composant avec le paramètre 'children'

```
<HelloNomPrenom prenom="Jean">Dupont</HelloNomPrenom>
```

Composant à état



Données externes ou paramètres du composant

Données internes du composant

React détecte les changements du state ou des props et met à jour le DOM virtuel

➔Version React < 16.8 => possible uniquement avec des « class component »

➔ Version React >= 16.8 => introduction des « hooks »
possible de gérer le state avec des « function component »

Composant à état de type class

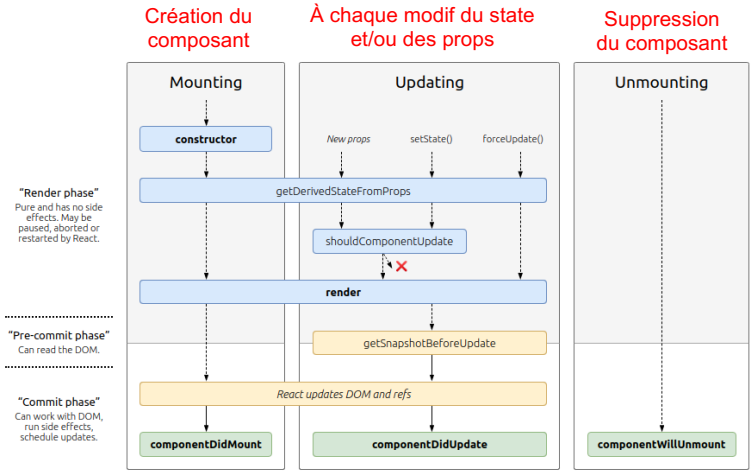
- composant déclaré sous la forme d'une classe (class base) qui hérite de la classe Component de React

```
import React from 'react';

class App extends React.Component {
   state = { … };

   render() {
      return (
         <div>
            …
         </div>
      );
   }
}
export default App;
```

L'état du composant est représenté par une propriété qui doit s'appeler obligatoirement state

Composant à état

- Un composant React est géré selon un cycle



Composant à état

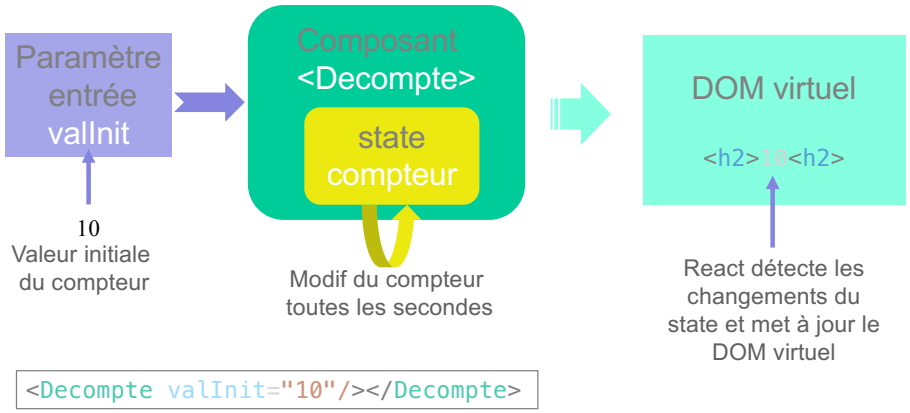
- Dans le cycle précédent, la méthode componentDidUpdate est déclenchée dès que les props et/ou le state sont modifiés
- Les props sont forcément modifiées par le composant père car un composant ne peut pas modifier les props (ses paramètres)
- L'état doit forcément être modifié par setState()
- Cette méthode accepte deux paramètres qui permettent d'avoir accès à l'ancienne valeur des props et à l'ancienne valeur du state

componentDidUpdate(prevProps, prevState)

Cela permet de savoir par exemple si c'est le state ou les props qui ont été modifiés

Composant à état : un exemple

un composant qui affiche un décompte d'une valeur initiale fixée jusqu'à 0 (-1 ttes les secondes)




```
<Decompte valInit="10"/></Decompte>
```


Composant à état : un exemple

- composant qui affiche un décompte d'une valeur initiale fixée jusqu'à 0 (-1 ttes les secondes)

L'état du composant (objet state) contient le **compteur**

Le composant affiche la valeur du **compteur**

```
import React from 'react'

class Decompte extends React.Component {
    state = { compteur : this.props.valInit }

    render() {
        return (
            <h2>Compteur : {this.state.compteur}</h2>
        )
    }
}
export default Decompte
```


Composant à état : un exemple

- Pour le moment, le compteur n'évolue pas ; il faut changer sa valeur toutes les secondes (utilisation de la fonction JS setInterval)
- Il faut lancer l'évolution du compteur juste après que le compteur se soit affiché pour la première fois => méthode componentDidMount()

La fonction setInterval() appelle la méthode moinsUn toutes les 1000 ms (1s)

```
componentDidMount() {
        this.timerID = setInterval( this.moinsUn, 1000);
}
moinsUn = () => {
        this.setState({compteur: this.state.compteur - 1})
}
```

On modifie l'état du composant avec la méthode setState (OBLIGATOIRE)

Composant à état : un exemple

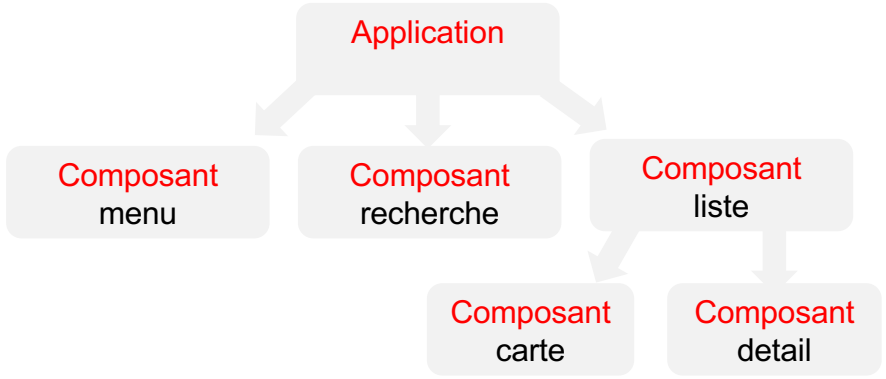
- Dès que l'état (state) est modifié, React se charge d'actualiser automatiquement l'affichage (le rendu) => cycle Updating => appel de la méthode render()
- Maintenant il faut arrêter le décompte quand le compteur est à 0
- Pour cela, dans le cycle Updating, on va surcharger la méthode componentDidUpdate()

```
componentDidUpdate() {
        if (this.state.compteur==0)
                clearInterval(this.timerID);
}
```

La fonction clearInterval() permet d'arrêter le timer donc l'évolution du compteur

React : anatomie -> composants

- Une application est un arbre de composants



JSX

JSX : JavaScript Syntax eXtension

- Le JSX permet de décrire *(de manière plus visuelle pour le développeur)* le résultat des composants c'est-à-dire ce qui doit s'afficher à l'écran *(le rendu)*
- C'est un langage à balise inspiré du XML *(mais ce n'est pas du HTML même si c'est proche)*
  -> par exemple tous les éléments doivent être fermés comme en XML
- La mise en forme utilise des styles *(mais ce n'est pas du CSS même si c'est proche)*
- Il est possible d'y insérer des expressions JS entre {} => mais pas d'instructions JS *(pas if, pas de for, …)*
- Un attribut peut contenir une valeur mais uniquement de 2 types : un chaine de caractères entre " " ou une expression entre { }

JSX

- L'utilisation de JSX dans React n'est pas obligatoire

```
const element = (
   <h1 className="greeting">
   Hello, world!
   </h1>
);
```

JSX

Compilateur BABEL

```
const element = React.createElement(
   'h1',
   {className: 'greeting'},
   'Hello, world!'
);
```

Javascript

JSX : des exemples

- Insertion de variables

```
const user = {
    prenom: 'Jean',
    nom: 'Dupont',
};
const element = <h1>Hello, {user.prenom} {user.nom}!</h1>;
```

- Appel de fonctions

```
function formatName(user) {
      return user.prenom + ' ' + user.nom;
}
const element = <h1>Hello, {formatName(user)} !</h1>;
```


JSX : nommage

- Par convention, les noms de composant commencent toujours par une lettre en majuscule, la minuscule est réservée au HTML
- React utilise la convention de dénomination de la propriété camelCase au lieu des noms d'attribut HTML.
  class devient className et tabindex devient tabIndex.
- Il est possible d'utiliser des attributs mais la valeur est soit une chaîne entre "" soit une expression entre { }

```
const element = <div tabIndex="0"></div>
const element = <img src={user.avatar}></img>
```


JSX : Fragment

- Le rendu en JSX doit toujours comporter un élément englobant
- Il est possible d'utiliser tout élément html sémantiquement acceptable (div, span, …)
- Mais si cet élément n'a pas d'interet pour le rendu final, React propose d'utiliser le composant Fragment

```
return (
  <div>
    <Hello />
    <HelloP prenom="Jean"/>
  </div>
);
```

 sera dans le rendu final

```
return (
  <Fragment>
    <Hello />
    <HelloP prenom="Jean"/>
  </Fragment>
);
```

<Fragment> ne sera pas dans le rendu final

JSX : mise en forme

- inline attribut style = {{ "prop1" : val1, "prop2" : val2, …}}
  - Les noms des propriétés des styles JSX sont proches de celles de CSS mais l'utilisation du – est remplacé par la notation camelCase *Ex : background-color => backgroundColor*

```
return (
<p style={{ backgroundColor:'black', color: 'white' }} >Hello</p>
)
```

```
return (
<p style={styles.blackWhite}>Hello</p>
)
const styles = {
  blackWhite : { backgroundColor:'black',
    color: 'white'
  }
}
```

JSX : événements

- La gestion d'événements avec des éléments React est très similaire à la gestion d'événements sur des éléments DOM. Il y a quelques différences syntaxiques:

  Les événements React sont nommés en utilisant camelCase, plutôt qu'en minuscule.

  Avec JSX, vous transmettez une fonction en tant que gestionnaire d'événements plutôt qu'une chaîne.

```
<button onclick="activateLasers()">
    Activate Lasers
</button>
```

```
<button onClick={activateLasers}>
    Activate Lasers
</button>
```


JSX : événements

- Le fonctionnement par défaut ne peut être annulé qu'à l'aide de la méthode preventDefault()

```
function gestionClick(e) {
      e.preventDefault();
      console.log('le lien est cliqué.');
}
<a href="#" onClick={gestionClick}>
Cliquez moi
</a>
```


JSX : événements

- Exemple intégré dans un composant : *bouton bascule* ON / OFF

```
class Toggle extends React.Component {
   constructor(props) {
      super(props);
      this.state = {isOn: true};
   }
   handleClick = (e) => {
      this.setState({isOn: !this.state.isOn});
   }
   render() {
      return (
         <button onClick={this.handleClick}>
            {this.state.isOn ? 'ON' : 'OFF'}
         </button>
      );
   }
}
```


JSX : événements

- Comment passer d'autres paramètres à la fonction de gestionnaire d'événements ?

  → en passant par une fonction flèche

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
```

composant avec 2 rendus possibles (2 composants)

- Objectif : créer un composant qui affiche un des 2 boutons (chacun étant un composant) Login / Logout

➔Les 2 boutons = 2 composants

```
function LoginButton(props) {
  return (
    <button
     onClick={props.onClick}>
     Login
    </button>
  );
}
```

```
function LogoutButton(props) {
  return (
    <button
       onClick={props.onClick}>
       Logout
    </button>
  );
}
```

Remarque : grâce aux "props" le gestionnaire d'événement est externe au composant

composant avec 2 rendus possibles (2 composants)

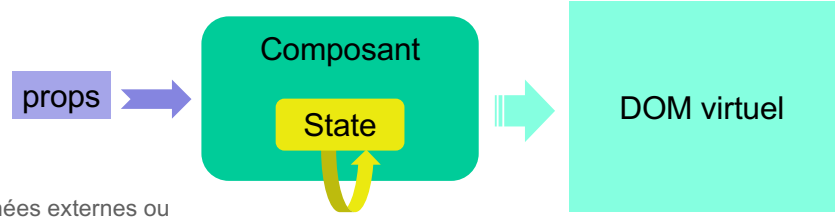
- Affichage alternatif => conditionnelle if () { } else { }
- Problème : pas possible en JSX car il n'accepte pas les instructions mais uniquement les expressions
  => utilisation d'une ternaire : *cond ? expr1 : expr2*

```
render() {
   return (
      <div>
       { this.state.isLoggedIn ? (
           <LogoutButton onClick={this.handleLogoutClick} />
         ) : (
           <LoginButton onClick={this.handleLoginClick} />
         )
       }
      </div>
   );
}
```


Communication entre composants

Props, children, méthodes,…

Un composant



Données externes ou paramètres du composant

Données internes du composant

React détecte les changements du state et/ou des props puis met à jour le DOM virtuel

Props & children

- En React tout composant E présent dans le render d'un composant P est considéré comme enfant de P
- P est le parent, E est l'enfant
- P peut avoir plusieurs enfants
- P peut transmettre des données à ses enfants en utilisant les props
- La transmission ne peut se faire que de P → E
- Syntaxiquement, la transmission se fait à l'aide d'attributs (au sens HTML)
- Une prop particulière nommée children du composant P est un tableau contenant la liste des composants fils

Liste

List & keys

- Il est souvent nécessaire de générer une liste d'éléments HTML pour représenter une liste de données
- Comment faire un rendu en JSX sachant qu'il n'y a pas de boucle for ➔ utilisation de map()

Exemple : affichage d'une tableau de produit dans une liste :

```
// composant avec en paramètre le tableau de produits à afficher
// en entrée par les props (propriété liste)
import React from 'react';
function ItemList(props) {
  return (
    <ul> { props.liste.map(
            (v) => <li key={v.id}>{v.qte} {v.article}</li>
          )} </ul>
  )
};
export default ItemList;
```


List & keys

```
// composant pour gérer une liste de courses
// il utilise le composant précédent pour l'affichage
// le tableau d'articles state.articles est passé en paramètre

class ListeCourses extends React.Component {
   state = { articles : [ {id:3, article:"Table", qte:2},
     {id:4, article: "Chaise", qte:4}, … ] }

   render() {
      return (
        <div>
          <h3>Liste de courses</h3>
          <ItemList liste={ this.state.articles }/>
        </div>
      );
   }
}
```

List & keys

- React impose que chaque élément de la liste générée comporte une clé unique (attribut key)

Formulaires

Formulaires

- La création d'un formulaire en JSX est très proche d'un formulaire HTML (même syntaxe)
- Cependant le fonctionnement est très différent de HTML
  ➔ c'est au développeur de gérer tous les changements

Exemple : formulaire de saisie d'un texte

- Composant à état
- Le state contient un propriété pour stocker la valeur saisie
- Dès que l'utilisateur modifie sa saisie il faut changer la valeur du state *[méthode handlerChange]*
- Quand l'utilisateur valide (submit) la valeur saisie est disponible *[méthode handlerSubmit]*


```
class App extends Component {
state = { texte : "" }
```

Composant à état
Le state contient la prop texte

```
render() {
 return (
  <form onSubmit={this.handleSubmit}>
   <label>
    Chose :
    <input type="text"
       value={this.state.texte}
       onChange={this.handleChange} />
   </label>
   <input type="submit" value="Submit" />
  </form>
 );
}
```

Le formulaire en HTML

Gestion de l'evt submit

La valeur affichée est celle du state

Gestion de l'evt change

Ce handler est appelé à chaque fois que la valeur saisie est modifiée
→ On met à jour le state avec la valeur saisie
→ Comme le state est modifié React actualise le DOM virtuel
→ La bouvelle valeur saisie est affichée

```
handleChange = (event) => {
   this.setState({ texte : event.target.value})
}
```

Ce handler est appelé quand l'utilisateur valide sa saisie (clic sur le bouton submit)
→ la valeur saisie est dans le state

```
handleSubmit = (event) => {
   event.preventDefault()
   console.log(this.state.texte)
}
```


Les hooks

Hook state

- Avec l'introduction du « hook state », il est maintenant possible de gérer l'état avec un composant fonctionnel
  [Il n'est plus nécessaire de créer un composant de type classe]
- La déclaration du state est remplacée par `state = { compteur : 0 }`

```
const [ compteur, setCompteur ] = useState(0)
```

➔déclaration de la variable compteur

➔la fonction setState est remplacée par setCompteur
[le nom n'est plus imposée]

➔la fonction useState permet d'initialiser la variable compteur

Hook state

Exemple : compteur avec composant de type classe

```
class Compteur extends React.Component {
    state = { compteur : 0 }
    plusun() { this.setState( { compteur: this.state.compteur+1 } ) }
    render() {
        return (
            <button onClick={()=>this.plusun()} >{`compteur : ${this.state.compteur}`}</button>
        )
    }
}
```

Exemple : compteur avec composant fonctionnel et hook state

```
function CompteurHook() {
    const [ compteur, setCompteur ] = useState(0)
    const plusun = () => setCompteur(compteur+1)
    return (
        <div>
            <button onClick={plusun} >{`compteur hook : ${compteur}` }</button>
        </div>
    )
}
```

Formulaire avec le hook d'état

- Code du formulaire précédent avec le hook d'état
- Utilisation d'un composant fonctionnel

Texte Submit

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

function FormTexte (props) {
    const [texte, setTexte] = useState("")
```

Utilisation du hook useState

Déclaration d'une variable d'état texte qui sera gérée par la fonction setTexte ; la variable est initialisée à ""

Formulaire avec le hook d'état

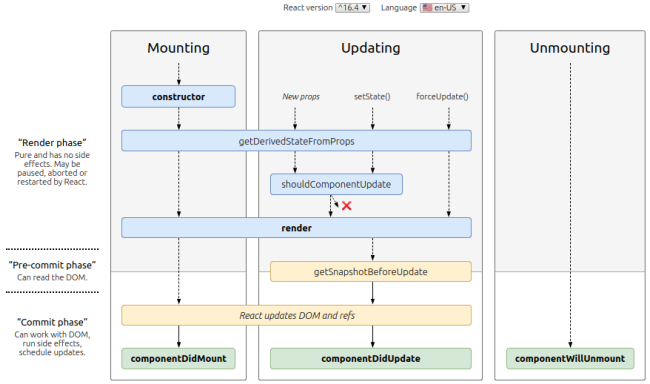
A chaque chgt dans l'input on utilise la fonction setTexte pour stocker dans la variable d'état texte la valeur saisie

```
const handlerChange = (event) => setTexte(event.target.value)
const handleSubmit = (event) => {
    event.preventDefault()
    console.log(texte)
    }
return (
    <form onSubmit={handleSubmit}>
        <label> Ville :
            <input
                type="text"
                value={texte}
                onChange={handlerChange} />
        </label>
        <input type="submit" value="Submit" />
    </form>
)
```

Lors de la validation on pourra récupérer la valeur saisie dans la variable texte

Le hook useEffect

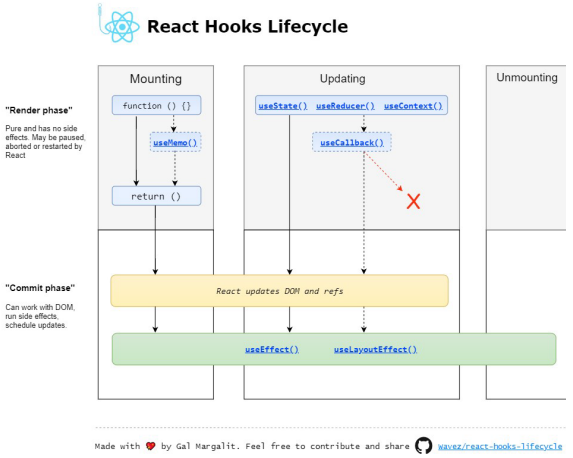
- Objectif : remplacer les 3 méthodes component… du composant de type classe permettant de gérer le cycle de vie



Le hook useEffect

Une combinaison de componentDidMount, componentDidUpdate, et componentWillUnmount

useEffect permet de définir la fonction qui sera exécutée après chaque affichage, donc après chaque chgt des props ou du state.



Le hook useEffect

La fonction useEffect a deux paramètres, la fonction et un tableau de dépendances :

- useEffect(() => {…}) → la fonction s'exécute à chaque fois
- useEffect(() => {…}, [ ]) → si le tableau de dépendances est présent mais vide, la fonction ne s'exécute qu'une fois
- useEffect(() => {…}, [compteur]) → la fonction ne s'exécute que lorsque la variable du state compteur est modifiée
- useEffect(() => {…}, [props.val]) → la fonction ne s'exécute que lorsque la props val est modifiée

Remarque : ne pas modifier les props ou le state dans la fonction sinon cycle infini… sauf si le 2ème param est [ ] (équivalent à componentDidMount)

Deployer une application React

Hébergement

- Il faut créer une version de production à l'aide de la commande : npm run build
- Le dossier build contient cette version optimisée pour de meilleures performances.
  1 fichier index.html + 1 fichier css + 1 fichier js
- Le service netlify.com permet d'héberger facilement un site
- Il est aussi possible de l'héberger sur un serveur Apache classique mais il y a souvent des pbs d'urls relatives ou virtuelles ➔ trouver le « bon » fichier .htaccess

Utilisation d'un router

Utilisation d'un routeur

- Dès que l'application devient un peu complexe avec de nombreuses fonctionnalités, il est nécessaire de mettre en place un routeur pour gérer la navigation dans l'application
- Par exemple, pour gérer un menu, il faut :
  - Associer à chaque item une « route »
  - Mettre en place le « router » qui va associer à chaque route le composant à afficher

React-router-dom

- react-router-dom est un exemple de router que l'on peut utiliser dans une application React

https://reactrouter.com/web/guides/quick-start

Un exemple

```
<Router>
<ul>                    LE MENU
   <li><Link to="/">Home</Link></li>
   <li><Link to="/about">About</Link></li>
   <li><Link to="/users">Users</Link></li>
</ul>




</Router>
```

```
A chaque route -> un composant
<Switch>
   <Route path="/about">
         <About />
   </Route>
   <Route path="/users">
         <Users />
   </Route>
   <Route path="/">
         <Home />
   </Route>
</Switch>
```


React-router-dom

- Il est fréquent d'avoir besoin de routes avec des paramètres
- React-router-dom permet de le gérer assez simplement

Par exemple, une route permettant de gérer un user particulier, connaissant son id

```
<Route path="/user/:idUser"><User></Route>
```

➔Le composant <User> peut récupérer la valeur du param en utilisant le hook useParams() :

```
let { idUser } = useParams();
```