



# Formation ReactJS, programmation avancée

Romain Bohdanowicz

Twitter : @bioub - Github : <https://github.com/bioub>

<http://formation.tech/>



- Romain Bohdanowicz  
Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle
- Expérience  
Formateur/Développeur Freelance depuis 2006  
Plus de 10 000 heures de formation animées
- Langages  
Expert : HTML / CSS / JavaScript / PHP / Java  
Notions : C / C++ / Objective-C / C# / Python / Bash / Batch
- Certifications  
PHP 5 / PHP 5.3 / PHP 5.5 / Zend Framework 1
- Divers  
Premier site web à 12 ans (HTML/JS/PHP), Loisirs : Triathlon
- Et vous ?  
Langages ? Expérience ? Utilité de cette formation ?



React



- Les props peuvent être vues comme les paramètres d'entrées d'un composant

```
export function Hello(props) {  
  return (  
    <div className="Hello">  
      Hello {props.name} !  
    </div>  
  );  
}  
  
function App() {  
  return (  
    <div className="App">  
      <Hello name="Romain" />  
    </div>  
  );  
}
```



- Il est conseillé d'utiliser la déstructuration d'objet ES2015

```
export function Hello({name = ''}) {  
  return (  
    <div className="Hello">  
      Hello {name} !  
    </div>  
  );  
}
```

```
function App() {  
  return (  
    <div className="App">  
      <Hello name="Romain" />  
    </div>  
  );  
}
```



- Le state est une valeur interne au composant, son rafraîchissement via `setState` ou `forceUpdate` va mettre à jour toute l'application

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

export default class Clock extends Component {
  constructor() {
    super();
    this.state = {
      now: new Date(),
    };
    setInterval(() => {
      this.setState({
        now: new Date(),
      });
    }, 1000);
  }
  render() {
    return <div className="Clock">{this.state.now.toLocaleTimeString()}</div>;
  }
}
```

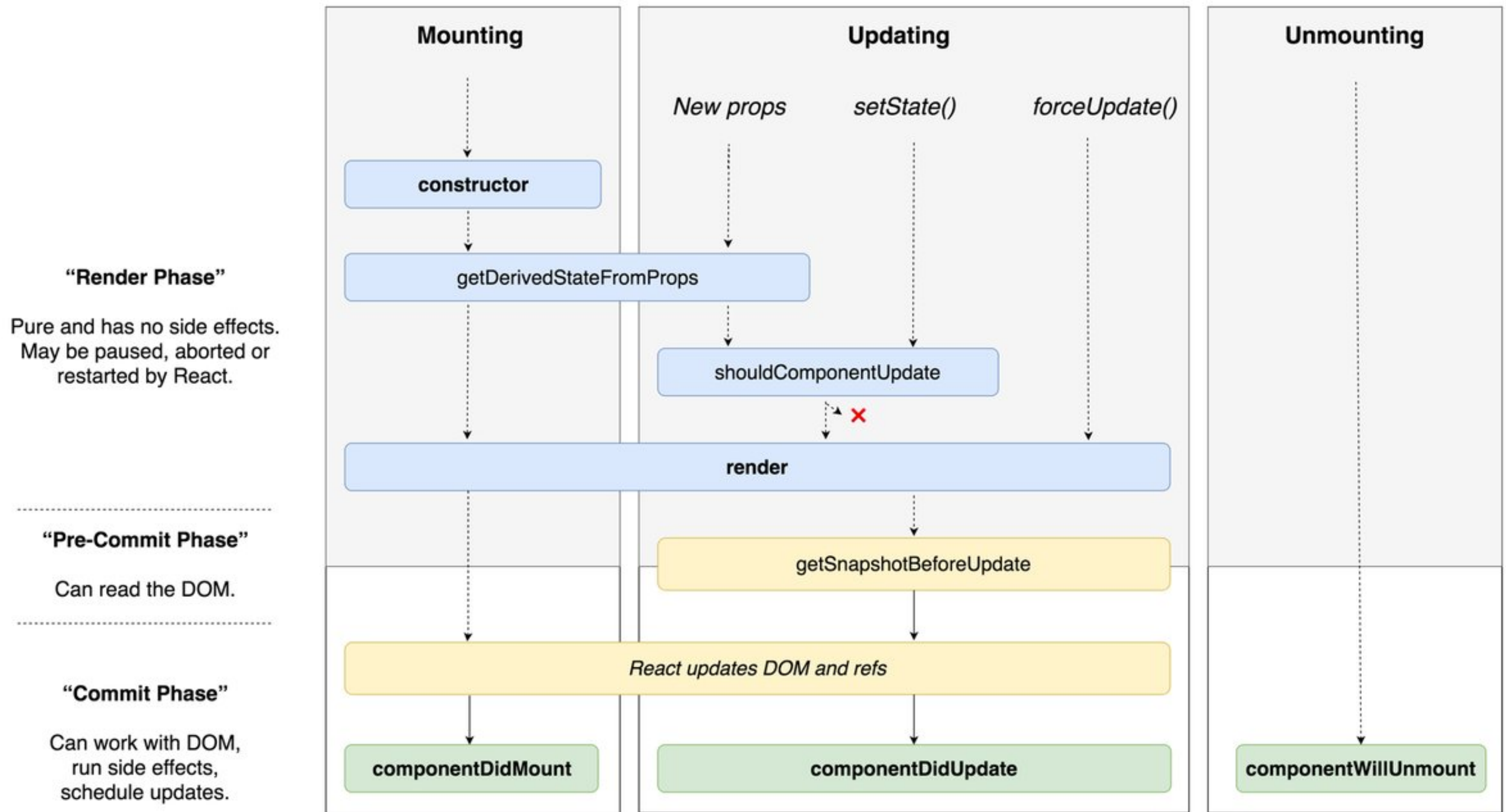


- Les méthodes du Lifecycle sont des méthodes appelées automatiquement à certains moments clés, principalement au moment où le composant apparaît pour la première fois, est mis à jour ou disparaît.

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

export default class Clock extends Component {
  constructor() {
    super();
    this.state = {
      now: new Date(),
    };
  }
  componentDidMount() {
    this._interval = setInterval(() => {
      this.setState({
        now: new Date(),
      });
    }, 1000);
  }
  componentWillUnmount() {
    clearInterval(this._interval);
  }
  render() {
    return <div className="Clock">{this.state.now.toLocaleTimeString()}</div>;
  }
}
```

# React - Lifecycle







- Ecouter les événements avec `on*` revient à utiliser `document.addEventListener` (`rootEl.addEventListener` depuis React 17)
- Certains événements sont réinterprétés (`onChange`, `onFocus`, `onBlur`...)

```
class Counter extends Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState({
      count: this.state.count + 1,
    });
  }
  render() {
    return (
      <button className="Counter" onClick={this.handleClick}>
        {this.state.count}
      </button>
    );
  }
}
```



- On utilise `.bind(this)` sinon le handler n'aurait pas accès à `this`

```
class Counter extends Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState({
      count: this.state.count + 1,
    });
  }
  render() {
    return (
      <button className="Counter" onClick={this.handleClick}>
        {this.state.count}
      </button>
    );
  }
}
```



- On peut également utiliser bind dans la méthode render mais une nouvelle fonction sera créée à chaque render

```
class Counter extends Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
  }
  handleClick() {
    this.setState({
      count: this.state.count + 1,
    });
  }
  render() {
    return (
      <button className="Counter" onClick={this.handleClick.bind(this)}>
        {this.state.count}
      </button>
    );
  }
}
```



- Les fonctions fléchées sont également créé à chaque render

```
class Counter extends Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
  }
  handleClick() {
    this.setState({
      count: this.state.count + 1,
    });
  }
  render() {
    return (
      <button className="Counter" onClick={() => this.handleClick()}>
        {this.state.count}
      </button>
    );
  }
}
```



- Le plus court et performant, les class properties de ES2022

```
class Counter extends Component {
  state = {
    count: 0,
  };
  handleClick = () => {
    this.setState({
      count: this.state.count + 1,
    });
  }
  render() {
    return (
      <button className="Counter" onClick={this.handleClick}>
        {this.state.count}
      </button>
    );
  }
}
```

# React - Composant non-contrôlé



- Un composant non-contrôlé est un composant dont le parent ne peut contrôler son contenu

```
class Counter extends Component {
  state = {
    count: 0,
  };
  handleClick = () => {
    this.setState({
      count: this.state.count + 1,
    });
  };
  render() {
    return (
      <button className="Counter" onClick={this.handleClick}>
        {this.state.count}
      </button>
    );
  }
}

function App() {
  return (
    <div className="App">
      <Counter />
    </div>
  );
}
```

# React - Composant contrôlé



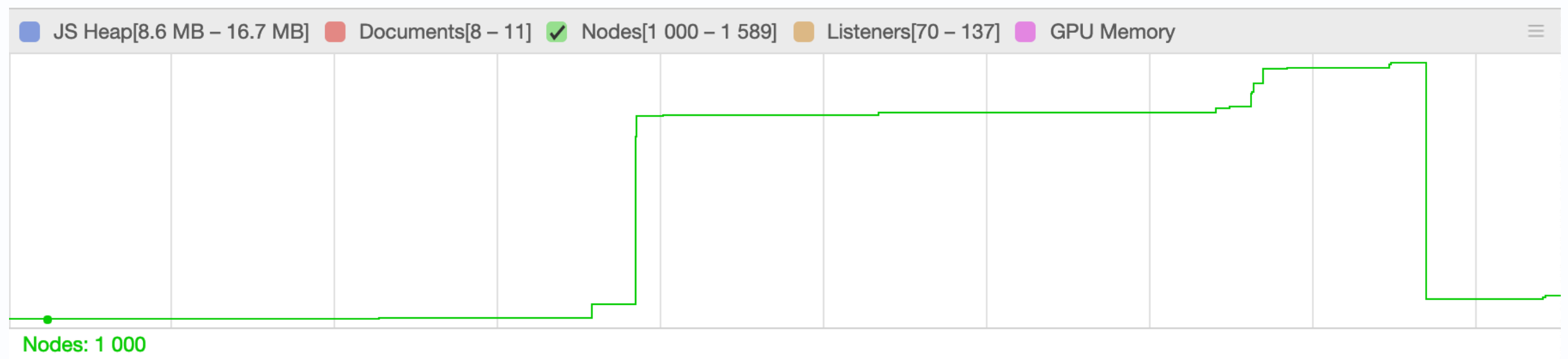
- Un composant contrôlé n'utilise pas le state mais est contrôlé par son parent

```
function Counter({ count, onIncrement }) {
  return (
    <button className="Counter" onClick={() => onIncrement()}>
      {count}
    </button>
  );
}

class App extends Component {
  state = {
    count: 0,
  };
  render() {
    const { count } = this.state;
    return (
      <div className="App">
        <Counter count={count} onIncrement={() => this.setState({count: count +
1})} />
      </div>
    )
  }
}
```



- Le DOM ou Document Objet Model est l'API du navigateur créé par Netscape en 1995 qui permet de manipuler le contenu de la page web
- Cet API est ancien même s'il reçoit des évolutions régulièrement
- Il est également très lourd, par exemple la page d'accueil de [formation.tech](https://formation.tech) va créer jusqu'à 1589 objet associés au DOM en mémoire



- Lorsqu'un composant React doit se rafraîchir (en appelant sa méthode render), il serait très coûteux de recréer tout les éléments du DOM qu'il contient. Pour éviter cela React met en place un "Virtual DOM"





- Voici un exemple de mini-framework sans Virtual DOM

```
class Component {
  _refresh() {
    this.host.innerHTML = '';
    this.render().forEach(elt => this.host.appendChild(elt));
  }
  setState(newState) {
    Object.assign(this.state, newState);
    this._refresh();
  }
}

function domRender(component, host) {
  component.host = host;
  component._refresh();
}
```

- Comme dans React, appeler la méthode `setState` ou `domRender` provoquera le rafraîchissement du composant en appelant sa méthode `render`.



- Comme React, le composant possède une méthode render qui construit le DOM

```
class ButtonCount extends Component {
  state = { count: 0 };
  increment = () => {
    this.setState({count: this.state.count + 1});
  };
  render() {
    const p = document.createElement('p');
    p.innerText = 'Démo : ';

    const button = document.createElement('button');
    button.innerText = this.state.count;
    button.onclick = this.increment;

    return [p, button];
  }
}

ReactDOM.render(new ButtonCount(), document.querySelector('hello-component'));
```

- On remarque que l'API DOM est lourd, si on pouvait chainer comme jQuery il n'y aurait que 2 lignes dans render
- Puis on peut demander le rendu dans une balise existante ici hello-component



- Lorsqu'on observe le résultat avec les DevTools de Chrome, on voit que l'ensemble du DOM associé au composant est rafraîchi

Démo :

22

```
Elements Console Source
<!doctype html>
<html lang="en">
  ><head>...</head>
  ▼<body>
    ... ▼<hello-component> == $0
      <p>Démo : </p>
      <button>22</button>
    </hello-component>
    <script src="dom.js"></script>
  </body>
</html>
```



- Avec `React.createElement` on va construire avec un API plus moderne un arbre léger en mémoire appelé Virtual DOM

```
class ButtonCount extends React.Component {
  state = { count: 0 };
  increment = () => {
    this.setState({count: this.state.count + 1});
  };
  render() {
    return [
      React.createElement('p', null, 'Démo : '),
      React.createElement('button', {onClick: this.increment}, this.state.count),
    ];
  }
}

ReactDOM.render(
  React.createElement(ButtonCount),
  document.querySelector('hello-component'),
);
```



- Avec React et son Virtual DOM on remarque que le navigateur ne rafraîchit pas plus d'élément que nécessaire :

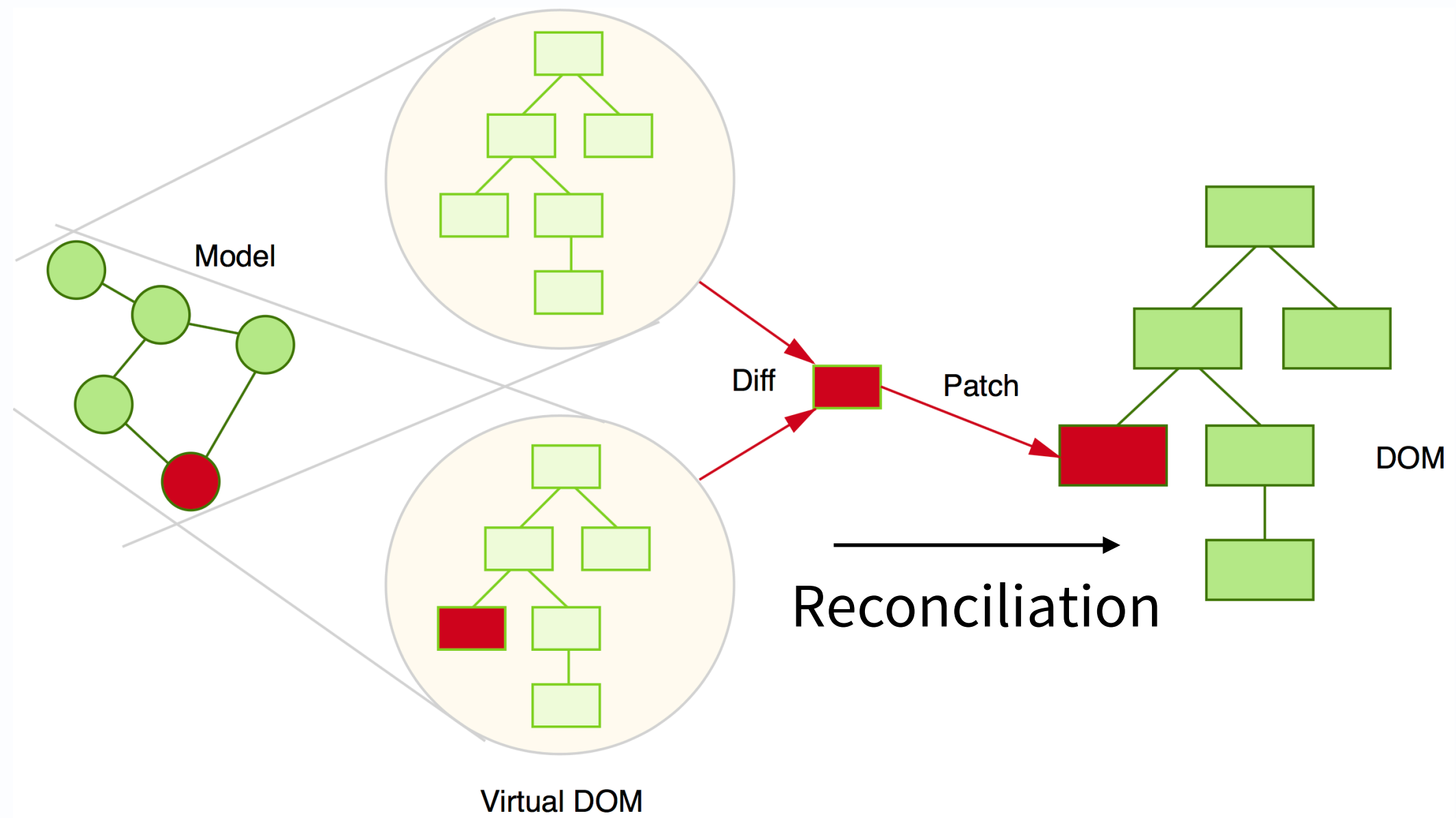
Démo :



```
Elements Console Source
<!doctype html>
<html lang="en">
  ><head>...</head>
  ▼<body>
    ... ▼<hello-component> == $0
      <p>Démo : </p>
      <button>22</button>
    </hello-component>
    <script src="../node_modules/react"
    <script src="../node_modules/react
    <script src="react.js"></script>
  </body>
```



## ▸ Réconciliation





- L'exemple précédent est encore trop verbeux. Afin de le simplifier, React a créé une syntaxe appelée JSX pour construire le Virtual DOM d'un composant
- Le navigateur ne reconnaissant pas cette syntaxe on va utiliser un compilateur (en général Babel et son plugin `@babel/plugin-transform-react-jsx`) pour transformer le JSX en `React.createElement`

```
class ButtonCount extends React.Component {
  state = { count: 0 };
  increment = () => {
    this.setState({count: this.state.count + 1});
  };
  render() {
    return [
      <p>Démo :</p>,
      <button onClick={this.increment}>{this.state.count}</button>,
    ];
  }
}

ReactDOM.render(
  <ButtonCount />,
  document.querySelector('hello-component'),
);
```



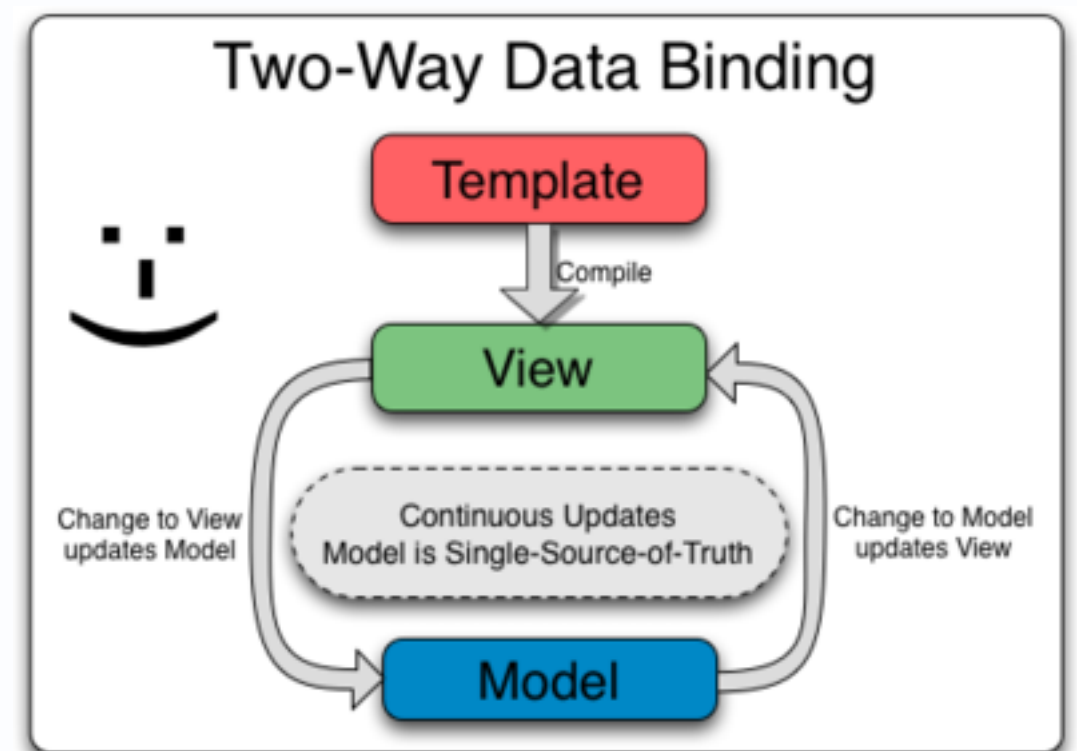
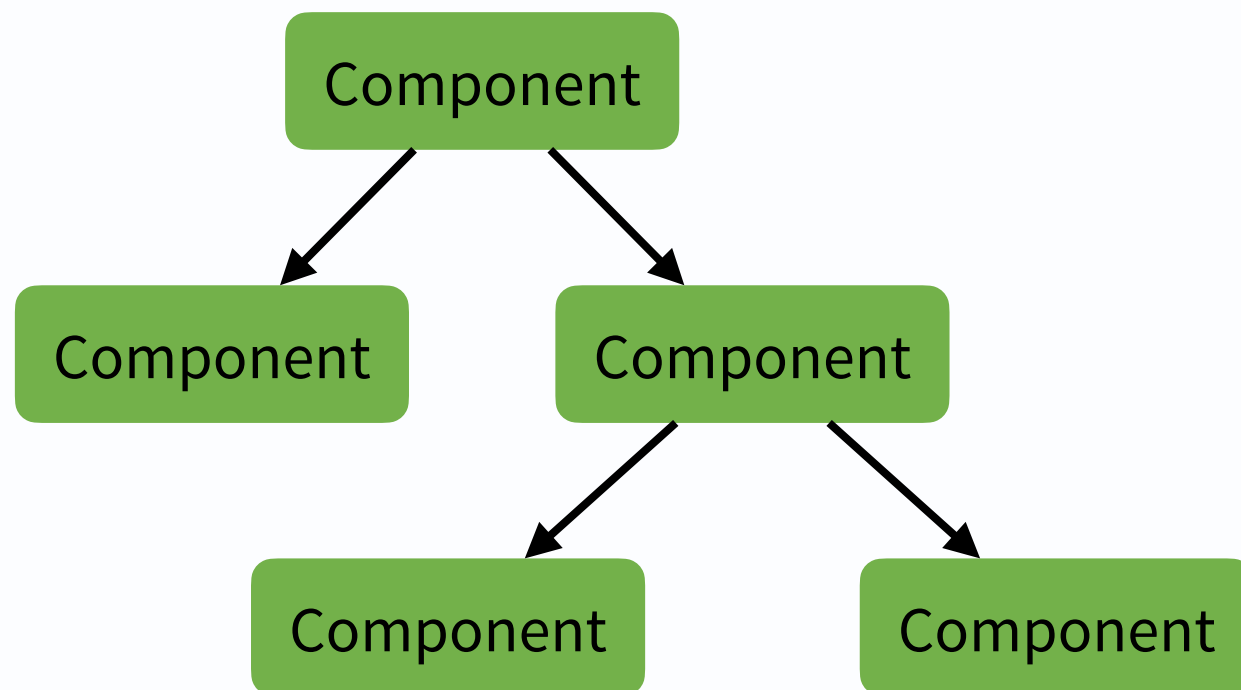
- Depuis React 17 il n'est plus nécessaire d'importer React pour pouvoir utiliser du JSX



# React - One Way Data Flow



- Par opposition aux frameworks de génération précédente comme AngularJS, Knockout ou Ember.js, les données circulent toujours dans un sens dans React : d'un composant parent vers un composant enfant. On parle de One-Way Data Flow, One-Way Data Binding ou Unidirectional Data Flow



# React - Outils de debug



- React Developer Tools
  - Extension officielle de Facebook
  - Fonctionne avec Chrome et Firefox
  - Permet de surveiller les objets props, state, context
- Téléchargement
  - Chrome  
<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>
  - Firefox  
<https://addons.mozilla.org/fr/firefox/addon/react-devtools/>

# React - Outils de debug



React DevTools interface showing the React component tree and props.

**React Component Tree:**

```
<Provider>
  <Context.Provider>
    <HashRouter hashType="noslash">
      <Router>
        <App>
          <div className="App">
            <h3>Faire un essai gratuit</h3>
            <Route path="/" exact={true}></Route>
            <Route path="/step2"></Route>
            <Route path="/step3">
              <Step3...</Step3> == $r
            </Route>
            <Route path="/step4"></Route>
            <Route path="/step5"></Route>
            <Route path="/step6"></Route>
          </div>
        </App>
      </Router>
    </HashRouter>
  </Context.Provider>
</Provider>
```

**Props for the selected component (Step3):**

- history: {...}
  - action: "POP"
  - block: block()
  - createHref: createHref()
  - go: go()
  - goBack: goBack()
  - goForward: goForward()
  - length: 4
  - listen: listen()
  - location: {...}
  - push: push()
  - replace: replace()
- location: {...}
  - hash: ""
  - pathname: "/step3"
  - search: ""
- match: {...}
  - ☒ isExact: true
  - params: {...}
  - path: "/step3"
  - url: "/step3"

**Breadcrumbs:** Provider > Context.Provider > HashRouter > Router > App > div > Route > Step3

# React - Outils de debug



```

    <Route path="/step2"></Route>
  <Route path="/step3">
    <Step3>...</Step3> == $r
  </Route>
  <Route path="/step4"></Route>
  <Route path="/step5"></Route>
  <Route path="/step6"></Route>
</div>
</App>
</Router>
</HashRouter>
</Context.Provider>
</Provider>

```

Provider Context.Provider HashRouter Router App div Route Step3

⋮ Console What's New

▶ top Filter Default levels ▼

```

> $r
< ▼Route {props: {...}, context: {...}, refs: {...}, updater: {...}, state: {...}, ...} ⓘ
  ▶ context: {router: {...}}
  ▶ props: {path: "/step3", component: f}
  ▶ refs: {}
  ▶ state: {match: {...}}

```

# React - Outils de debug



The screenshot displays the Redux DevTools interface, which is used for debugging state changes in a Redux application. The interface is divided into several panels:

- Top Panel:** Shows the browser address bar with the URL `http://localhost:3000/#extension-demo`.
- Left Panel (Inspector):** Displays a list of actions performed on the state. The actions include `@@@INIT`, `INCREMENT_COUNTER`, `INCREMENT_COUNTER`, `DECREMENT_COUNTER`, and `INCREMENT_COUNTER`. Each action is timestamped.
- Right Panel (Diff):** Shows the state before and after each action. For example, the state changes from `counter: 0` to `counter: 1` after the first `INCREMENT_COUNTER` action.
- Bottom Panel (Log monitor):** Displays the state of the application at each step. It shows the state object `{ counter: 0 }` and the state object `{ counter: 1 }`.
- Bottom Right Panel:** Shows the state of the application at each step, including the state object `{ counter: 1 }` and the state object `{ counter: 2 }`.

The interface also includes a **Chart** panel on the left, an **Autoselect instances** dropdown, and a **Dispatch** button. The **Diff** panel shows the state before and after each action, with the state object `{ counter: 1 }` and the state object `{ counter: 2 }`.



- Récupérer le projet suivant et installer les dépendances  
<https://gitlab.com/react-avance>
- Créer un composant Home dans src/components sous forme de classe, y ajouter le titre Home dans le JSX
- Utiliser le Router pour afficher ce composant associé à l'URL /, utiliser l'option exact
- Ajouter un lien dans le composant TopBar



- Créer un composant Select dans src/components sous forme de classe
  - 3 props : items (string[]), selected (string), onSelect (function)
  - 1 state : open (boolean default false)
- La méthode render doit retourner le JSX suivant

```
<div className="Select">  
  <div className="selected">Rouge</div>  
  <div className="items">  
    <div className="item">Rouge</div>  
    <div className="item">Vert</div>  
    <div className="item">Bleu</div>  
  </div>  
</div>
```



- Le CSS est le suivant :

```
.Select {  
  position: relative;  
  width: 200px;  
  cursor: pointer;  
}  
  
.Select .selected {  
  border: 1px solid black;  
  padding: 5px;  
}  
  
.Select .items {  
  position: absolute;  
  border: 1px solid black;  
  width: 100%;  
  box-sizing: border-box;  
  background-color: white;  
}  
  
.Select .item {  
  padding: 5px;  
}
```





- Remplacer le code de Home par :

```
class Home extends Component {
  state = {
    prenom: ['Jean', 'Paul', 'Eric'],
    selectedPrenom: 'Jean',
  };
  render() {
    const { prenom, selectedPrenom } = this.state;
    return (
      <div>
        <p>Vous avez sélectionné : {selectedPrenom}</p>
        <Select
          items={prenom}
          selected={selectedPrenom}
          onSelect={item => this.setState({ selectedPrenom: item })}
        />
      </div>
    );
  }
}
```



- Afficher les props items et selected dans le JSX
- Conditionner l'affichage des items au state open
- Au clic de selected, passer open à !open
- Au clic de item, remonter la valeur en appelant la props onSelect, passer open à !open
- Ecouter le click de document dans componentDidMount, y passer open à false
- Retirer le click de document dans componentWillUnmount avec removeEventListener
- Installer node-sass et remplacer le fichier .css de Select par un .module.scss



# React Avancé

# React Avancé - Fragments



- React 16 a introduit la possibilité pour un composant de ne plus avoir un élément racine sous forme de tableau

```
function ListItem({ term, definition }) {  
  return [  
    <dt>{term}</dt>,  
    <dd>{definition}</dd>,  
  ];  
}
```

- Ceci pour permettre de créer des composants plus fin dans un contexte où une balise intermédiaire serait problématique

```
function DefinitionList() {  
  const abbrs = { JS: "JavaScript", CSS: "Cascading Style Sheets" };  
  return (  
    <dl>  
      {Object.entries(abbrs).map(([term, definition]) => (  
        <ListItem key={term} term={term} definition={definition} />  
      ))}  
    </dl>  
  );  
}
```



- Les tableaux ont des contraintes :
  - Il faut séparer le contenu par des virgules
  - Il faut ajouter un paramètre key pour la réconciliation
  - Le texte doit être entre guillemets
  - Les commentaires sont différents du JSX

# React Avancé - Fragments



- Avec les Fragments plus besoin de tableaux

```
function ListItem({ term, definition }) {  
  return (  
    <React.Fragment>  
      <dt>{term}</dt>  
      <dd>{definition}</dd>,  
    </React.Fragment>  
  );  
}
```

- En syntaxe courte :

```
function ListItem({ term, definition }) {  
  return (  
    <>  
      <dt>{term}</dt>  
      <dd>{definition}</dd>,  
    </>  
  );  
}
```



- Les refs permettent de récupérer une référence vers un objet
- Elles peuvent être utilisées :
  - Pour récupérer un élément du DOM
  - En optimisation car elles ne sont pas recréées d'un render à un autre

```
class Form extends Component {  
  constructor(props) {  
    super(props);  
    this.inputRef = React.createRef();  
  }  
  componentDidMount() {  
    this.inputRef.current.focus();  
  }  
  render() {  
    return <input ref={this.inputRef} />;  
  }  
}
```

# React Avancé - Higher Order Components



- Permettent d'ajouter des fonctionnalités à un composants de manière générique
- HOC = une fonction qui reçoit un composant en entrée et qui retourne un nouveau composant composé du premier
- Exemple : connect de react-redux, withRouter de react-router-dom

```
▼ <withRouter(Step) number={3} title="Quel est le niveau de l'élève ?">
  ▼ <Route>
    ▼ <Step number={3} title="Quel est le niveau de l'élève ?"> == $r
      ▼ <div className="Step Step3">
        ► <Link className="previous" to="/step2" replace={false}>...</Link>
          <h4 className="title">Quel est le niveau de l'élève ?</h4>
        ► <Choices>...</Choices>
      </div>
    </Step>
  </Route>
</withRouter(Step)>
```



# React Avancé - Higher Order Components



- Bonnes pratiques
  - Le nom du composant résultant :  
`nomDuHOC (NomDuComposant)`
  - Les props passées au composant résultant doivent être transmises au composant imbriqué (à l'exception de celles ne servant qu'au HOC) :  
`OuterCmp.displayName = `hideable(${InnerCmp.displayName})` ;`

# React Avancé - Higher Order Components



## ▸ Exemple

```
function hideable(InnerComponent) {  
  class OuterComponent extends Component {  
    state = {  
      show: this.props.show,  
    };  
    handleClick = () => {  
      this.setState({  
        show: !this.state.show,  
      });  
    };  
    render() {  
      const {show, ...innerProps} = this.props;  
  
      return (  
        <div className="HideableClock">  
          {this.state.show && <InnerComponent {...innerProps} />}  
          <button onClick={this.handleClick}>  
            {this.state.show ? 'Off' : 'On'}  
          </button>  
        </div>  
      )  
    }  
  }  
  return OuterComponent;  
}
```



- Permettent le rendu dans des éléments DOM distants
- Exemple :  
une Modal avec Bootstrap

```
<div class="modal fade" id="exampleModalLong">
  <div class="modal-dialog" role="document">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="exampleModalLongTitle">Modal title</h5>
        <button type="button" class="close" data-dismiss="modal">
          <span>&times;</span>
        </button>
      </div>
      <div class="modal-body">
        // contenu React à afficher ici...
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-
dismiss="modal">Close</button>
        <button type="button" class="btn btn-primary">Save changes</button>
      </div>
    </div>
  </div>
</div>
```



- Le composant Modal pourra faire son rendu dans l'élément

```
class Modal extends Component {
  el = document.createElement('div');

  componentDidMount() {
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(
      this.props.children,
      document.querySelector('#exampleModalLong'),
    );
  }
}
```

```
<Modal>
  Contenu
</Modal>
```



- Context est un objet dont la modification provoque le rendu comme props et state
- Context est utile pour des cas où une valeur doit être fournie globalement pour une hiérarchie de composant (via props il faudrait passer cette valeur à chaque composant ou sous-composant)
- Cas d'utilisation
  - Thèmes
  - Locale
  - Utilisateur connecté
  - Services interchangeables

# React Avancé - Context



- Pour créer un Context on utilise la méthode `createContext` de React  
Bonne pratique exporter le context (ici 'dark' est la valeur par défaut)

```
import React, { createContext } from 'react';  
  
export const ThemeContext = createContext('dark');
```

- Pour fournir une nouvelle valeur on utilise le composant `Provider` du context

```
<ThemeContext.Provider value="light">  
  <Navbar />  
</ThemeContext.Provider>
```

- Enfin `Navbar` ou n'importe quel autre composant présent dans la hiérarchie sous `Provider` pourra souscrire aux modifications du context via la propriété `contextType`

```
export class Navbar extends Component {  
  render() {  
    return <div className={"UserList " + this.context}>Menu</div>;  
  }  
}  
  
Navbar.contextType = ThemeContext;
```



- Inconvénient de `contextType` :
  - obligation d'utiliser une classe
  - ne permet d'obtenir qu'un seul contexte
- On peut également utiliser `Context.Consumer` :

```
function Profile() {  
  return (  
    <UserContext.Consumer>  
      ({ { name } }) => {  
        return (  
          <div className="Profile">  
            Name : {name}  
          </div>  
        );  
      }  
    </UserContext.Consumer>  
  );  
}
```

# React Avancé - Render Props



- Render Props est une technique consistant à passer une fonction dans les propriétés du composant qui sera en charge du rendu
- Cette fonction aura elle même accès aux propriétés du composants
- Exemple : le composant Field de redux-form

```
const renderField = ({
  input,
  label,
  placeholder,
  type,
  meta: { touched, error },
}) => (
  <div>
    <label>{label}</label>
    <input
      {...input}
      placeholder={placeholder}
      type={type}
      className={classNames({ error: error && touched })}
    />
    {touched && error && (
      <span className={classNames({ error: error && touched })}>{error}</span>
    )}
  </div>
);
```



# React Avancé - Render Props



```
<Field  
  name="prenom"  
  type="text"  
  component={renderField}  
  placeholder="Ex: Guillaume"  
  label="Prénom de l'élève"  
>
```



- Reprendre le projet précédent
- Utiliser un Fragment dans le composant TodoList
- Dans le composant Select, créer une ref sur l'élément racine, utiliser cette ref pour savoir si le clic à eu lieu dans l'élément ou en dehors (`votreRef.current.contains(event.target)`), ne refermer la balise Select que si le click à eu lieu en dehors
- Ajouter une render prop nommée `component` à Select, utiliser cette fonction quand elle est définie pour faire le rendu de l'item
- Créer un contexte `PrenomContext`, lorsque l'ont clique sur un item dans le menu, afficher la valeur dans la TopBar



# Hooks



- Historiquement on déclare un composant React sous forme de fonction ou sous forme de classe
- Avantage des fonctions : simple, court
- Avantage des classes : state, context, refs, lifecycle methods...
- Problèmes :
  - inconsistence dans l'application
  - écriture d'un composant sous forme de fonction pour se rendre compte en cours de dev qu'il aurait fallu une classe
- Les Hooks permettent d'écrire des fonctions qui auront accès aux mêmes éléments qu'une classe
- Si vous connaissiez les composants sous forme de fonction sous le nom : "Stateless Component", cela n'a plus de sens on devrait plutôt dire "Function Component"

# Hooks - Exemple de classe (ES6)



- Voici un exemple d'utilisation d'un composant sous forme de classe (370 octets)

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

export class Counter extends Component {
  constructor() {
    super();
    this.state = { count: 0 };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState({ count: this.state.count + 1 });
  }
  render() {
    return <button onClick={this.handleClick}>{this.state.count}</button>;
  }
}
```

# Hooks - Exemple de classe (ESNext)



- On pourrait réduire le poids et donc la lisibilité de ce fichier en utilisant les propriétés de classes (pas encore normé dans JavaScript) (272 octets)

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

export class Counter extends Component {
  state = { count: 0 };
  handleClick = () => this.setState({ count: this.state.count + 1 });
  render() {
    return <button onClick={this.handleClick}>{this.state.count}</button>;
  }
}
```

# Hooks - Exemple de hook



- Avec les hooks, on peut réduire encore le poids du fichier (219 octets) mais surtout être consistant dans le code puisque tous les composants seront alors des fonctions

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

export function Counter() {
  const [count, setCount] = useState(0);
  const handleClick = () => setCount(count + 1);
  return <button onClick={handleClick}>{count}</button>;
}
```

- useState est le hook qui permet de manipuler le state
- useEffect remplace les méthodes du cycle de vie

# Hooks - Recompose



- Historiquement on aurait pu arriver au même résultat avec la bibliothèque `recompose`

```
const Counter = withStateHandlers(  
  ({ initialCounter = 0 }) => ({  
    counter: initialCounter,  
  }),  
  {  
    incrementOn: ({ counter }) => (value) => ({  
      counter: counter + value,  
    }),  
    decrementOn: ({ counter }) => (value) => ({  
      counter: counter - value,  
    }),  
    resetCounter: (_, { initialCounter = 0 }) => () => ({  
      counter: initialCounter,  
    }),  
  }  
)(  
  ({ counter, incrementOn, decrementOn, resetCounter }) =>  
    <div>  
      <Button onClick={() => incrementOn(2)}>Inc</Button>  
      <Button onClick={() => decrementOn(3)}>Dec</Button>  
      <Button onClick={resetCounter}>Reset</Button>  
    </div>  
)
```





- Transformez Select en fonction avec des Hooks
  - Utiliser useState pour open
  - Utiliser useEffect pour componentDidMount et componentWillUnmount
  - Utiliser useRef pour créer la ref
- Transformez Clock en fonction avec des Hooks
  - Utiliser useState pour now
  - Utiliser useEffect pour componentDidMount et componentWillUnmount
  - Utiliser useRef pour stocker \_interval
  - Déporter ensuite tout ce code dans une fonction useClock(1000) où 1000 est le délai de rafraichissement



# Optimisation des performances

# Optimisation - Introduction



- Plus l'application React va grandir, plus le nombre de composants va être élevé et donc les appels à render longs à exécuter
- Les composants les plus problématiques : ceux recevant en props une liste d'élément dont chacun sera rendu sous forme d'un composant. Un changement dans une liste de 1000 éléments === 1000 appels à render + 1000 mises à jour du DOM potentielles
- Les performances des différents frameworks JavaScript sur des listes : <https://github.com/krausest/js-framework-benchmark>

# Optimisation - Keys



- Au moment de la réconciliation, React va comparer la version précédente du Virtual DOM d'un composant avec la version actuelle (juste après l'appel à render)
- Avec un tableau passé en props ou state, si une valeur est insérée au début, l'ensemble des éléments du DOM devront être mis à jour.
- Pour éviter cela, il faut passer au Virtual DOM un paramètre key dans les props lui permettant d'établir un lien entre l'élément du Virtual DOM et l'élément du DOM
- Choisir une valeur unique, et non modifiée en cas de mise à jour de l'élément (id de la database, uuid généré à la création de l'élément)

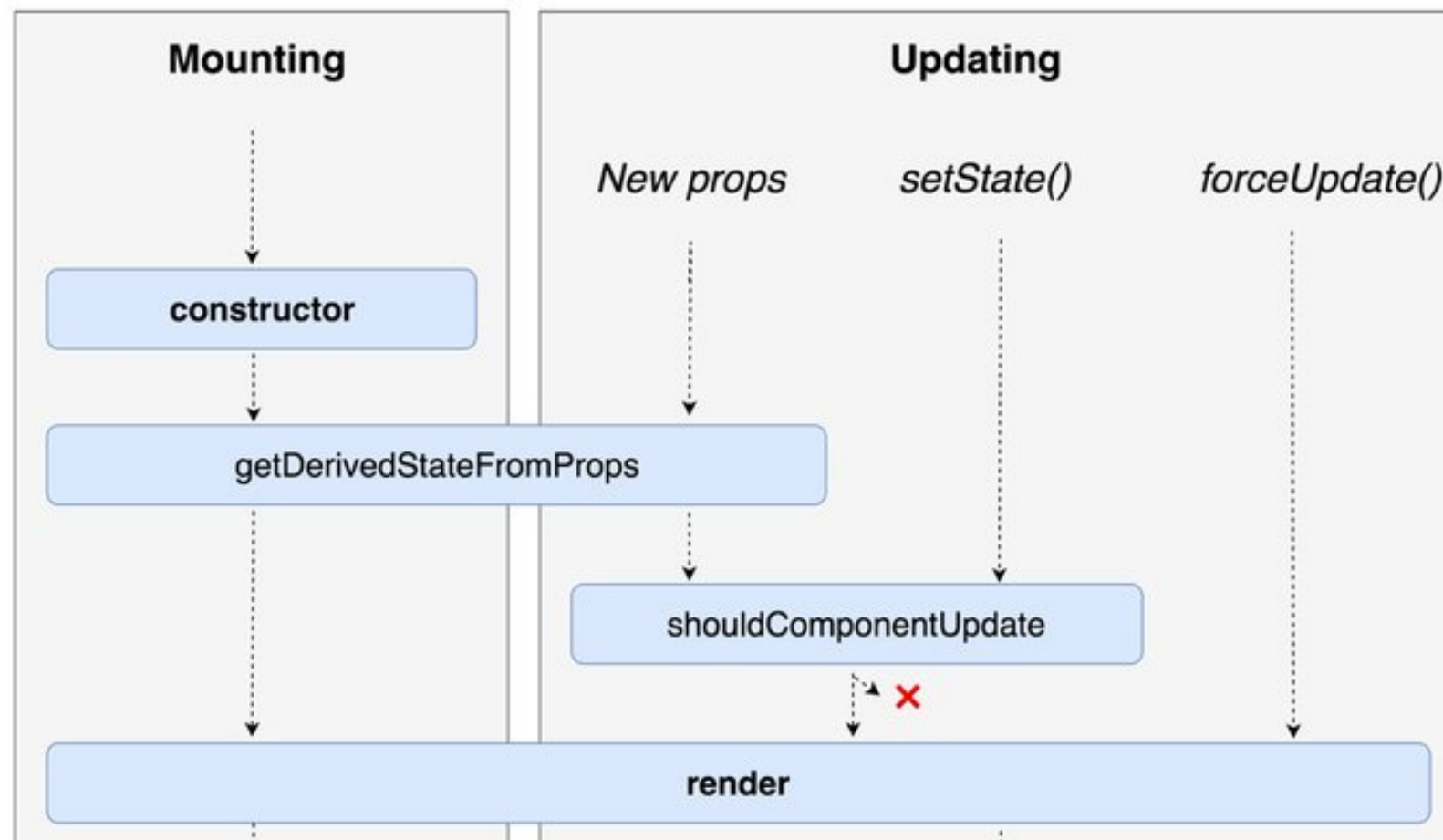
```
function DefinitionList() {  
  const abbrs = { JS: "JavaScript", CSS: "Cascading Style Sheets" };  
  return (  
    <dl>  
      {Object.entries(abbrs).map(([term, definition]) => (  
        <ListItem key={term} term={term} definition={definition} />  
      ))}  
    </dl>  
  );  
}
```

# Optimisation - shouldComponentUpdate



- Lorsque que le state ou les props d'un éléments sont mis à jour, une cascade de render va s'effectuer pour les composants enfants
- Il est possible de bloquer les render liés à l'update d'un élément en créant une méthode `shouldComponentUpdate` sur l'élément

```
shouldComponentUpdate(nextProps) {  
  return this.props.todos !== nextProps.todos;  
}
```



# Optimisation - PureComponent



- Un composant "pur" est un composant contenant une méthode `shouldComponentUpdate` vérifiant que chacune des propriétés est différente de la valeur précédente
- Lorsque qu'on utilise la classe `PureComponent`, il faudra donc mettre à jour les tableaux et les objets de façon "immuable"

```
class TodoList extends PureComponent {  
  render() {  
    const todoItems = this.props.todos.map((todo) => (  
      <TodoItem key={todo.id} todo={todo}  
        onDelete={() => this.props.onDelete(todo)} />  
    ));  
  
    return (  
      <div className="TodoList">  
        {todoItems}  
      </div>  
    );  
  }  
}
```

# Optimisation - Mémoïsation



- La mémoïsation est une technique de programmation qui consiste à mettre en cache le retour d'une fonction lorsque ses paramètres sont inchangés :

<https://fr.wikipedia.org/wiki/Mémoïsation>

- Lodash contient par exemple une fonction memoize

```
const nbs = (new Array(1_000_000)).fill(0).map(() => Math.random());

function findLowerCount(arrayNbs, val) {
  return arrayNbs.filter((el) => el < val).length;
}

// Sans memoisation
console.time('findLowerCount');
console.log(findLowerCount(nbs, 0.5));
console.timeEnd('findLowerCount'); // 71.366ms

console.time('findLowerCount');
console.log(findLowerCount(nbs, 0.5));
console.timeEnd('findLowerCount'); // 60.217ms

console.time('findLowerCount');
console.log(findLowerCount(nbs, 0.5));
console.timeEnd('findLowerCount'); // 43.323ms
```

# Optimisation - Mémoïsation



```
const { memoize } = require('lodash');

function findLowerCount(arrayNbs, val) {
  return arrayNbs.filter((el) => el < val).length;
}

// Avec memoisation
const findLowerCountMemo = memoize(findLowerCount);
console.time('findLowerCountMemo');
console.log(findLowerCountMemo(nbs, 0.5));
console.timeEnd('findLowerCountMemo'); // 71.366ms

console.time('findLowerCountMemo');
console.log(findLowerCountMemo(nbs, 0.5)); // pas de rappel
console.timeEnd('findLowerCountMemo'); // 0.102ms

console.time('findLowerCountMemo');
console.log(findLowerCountMemo(nbs, 0.5)); // pas de rappel
console.timeEnd('findLowerCountMemo'); // 0.045ms
```



# Optimisation - PureComponent



- Un composant "pur" est un composant contenant une méthode `shouldComponentUpdate` vérifiant que chacune des propriétés est différente de la valeur précédente
- Lorsque qu'on utilise la classe `PureComponent`, il faudra donc mettre à jour les tableaux et les objets de façon "immuable"

```
class TodoList extends PureComponent {
  render() {
    const todoItems = this.props.todos.map((todo) => (
      <TodoItem key={todo.id} todo={todo}
        onDelete={() => this.props.onDelete(todo)} />
    ));

    return (
      <div className="TodoList">
        {todoItems}
      </div>
    );
  }
}
```

# Optimisation - Exercice



- Utiliser `React.memo` ou `PureComponent` sur `TodoForm`, `TodoList` et `TodoItem`



# Immuabilité

# Immuabilité - Introduction



- Lors de la modification d'un objet, le changement peut-être muable en modifiant l'objet d'origine ou immuable en créant un nouvel objet
- Les algorithmes de détections de changements préféreront les changements immuables, ayant ainsi juste à comparer les références plutôt que l'ensemble du contenu de l'objet
- Exemple, en JS les tableaux sont muables, les chaines de caractères immuables

```
const firstName = 'Romain';  
firstName.concat('Edouard');  
console.log(firstName); // Romain  
  
const firstNames = ['Romain'];  
firstNames.push('Edouard');  
console.log(firstNames.join(', ')); // Romain, Edouard
```



- Ajouter à la fin

```
const firstNames = ['Romain', 'Edouard'];

function append(array, value) {
  return [...array, value];
}

const newfirstNames = append(firstNames, 'Jean');
console.log(newfirstNames.join(', ')); // Romain, Edouard, Jean
console.log(firstNames === newfirstNames); // false
```

- Ajouter au début

```
const firstNames = ['Romain', 'Edouard'];

function prepend(array, value) {
  return [value, ...array];
}

const newfirstNames = prepend(firstNames, 'Jean');
console.log(newfirstNames.join(', ')); // Jean, Romain, Edouard
console.log(firstNames === newfirstNames); // false
```



- Ajouter à un indice donné

```
const firstNames = ['Romain', 'Edouard'];

function insertAt(array, value, i) {
  return [
    ...array.slice(0, i),
    value,
    ...array.slice(i),
  ];
}

const newfirstNames = insertAt(firstNames, 'Jean', 1);
console.log(newfirstNames.join(', ')); // Romain, Jean, Edouard
console.log(firstNames === newfirstNames); // false
```



- Modifier un élément

```
const firstNames = ['Romain', 'Edouard'];

function modify(array, value, i) {
  return [
    ...array.slice(0, i),
    value,
    ...array.slice(i + 1),
  ];
}

const newfirstNames = modify(firstNames, 'Jean', 1);
console.log(newfirstNames.join(', ')); // Romain, Jean
console.log(firstNames === newfirstNames); // false
```



- Supprimer un élément

```
const firstNames = ['Romain', 'Edouard'];

function remove(array, i) {
  return [
    ...array.slice(0, i),
    ...array.slice(i + 1),
  ];
}

const newfirstNames = remove(firstNames, 1);
console.log(newfirstNames.join(', ')); // Romain
console.log(firstNames === newfirstNames); // false
```





- Ajouter un élément

```
const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

function add(object, key, value) {
  return {
    ...object,
    [key]: value,
  };
}

const newContact = add(contact, 'city', 'Paris');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain","lastName":"Bohdanowicz","city":"Paris"}
console.log(contact === newContact); // false
```



- Modifier un élément

```
const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

function modify(object, key, value) {
  return {
    ...object,
    [key]: value,
  };
}

const newContact = modify(contact, 'firstName', 'Thomas');
console.log(JSON.stringify(newContact));
// {"firstName":"Thomas","lastName":"Bohdanowicz"}
console.log(contact === newContact); // false
```



- Supprimer un élément

```
const contact = {
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
};

function remove(object, key) {
  const { [key]: val, ...rest } = object;
  return rest;
}

const newContact = remove(contact, 'lastName');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain"}
console.log(contact === newContact); // false
```

# Immuabilité - Immutable.js



- Pour simplifier la manipulation d'objets ou de tableaux immuables, Facebook a créé Immutable.js
- Installation  
`npm install immutable`

# Immuabilité - Immutable.js List



- Ajouter à la fin

```
const immutable = require('immutable');  
  
const firstNames = immutable.List(['Romain', 'Edouard']);  
  
const newfirstNames = firstNames.push('Jean');  
console.log(newfirstNames.join(', ')); // Romain, Edouard, Jean  
console.log(firstNames === newfirstNames); // false
```

- Ajouter au début

```
const immutable = require('immutable');  
  
const firstNames = immutable.List(['Romain', 'Edouard']);  
  
const newfirstNames = firstNames.unshift('Jean');  
console.log(newfirstNames.join(', ')); // Jean, Romain, Edouard  
console.log(firstNames === newfirstNames); // false
```

# Immuabilité - Immutable.js List



- Ajouter à un indice donné

```
const immutable = require('immutable');  
  
const firstNames = immutable.List(['Romain', 'Edouard']);  
  
const newfirstNames = firstNames.insert(1, 'Jean');  
console.log(newfirstNames.join(', ')); // Romain, Jean, Edouard  
console.log(firstNames === newfirstNames); // false
```

# Immuabilité - Immutable.js List



- Modifier un élément

```
const immutable = require('immutable');  
  
const firstNames = immutable.List(['Romain', 'Edouard']);  
  
const newfirstNames = firstNames.set(1, 'Jean');  
console.log(newfirstNames.join(', ')); // Romain, Jean  
console.log(firstNames === newfirstNames); // false
```

# Immuabilité - Immutable.js List



- Supprimer un élément

```
const immutable = require('immutable');  
  
const firstNames = immutable.List(['Romain', 'Edouard']);  
  
const newfirstNames = firstNames.delete(1);  
console.log(newfirstNames.join(', ')); // Romain  
console.log(firstNames === newfirstNames); // false
```



# Immuabilité - Immutable.js Map



- Ajouter un élément

```
const immutable = require('immutable');

const contact = immutable.Map({
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
});

const newContact = contact.set('city', 'Paris');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain","lastName":"Bohdanowicz","city":"Paris"}
console.log(contact === newContact); // false
```

# Immuabilité - Immutable.js Map



- Modifier un élément

```
const immutable = require('immutable');

const contact = immutable.Map({
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
});

const newContact = contact.set('firstName', 'Thomas');
console.log(JSON.stringify(newContact));
// {"firstName":"Thomas","lastName":"Bohdanowicz"}
console.log(contact === newContact); // false
```

# Immuabilité - Immutable.js Map



- Supprimer un élément

```
const immutable = require('immutable');

const contact = immutable.Map({
  firstName: 'Romain',
  lastName: 'Bohdanowicz',
});

const newContact = contact.remove('lastName');
console.log(JSON.stringify(newContact));
// {"firstName":"Romain"}
console.log(contact === newContact); // false
```

# Immuabilité - Exercice



- Installer Immutable.js
- Utiliser Immutable.js pour manipuler le tableau dans todosReducer
- Aide : <https://redux.js.org/recipes/using-immutablejs-with-redux>



# Typage statique React

# Typage statique React - Introduction



- Typage statique vs typage dynamique  
JavaScript contrairement à d'autres langages n'offre pas la possibilité de typer statiquement ses variables ou fonctions.
- Pourquoi typer statiquement ?
  - Autocomplétion dans les IDEs modernes (Visual Studio Code, Webstorm...)
  - Détection statique des erreurs dans l'IDE / à la compilation
- Pourquoi type dynamiquement ?
  - Flexibilité, une même instruction / fonction peut-être réutilisée pour plusieurs types
  - Temps de développement, pas avoir à définir statiquement le types des objets par exemple

# Typage statique React - JSDoc



- Les commentaires JSDoc sont bien reconnus par les IDEs modernes (VSCode, Webstorm) voir <http://usejsdoc.org/>
- Un commentaire JSDoc commence par 2 étoiles `/**` commentaire `*/`
- Typer une fonction :

```
/**
 * Reducer of todos
 * @param {object[]} previousState
 * @param {object} action
 * @param {string} action.type
 * @param {object} action.payload
 * @param {number} action.payload.id
 * @param {string} action.payload.text
 * @param {boolean} action.payload.completed
 */
function todosReducer(previousState = [], { type, payload }) {
  switch (type) {
    case TODO_ADD:
      return [...previousState, payload];
    default:
      return {
        completed: (property) completed: boolean ⓘ
        id
        text
      }
  }
}
```

# Typage statique React - JSDoc



- Pour typer les paramètres d'entrées et de retour d'une fonction

```
/**
 * Reducer of todos
 * @param {object[]} previousState
 * @param {object} action
 * @param {string} action.type
 * @param {object} action.payload
 * @param {number} action.payload.id
 * @param {string} action.payload.text
 * @param {boolean} action.payload.completed
 * @returns {object[]}
 */
function todosReducer(previousState = [], { type, payload }) {
  switch (type) {
    case TODO_ADD:
      return [...previousState, payload];
    default:
      return previousState;
  }
}
```



# Typage statique React - JSDoc



- Définir des types réutilisables

```
/**
 * @typedef Todo
 * @property {number} id
 * @property {string} text
 * @property {boolean} completed
 */

/**
 * @typedef TodoAction
 * @property {string} type
 * @property {Todo} payload
 */

/**
 * Reducer of todos
 * @param {Todo[]} previousState
 * @param {TodoAction} action
 * @returns {Todo[]}
 */
function todosReducer(previousState = [], { type, payload }) {
  switch (type) {
    case TODO_ADD:
      return [...previousState, payload];
    default:
      return previousState;
  }
}
```

- Typen des variables

```
/** @type {string[]} items */  
let items = this.props.items;
```

```
/** @type {string[]} items */  
let items = this.props.items;
```

```
items = items.map(item => {
  const classes = classNames(css.item, {
    [css.itemSelected]: item. === selected
  });
});
```

# Typage statique React - JSDoc



- Importer des types provenant d'autres fichiers

```
/** @type {import('webpack').Configuration} */  
const config = {};
```

- Nécessite d'écrire le code importé en TypeScript ou de créer des interfaces TypeScript supplémentaires.
- Le projet DefinitelyTyped permet de trouver des interfaces TypeScript pour la plupart des projets open-source :  
<https://github.com/DefinitelyTyped/DefinitelyTyped>  
(Dans le top 10 des projets en nombre de contributeurs <https://octoverse.github.com/projects#repositories>)

```
/** @type {import('webpack').Configuration} */  
const config = {  
  amd: (property) webpack.Configuration.a  
  bail  
  // ...  
}
```

# Typage statique React - PropTypes



- Pour typer des composants React on peut utiliser PropTypes
- Inclus dans React jusqu'à la version 15, dans un paquet npm séparé depuis la 16
- Installation  
npm install prop-types

```
import React from 'react';
import { string } from 'prop-types';

function Hello({name}) {
  return (
    <div className="Hello">
      Hello {name}
    </div>
  );
}



Hello.propTypes = {
  name: string,
};

export { Hello };
```

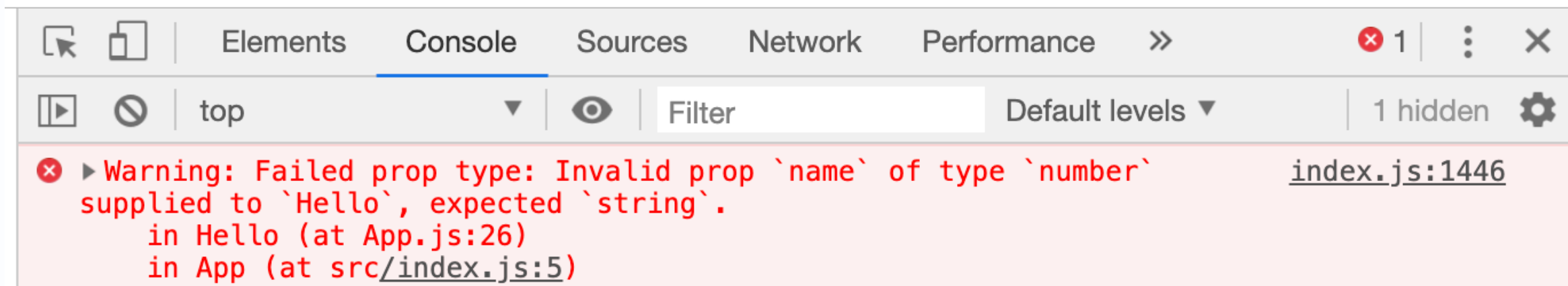
# Typage statique React - PropTypes



- Complétion améliorée depuis le JSX :

```
<Hello na| />  
<Clock />  name? (JSX attribute) name?: string 
```

- Warning dans les DevTools du navigateur si on passe le mauvais type



# Typage statique React - PropTypes



- Il est possible également de valider avec `isRequired` et de définir ses propres validateurs :

```
Contact.propTypes = {  
  name: PropTypes.string.isRequired,  
  age(props, propName, component) {  
    if (props[propName] && (props[propName] < 0 || props[propName] > 120)) {  
      return new Error(`${propName} should be between 0 and 120`)  
    }  
  },  
};
```

- Documentation :  
<https://github.com/facebook/prop-types>
- Airbnb propose aussi ses validateurs :  
<https://github.com/airbnb/prop-types>

# Typage statique React - Flow



- Les commentaires JSDoc ne préviennent pas d'erreur potentielles, PropTypes affiche des warnings au moment de l'exécution
- Pour détecter statiquement des erreurs dans l'IDE ou au moment du build, Facebook propose un analyseur de type statique appelé Flow
- Flow est supporté par Create React App

# Typage statique React - Flow



- Installation

```
npm i flow-bin -D
```

- Création d'un script dans le fichier package.json

```
"scripts": {  
  "start": "react-scripts start",  
  "build": "react-scripts build",  
  "flow": "flow"  
}
```

- Création du fichier de configuration

```
npm run flow init
```

- Installer l'extension VSCode Flow-Language-Support

- Désactiver la validation JavaScript de VSCode :

```
{  
  "files.autoSave": "onFocusChange",  
  "javascript.validate.enable": false  
}
```



# Typage statique React - Flow



- Pour activer Flow il suffit ensuite d'utiliser le commentaire @flow

```
// @flow
function square(n: number): number {
  return n * n;
}

square("2"); // Error!
```

# Typage statique React - Flow



- Typage des objets (? pour les propriétés optionnelles)

```
function Hello({name = ''}: {name: ?string}) {  
  return (  
    <div className="Hello">  
      Hello {name}  
    </div>  
  );  
}
```

- Typage avec des interfaces

```
interface HelloProps {  
  name: ?string;  
}  
  
function Hello({name = ''}: HelloProps) {  
  return (  
    <div className="Hello">  
      Hello {name}  
    </div>  
  );  
}
```

# Typage statique React - TypeScript



- Create React App inclus le support de TypeScript depuis la version 2.1
- A la création du projet  
`create-react-app mon-projet --template typescript`
- Avantages
  - Language avec concepts supplémentaires (public/private/protected/décorateurs...)
  - Intégrations avec des bibliothèques TypeScript ou les fichiers DefinitelyTyped (imports de types...) / Intégration avec les IDEs
  - Popularité : <https://www.npmtrends.com/flow-bin-vs-typescript>
- Inconvénients
  - Flow peut s'appliquer qu'à certains fichiers
  - Flow peut s'utiliser sous forme de commentaire
  - Plus lourd à intégrer à un projet existant
- <https://github.com/niieani/typescript-vs-flowtype>

# Typage statique React - Exercice



- Récupérer le projet my-app sur <https://gitlab.com/react-avance>
- Installer les dépendances
- Ajouter les PropTypes sur les composants TodoForm, TodoList et TodoItem
- Ajouter les Annotations Flow dans tous les fichiers du dossier src/todos



# TypeScript

# TypeScript - Introduction



- TypeScript : JavaScript + Typage statique
  - TypeScript est un langage créé par Microsoft, construit comme un sur-ensemble d'ECMAScript
  - Pour pouvoir exécuter le code il faut le transformer en JavaScript avec un compilateur
  - A quelques exceptions près et selon la configuration, le JavaScript est valide en TypeScript
  - Le principal intérêt de TypeScript est l'ajout d'un typage statique

# TypeScript - Installation



- Installation
  - `npm install -g typescript`
- Création d'un fichier de configuration
  - `tsc --init`
- Compilation
  - `tsc`

# TypeScript - Typage statique



- Le principal intérêt de TypeScript est l'introduction d'un typage statique

```
const lastName: string = 'Bohdanowicz';  
const age: number = 32;  
const isTrainer: boolean = true;
```

- Types basiques :

- *boolean*
- *number*
- *string*



# TypeScript - Typage statique



- Avantages

- Complétion

```
const firstName: string = 'Romain';

function hello(firstName: string): string {
  return `Hello ${firstName}`;
}
```

- charAt(pos: number)
- charCodeAt(index: number)
- concat(... strings: string)
- indexOf(searchString: string,

- Détection des erreurs

```
const firstName: string = 'Romain';

function hello(firstName: string): string {
  return `Hello ${firstName}`;
}

hello({
  firstName: 'Romain',
});
```

# TypeScript - Typage statique



## ▸ Tableaux

```
const firstNames: string[] = ['Romain', 'Edouard'];  
const colors: Array<string> = ['blue', 'white', 'red'];
```

## ▸ Tuples

```
const email: [string, boolean] = ['romain.bohdanowicz@gmail.com', true];
```

## ▸ Enum

```
enum Choice {Yes, No, Maybe}  
  
const c1: Choice = Choice.Yes;  
const choiceName: string = Choice[1];
```

## ▸ Never

```
function error(message: string): never {  
    throw new Error(message);  
}
```

# TypeScript - Typage statique



## ▸ Any

```
let anyType: any = 12;  
anyType = "now a string string";  
anyType = false;  
anyType = {  
  firstName: 'Romain'  
};
```

## ▸ Void

```
function withoutReturn(): void {  
  console.log('Do something')  
}
```

## ▸ Null et undefined

```
let u: undefined = undefined;  
let n: null = null;
```

# TypeScript - Assertion de type



- Le compilateur ne peut pas toujours déterminer le type adéquat :

```
const formElt = document.querySelector('#myForm');  
const url = formElt.action; // error TS2339: Property 'action' does not exist on  
type 'Element'.
```

- Il faut alors lui préciser, 3 syntaxes possibles

```
let formElt = <HTMLFormElement> document.querySelector('#myForm');  
const url = formElt.action;
```

```
let formElt = document.querySelector<HTMLFormElement>('#myForm');  
const url = formElt.action;
```

```
let formElt = document.querySelector('#myForm') as HTMLFormElement;  
const url = formElt.action;
```

# TypeScript - Inférence de type



- TypeScript peut parfois déterminer automatiquement le type :

```
const title = 'First Names';  
console.log(title.toUpperCase());  
  
const names = ['Romain', 'Edouard'];  
for (let n of names) {  
    console.log(n.toUpperCase());  
}
```

# TypeScript - Interfaces



- Pour documenter un objet on utilise une interface
  - Anonyme

```
function helloInterface(contact: {firstName: string}) {  
    console.log(`Hello ${contact.firstName.toUpperCase()}`);  
}
```

- Nommée

```
interface ContactInterface {  
    firstName: string;  
}  
  
function helloNamedInterface(contact: ContactInterface) {  
    console.log(`Hello ${contact.firstName.toUpperCase()}`);  
}
```

# TypeScript - Interfaces



- Les propriétés peuvent être :
  - optionnelles (ici *lastName*)
  - en lecture seule, après l'initialisation (ici *age*)
  - non déclarées (avec les crochets)

```
interface ContactInterface {  
  firstName: string;  
  lastName?: string;  
  readonly age: number;  
  [propName: string]: any;  
}  
  
function helloNamedInterface(contact: ContactInterface) {  
  console.log(`Hello ${contact.firstName.toUpperCase()}`);  
}
```



- Quelques différences avec JavaScript sur le mot clé class
  - On doit déclarer les propriétés
  - On peut définir une visibilité pour chaque membre : *public*, *private*, *protected*

```
class Contact {  
  private firstName: string;  
  
  constructor(firstName: string) {  
    this.firstName = firstName;  
  }  
  
  hello() {  
    return `Hello my name is ${this.firstName}`;  
  }  
}  
  
const romain = new Contact('Romain');  
console.log(romain.hello()); // Hello my name is Romain
```





- Une classe peut
  - Hériter d'une autre classe (comme en JS)
  - Implémenter une interface
  - Être utilisée comme type

```
interface Writable {  
  write(data: string): void;  
}  
  
class FileLogger implements Writable {  
  write(data: string): Writable {  
    console.log(`Write ${data}`);  
    return this;  
  }  
}
```



- Permet de paramétrer le type de certaines méthodes

```
class Stack<T> {  
  private data: Array<T> = [];  
  push(val: T) {  
    this.data.push(val);  
  }  
  pop(): T {  
    return this.data.pop();  
  }  
  peek(): T {  
    return this.data[this.data.length - 1];  
  }  
}  
  
const strStack = new Stack<string>();  
strStack.push('html');  
strStack.push('body');  
strStack.push('h1');  
console.log(strStack.peek().toUpperCase()); // H1  
console.log(strStack.pop().toUpperCase()); // H1  
console.log(strStack.peek().toUpperCase()); // BODY
```

# TypeScript - Décorateurs



- Permettent l'ajout de fonctionnalités aux classes ou membre d'une classe en annotant plutôt que via du code à l'utilisation
- Norme à l'étude en JavaScript par le TC39  
<https://github.com/tc39/proposal-decorators>
- Supporté de manière expérimentale en TypeScript
- Pour activer leur support il faut éditer le tsconfig.json ou passer une option au compilateur

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "experimentalDecorators": true  
  }  
}
```

# TypeScript - Décorateurs



## ▸ Décorateur de classes

```
'use strict';

function Freeze(obj) {
  Object.freeze(obj);
}

@Freeze
class MyMaths {
  static sum(a, b) {
    return Number(a) + Number(b);
  }
}

try {
  MyMaths['subtract'] = function(a, b) {
    return a - b;
  };
}
catch(err) {
  // Cannot add property subtract, object is not extensible
  console.log(err.message);
}
```



## ▸ Décorateur de propriétés

```
import 'reflect-metadata';

const minLengthMetadataKey = Symbol("minLength");

function MinLength(length: number) {
  return Reflect.metadata(minLengthMetadataKey, length);
}

function validateMinLength(target: any, propertyKey: string): boolean {
  const length = Reflect.getMetadata(minLengthMetadataKey, target, propertyKey);
  return target[propertyKey].length >= length;
}

class Contact {
  @MinLength(7)
  protected firstName;

  constructor(firstName: string) {
    this.firstName = firstName;
  }

  isValid(): boolean {
    return validateMinLength(this, 'firstName');
  }
}

const romain = new Contact('Romain');
console.log(romain.isValid()); // false
```



# Redux Avancé

# Redux Avancé - Rappels



- Actions
- Actions Creators
- Constants
- Reducers
- Selectors
- `mapStateToProps`
- `dispatch`
- `mapDispatchToProps`

# Redux Avancé - Redux Form



- Redux Form est une bibliothèque qui simplifie la gestion des formulaires en lien avec Redux, notamment la validation des champs
- Installation  
npm i redux-form
- Utilisation du reducer de redux-form

```
import { combineReducers } from 'redux';  
import { reducer as reduxFormReducer } from 'redux-form';  
  
export const rootReducer = combineReducers({  
  // ...  
  form: reduxFormReducer,  
});
```





## ▸ Exemple

```
const ContactForm = (props) => {
  const { invalid, submit } = props;
  return (
    <div>
      <Field
        name="prenom"
        type="text"
        component={renderField}
        label="Prénom de l'élève"
      />
      <Field
        name="nom"
        type="text"
        component={renderField}
        label="Nom de l'élève"
      />
      <NextButton disabled={invalid} onClick={submit}>Envoyer</NextButton>
    </div>
  );
};

export const Form = reduxForm({
  form: 'contact',
  validate,
  destroyOnUnmount: false,
})(ContactForm);
```

# Redux Avancé - Redux Persist



- Permet de faire persister le state entre 2 démarrages de l'application
- Le state sera automatiquement stocké dans le localStorage, sessionStorage...
- Documentation  
<https://github.com/rt2zz/redux-persist>

```
import { persistStore, persistReducer } from 'redux-persist';
import storage from 'redux-persist/lib/storage';

import rootReducer from './reducers';

const persistConfig = {
  key: 'root',
  storage,
};

const persistedReducer = persistReducer(persistConfig, rootReducer);
```

# Redux Avancé - Redux Persist



- Le composant PersistGate permet de passer la version précédente du state à l'application

```
import { PersistGate } from 'redux-persist/integration/react'

function App() {
  return (
    <Provider store={store}>
      <PersistGate loading={null} persistor={persistor}>
        <RootComponent />
      </PersistGate>
    </Provider>
  );
};
```

# Redux Avancé - Middleware



- Un middleware est un plugin qui est exécuté à chaque dispatch. Il permet d'accéder au store et à l'action
- Pour passer au prochain middleware on utilise la fonction next comme ci-dessous

```
import { createStore, combineReducers, applyMiddleware } from "redux";
import { composeWithDevTools } from "redux-devtools-extension";

const rootReducer = combineReducers({
  // ...
});

const logger = store => next => action => {
  console.group(action.type);
  console.info("dispatching", action);
  let result = next(action);
  console.log("next state", store.getState());
  console.groupEnd();
  return result;
};

const store = createStore(
  rootReducer,
  composeWithDevTools(applyMiddleware(logger))
);
```



- react-persist
  - Ajouter react-persist pour faire persister le state dans le localStorage  
<https://github.com/rt2zz/redux-persist#basic-usage>
- redux-form
  - En prenant exemple sur <https://redux-form.com/8.2.1/examples/syncvalidation/>
  - Créer un formulaire avec des champs : name, email et phone
  - Ecrire une fonction de validation
  - Ecrire une fonction de rendu pour chaque élément (name type="text", email type="email", phone type="phone")
- redux-thunk
  - Au submit du formulaire, faire un dispatch d'une action userCreateRequested, cette action fera :
    - dispatch de userCreate (loading: true)
    - La requête POST <https://jsonplaceholder.typicode.com/users>
    - dispatch de userCreateSuccess (loading: false) (facultatif userCreateError)



# Tests Automatisés

# Tests Automatisés - Introduction



- Comment tester son code ?
  - Manuellement : une personne effectue les tests
  - Automatiquement : les tests ont été programmés
- Historique
  - à partir de 1989 en Smalltalk et le framework SUnit
  - à partir de 1997 en Java avec JUnit
  - à partir de 2004 dans le navigateur avec Selenium

# Tests Automatisés - Pourquoi ?



- Pourquoi automatiser les tests ?
  - plus l'application grandit, plus le risque d'introduire une régression est grand  
ex: modifier une fonction qui est partagé par différentes
  - tester manuellement à chaque itération prendra à terme plus de temps qu'écrire le code du test
  - les tests automatisés peuvent se lancer sur différentes plate-formes et navigateurs très simplement
  - les tests aident à la compréhension du code, les lire permet de comprendre des comportements qui n'ont pas toujours été documentés
- Pourquoi tester manuellement ?
  - certains tests peuvent être simple à faire manuellement mais compliqués à automatiser (drag-n-drop...)
  - automatiser permet d'avoir accès à des choses inaccessible manuellement (bouton caché par une popup...)



# Tests Automatisés - Types de tests



- Types de tests :
  - tests de code statiques / linters
  - tests de code dynamiques / tests unitaires...
  - tests de déploiement
  - tests de sécurité
  - tests de montée en charge
  - ...

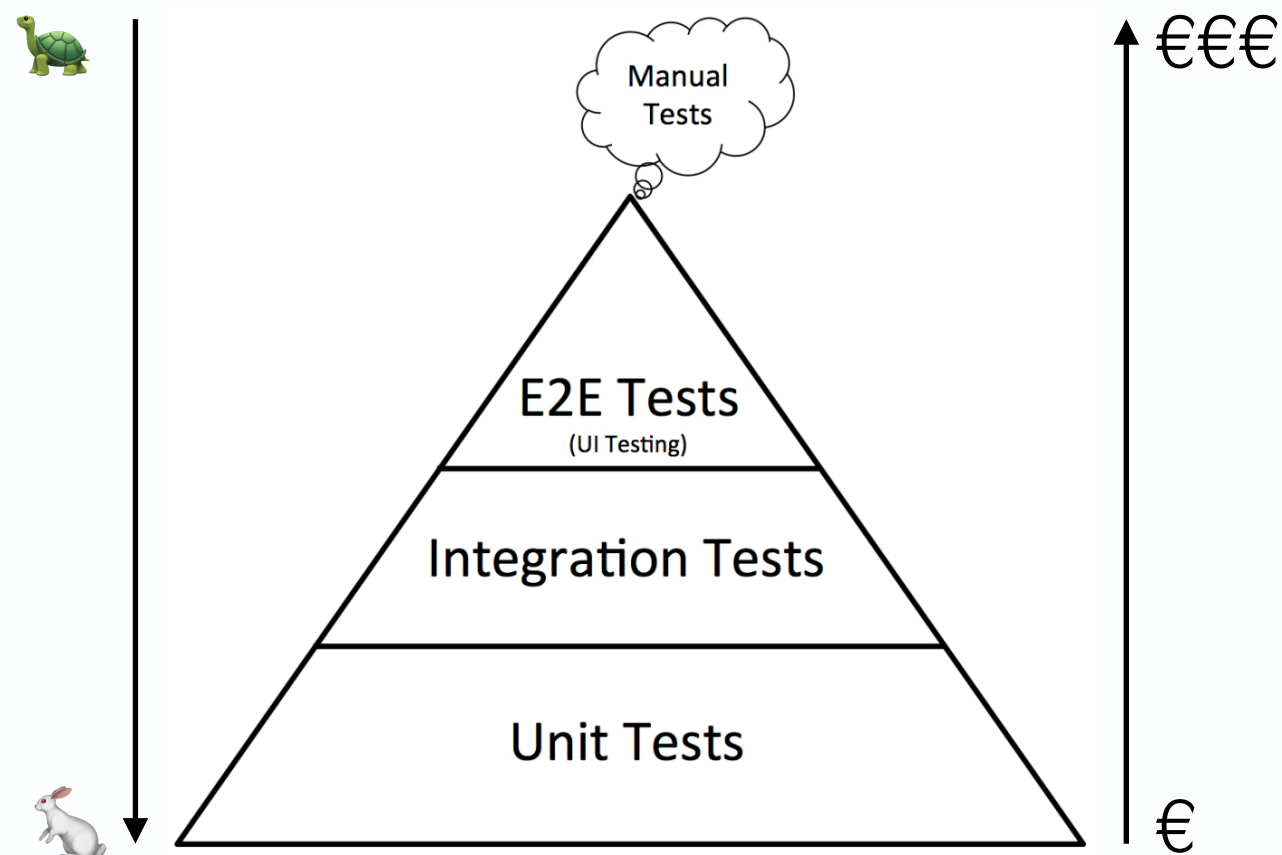
# Tests Automatisés - Pyramide des tests



- 3 types de tests automatisés au niveau code côté Front :
  - Test unitaire  
Permet de tester les briques d'une application (classes / fonctions)
  - Test d'intégration  
Teste que les briques fonctionnent correctement ensemble
  - Test End-to-End (E2E)  
Vérifie l'application dans le client

- Un pyramide

- plus le test est haut plus il est lent
- plus le test est haut plus il coûte cher



# Tests Automatisés - Quand exécuter ?



- Quand exécuter ?
  - tout le temps si on arrive à maintenir des tests performants (max 1-2 minutes)
  - avant un commit
  - avant un push
  - sur une plateforme d'intégration ou de déploiement continue (CI/CD)



- Ou placer ses tests ?
  - dans le même répertoire que le code testé
  - dans un répertoire *test* en préservant l'arborescence du répertoire *src*
  - dans un répertoire *test* sans lien avec l'arborescence



# Tests avec Jest

# Tests avec Jest - Introduction



- Framework de test créé en 2014 par Facebook
- Sous Licence MIT depuis septembre 2017
- Permet de lancer des tests :
  - unitaires / d'intégration (dans Node.js)
  - fonctionnels / E2E (via Puppeteer)
- Peut s'utiliser avec ou sans configuration
- Les tests se lancent en parallèle dans les Workers Node.js
- Intègre par défaut :
  - Calcul de coverage (via Istanbul)
  - Mocks (natifs ou en installant Sinon.JS)
  - Snapshots

# Tests avec Jest - Installation



- Installation

```
npm install --save-dev jest
```

```
yarn add --dev jest
```
- Déjà intégré à Create React App

# Tests avec Jest - Hello, world !



- Sans configuration, les tests doivent se trouver dans un répertoire `__tests__`, ou bien se nommer `*.test.js` ou `*.spec.js`

```
// src/hello.js
const hello = (name = 'World') => `Hello ${name} !`;

module.exports = hello;
```

```
// __tests__/hello.js
const hello = require('../src/hello');

test('Hello, world !', () => {
  expect(hello()).toBe('Hello World !');
  expect(hello('Romain')).toBe('Hello Romain !');
});
```



# Tests avec Jest - Lancements des tests



- Si Jest localement  
node\_modules/.bin/jest
- Si Jest globalement  
jest
- Avec un script test dans package.json  
npm run test  
npm test  
npm t

```
// package.json
{
  "devDependencies": {
    "jest": "^22.0.6"
  },
  "scripts": {
    "test": "jest"
  }
}
```

```
MacBook-Pro:hello-jest romain$ node_modules/.bin/jest
```

```
PASS __tests__/hello.js
  ✓ Hello, world ! (3ms)
```

```
Test Suites: 1 passed, 1 total
```

```
Tests: 1 passed, 1 total
```

```
Snapshots: 0 total
```

```
Time: 0.701s, estimated 1s
```

```
Ran all test suites.
```

# Tests avec Jest - Watchers



- En mode Watch

```
node_modules/.bin/jest --watchAll
```

```
jest --watchAll
```

```
npm t -- --watchAll
```

```
MacBook-Pro:hello-jest romain$ npm t -- --watchAll
```

```
PASS __tests__/hello.js
```

```
PASS __tests__/calc.js
```

```
Test Suites: 2 passed, 2 total
```

```
Tests: 3 passed, 3 total
```

```
Snapshots: 0 total
```

```
Time: 0.65s, estimated 1s
```

```
Ran all test suites.
```

## Watch Usage

- > Press f to run only failed tests.
- > Press o to only run tests related to changed files.
- > Press p to filter by a filename regex pattern.
- > Press t to filter by a test name regex pattern.
- > Press q to quit watch mode.
- > Press Enter to trigger a test run.

# Tests avec Jest - Coverage



- Avec calcul du coverage

```
node_modules/.bin/jest --coverage
jest --coverage
npm t -- --coverage
```
- Par défaut le coverage s'affiche dans la console et génère des fichiers Clover, JSON et HTML dans le dossier coverage

```
MacBook-Pro:hello-jest romain$ npm t -- --coverage
```

```
PASS  __tests__/calc.js
PASS  __tests__/hello.js
```

```
Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        0.722s, estimated 1s
Ran all test suites.
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
All files	86.67	100	60	100	
calc.js	83.33	100	50	100	
hello.js	100	100	100	100	

# Tests avec Jest - Mocks



- Jest intègre par défaut une bibliothèque de Mocks

```
// __tests__/Array.prototype.forEach.js
const names = ['Romain', 'Edouard'];

test('Array forEach method', () => {
  const mockCallback = jest.fn();
  names.forEach(mockCallback);
  expect(mockCallback.mock.calls.length).toBe(2);
  expect(mockCallback).toHaveBeenCalledTimes(2);
});
```

# Tests avec Jest - Tester les timers



- La fonction `jest.useFakeTimers()` transforme les timers (`setTimeout`, `setInterval`...) en mock

```
// src/timeout.js
const timeout = (delay, arg) => {
  return new Promise((resolve) => {
    setTimeout(resolve, delay, arg);
  });
};

module.exports = timeout;
```

```
// __tests__/timeout.js
jest.useFakeTimers();

const timeout = require('../src/timeout');

test('waits 1 second', () => {
  const arg = timeout(10000, 'Hello');

  expect(setTimeout).toHaveBeenCalledTimes(1);
  expect(setTimeout).toHaveBeenLastCalledWith(expect.any(Function), 10000,
  'Hello');
});
```

# Tests avec Jest - React



- Une application créée avec create-react-app est déjà configurée pour fonctionner avec React
- Sinon il faudrait installer des dépendances comme babel, babel-jest...  
<https://facebook.github.io/jest/docs/en/tutorial-react.html>

```
// src/App.js
import React, { Component } from 'react';
import { Hello } from './Hello';
import { CounterButton } from './CounterButton';

class App extends Component {
  render() {
    return (
      <div>
        <Hello firstName="Romain" />
        <hr />
        <CounterButton/>
      </div>
    );
  }
}

export default App;
```



- Pour tester un composant React il faut en faire le rendu

```
// src/App.test.js
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App';

it('renders without crashing', () => {
  const div = document.createElement('div');
  ReactDOM.render(<App />, div);
});
```

- 2 inconvénients ici :
  - Nécessite que document existe (exécuter les tests dans un navigateur ou utiliser des implémentations de document côté Node.js comme JSDOM)
  - Les composants enfants du composant testé seront également rendu, le test n'est pas un test unitaire mais un test d'intégration

# Tests avec Jest - Snapshot Testing



- Facebook fourni un paquet npm pour simplifier les tests : react-test-renderer
- Ici on fait un simple Snapshot, c'est à dire une capture du rendu du composant, si lors d'un test futur le rendu est modifié le test échoue

```
import React from 'react';
import renderer from 'react-test-renderer';
import { Hello } from './Hello';

test('it renders like last time', () => {
  const tree = renderer
    .create(<Hello />)
    .toJSON();
  expect(tree).toMatchSnapshot();
});
```



# Tests avec Jest - Shallow Rendering



- On peut également faire appel à `ShallowRenderer` qui ne va faire qu'un seul niveau de rendu, et donc rendre le test unitaire

```
// src/App.test.js
import React from 'react';
import App from './App';
import ShallowRenderer from 'react-test-renderer/shallow';
import { Hello } from './Hello';
import { CounterButton } from './CounterButton';

it('renders without crashing', () => {
  const renderer = new ShallowRenderer();
  renderer.render(<App />);
  const result = renderer.getRenderOutput();

  expect(result.type).toBe('div');
  expect(result.props.children).toEqual([
    <Hello firstName="Romain" />,
    <hr />,
    <CounterButton />,
  ]);
});
```

# Tests avec Jest - Enzyme



- Facebook recommande également l'utilisation de la bibliothèque Enzyme, créée par AirBnB.
- Elle fournit un API haut niveau (proche de jQuery) pour manipuler les tests des composants

```
import React from 'react';
import { Hello } from './Hello';
import { shallow } from 'enzyme';

test('it renders without crashing with enzyme', () => {
  shallow(<Hello />);
});

test('it renders without crashing with enzyme', () => {
  const wrapper = shallow(<Hello />);
  expect(wrapper.contains(<div>Hello !</div>)).toEqual(true);
});

test('it renders without crashing with enzyme', () => {
  const wrapper = shallow(<Hello firstName="Romain"/>);
  expect(wrapper.contains(<div>Hello Romain !</div>)).toEqual(true);
});
```

# Tests avec Jest - Tester des événements



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

export class CounterButton extends Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState({
      count: this.state.count + 1,
    });
  }

  render() {
    return (
      <button onClick={this.handleClick}>{this.state.count}</button>
    );
  }
}
```

# Tests avec Jest - Tester des événements



```
import React from 'react';
import { CounterButton } from './CounterButton';
import { shallow } from 'enzyme';

test('it renders without crashing', () => {
  shallow(<CounterButton />);
});

test('it contains 0 at first rendering', () => {
  const wrapper = shallow(<CounterButton />);
  expect(wrapper.text()).toBe('0');
});

test('it contains 1 after click', () => {
  const wrapper = shallow(<CounterButton />);
  wrapper.simulate('click');
  expect(wrapper.text()).toBe('1');
});
```

# Tests avec Jest - Exercices



- Tester unitairement les fonctions liées à Redux :
  - Actions Creators
  - Selectors
  - Reducers
- Tester les composants React (pas les containers) avec Enzyme et les mocks



# Animations/Transitions

# Animations/Transitions - react-transition-group



- Bibliothèque intégrée à React à l'origine puis confié à la communauté
- Inspirée par ng-animate d'AngularJS
- Installation  
npm i react-transition-group

# Animations/Transitions - react-transition-group



## ▸ Examples

```
function App() {  
  const [inProp, setInProp] = useState(false);  
  return (  
    <div>  
      <CSSTransition in={inProp} timeout={200} classNames="my-node">  
        <div>  
          {"I'll receive my-node-* classes"}  
        </div>  
      </CSSTransition>  
      <button type="button" onClick={() => setInProp(true)}>  
        Click to Enter  
      </button>  
    </div>  
  );  
}
```



# Animations/Transitions - Principales Bibliothèques



- Quelques ressources
  - Comparatif de 20 bibliothèque d'animation pour React  
<https://bashooka.com/coding/20-useful-react-animation-libraries/>
  - Un même exemple écrit avec 7 bibliothèques d'animations  
<https://github.com/aholachek/react-animation-comparison>



# Internationalisation (i18n)



- Les 2 principales bibliothèques d'internationalisation sous React sont
  - react-intl
  - react-i18next
- Nous allons voir react-i18next qui propose certaines fonctionnalités plus avancées comme l'extraction des clés de traduction



- react-i18next dépend de i18next  
`npm i i18next react-i18next`
- De nombreux plugins existent  
<https://www.i18next.com/overview/plugins-and-utils>
- i18next-xhr-backend  
Permet de récupérer les clés de traduction avec des requêtes XHR  
`npm i i18next-xhr-backend`

# i18n - Configuration



- Dans le fichier index.js

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import i18nextXHRBackend from 'i18next-xhr-backend';

i18n
  .use(initReactI18next)
  .use(i18nextXHRBackend)
  .init({
    lng: 'fr',
    fallbackLng: 'fr',
    interpolation: {
      escapeValue: false, // react echappe déjà
    },
  });
```



- Les fichiers de traduction doivent être placés dans `public/locales/{{lng}}/translation.json`

```
// public/locales/en/translation.json
{
  "title": "Welcome to react using react-i18next",
  "description": {
    "part1": "To get started, edit <1>src/App.js</1> and save to reload.",
    "part2": "Switch language between english and german using buttons above."
  }
}
```



- Pour appeler les traductions on peut utiliser un hook, un composant ou un hoc
- Exemple avec le hook :

```
import React from 'react';
import { useTranslation } from 'react-i18next';

export function TodoCount({ count }) {
  const { t } = useTranslation();
  return (
    <div className="TodoCount">
      {count > 1 ? count + ' todos' : count + ' todo'} {t('remaining')}
      {t('{{count}} todo(s) remaining', { count })}
    </div>
  );
}
```



- Installation

`npm i -D i18next-scanner`

Lancer la commande via un script npm

```
// i18next-scanner.config.js
module.exports = {
  input: ['src/**/*.js', 'src/**/*.jsx', '!src/**/*.spec.js', '!src/**/*.spec.jsx'],
  options: {
    func: {
      list: ['t'],
    },
    lngs: ['fr', 'en'],
    resource: {
      loadPath: 'public/locales/{{lng}}/translation.json',
      savePath: 'public/locales/{{lng}}/translation.json',
    },
  },
};
```