Formation JavaScript React Redux

Romain Bohdanowicz
Twitter : @bioub - Github : https://github.com/bioub
http://formation.tech/

formation.tech

Introduction

Présentations

- ## Romain Bohdanowicz
  Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle
- ## Expérience
  Formateur/Développeur Freelance depuis 2006
  Plus de 9500 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
- ## Particularités
  Premier site web à 12 ans (HTML/JS/PHP), Triathlète à mes heures perdues
- ## Et vous ?
  Langages ? Expérience ? Utilité de cette formation ?

formation.tech

React

React - Introduction

- React est une bibliothèque de création de composants capables d'être « rendus » (render en anglais) à chaque changement d'état
- React n'offre pas d'architecture comme MVC, on organise en général ses composants autour d'un concept nommé Flux

- Créée par un employé Facebook en 2011
- Rendue Open-Source en 2013
- Licence MIT depuis novembre 2017

React - Installation

- Pour mettre en place rapidement un environnement React fonctionnel, on peut utiliser le package `create-react-app`
- Installation avec npm :
  `npm install -g create-react-app`
- Installation avec Yarn :
  `yarn add create-react-app`
- Pour créer le projet :
  `create-react-app NOM_DU_DOSSIER`
- Créer le projet directement
  `npx create-react-app NOM_DU_DOSSIER`
  `npm init react-app NOM_DU_DOSSIER`
  `yarn create react-app NOM_DU_DOSSIER`

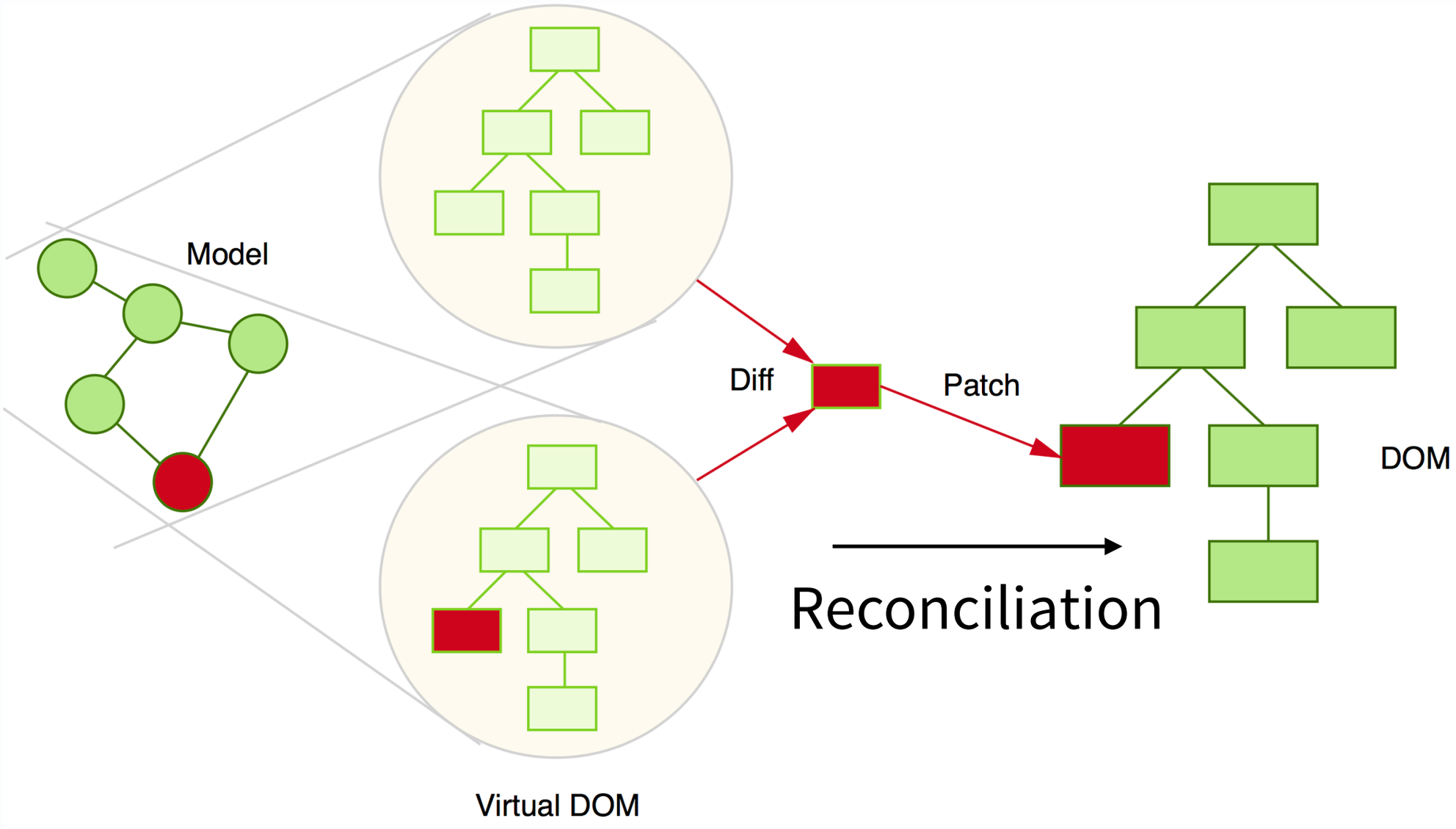
```
My-React-Project
├── README.md
├── node_modules
│   └── ...
├── package.json
├── public
│   ├── favicon.ico
│   ├── index.html
│   └── manifest.json
├── src
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   ├── logo.svg
│   └── registerServiceWorker.js
└── yarn.lock
```

React - Premier composant

```
import React from 'react';
import ReactDOM from 'react-dom';

const App = () => <h1 className="my-app">Hello</h1>;

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

- 2 dépendances :
  - react : permet la création de composants
  - react-dom : permet le rendu de ces composants dans le contexte du DOM
- Notre premier composant App est une simple fonction, qui retourne une syntaxe proche du HTML appelée JSX (l'import de React est obligatoire dans ce cas)
- Par convention les composants React commencent par une majuscule (si vous ne le faites pas, React voit du HTML)

React - Virtual DOM



React - JSX

- Documentation
  https://facebook.github.io/jsx/
- Language proche du HTML nécessitant une compilation (avec Babel par exemple et son plugin babel-plugin-transform-react-jsx)
- Exemple précédent sans JSX :

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  React.createElement('h1', {className: 'my-app'}, 'Hello'),
  document.getElementById('root'),
);
```

React - JSX

- Conditions en JSX
  - if simple

```
import React from 'react';

export const Todo = (props) => {
  return (
    <div>
      <input value={props.value} />
      {props.isDeletable && <button>-</button>}
    </div>
  );
};
```

  - if … else

```
import React from 'react';

export const Todo = (props) => {
  return (
    <div>
      <input value={props.value} />
      {(props.isDeletable) ? <button>-</button> : <button disabled>-</button> }
    </div>
  );
};
```

React - JSX

- Listes en JSX

```
import React from 'react';

export const TodoList = (props) => {
  const listItems = props.todos.map((val, i) =>
    <div key={i}>{val}</div>
  );
  return <div>{listItems}</div>;
};
```

- L’attribut key est obligatoire
  Il permet à React de savoir si cet élément de la liste doit être ou non rafraîchi.
  Idéalement une clé id d’un Enregistrement ou Document de base de données.

- Bonne pratique sinon : générer un id avec uuid par exemple.

React - Stateful components

- Autant les composants les plus simple peuvent être de simple fonction comme vu précédemment, autant la plupart du temps il faudra créer des fonctions constructeurs JS (class en ES6).
- Ces composants auront la possibilité d'entrer en interaction avec d'autres fonctions et leur « state ».
- A minima, écrire une classe qui hérite de `React.Component` et qui implémentent une méthode `render` retournant un sous-arbre JSX.

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
  render() {
    return <h1 className="my-app">Hello</h1>
  }
}

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

React - Stateful components

- Sur plusieurs lignes :

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
  render() {
    return (
      <div>
        <h1 className="my-app">Hello</h1>
        <p>World</p>
      </div>
    );
  }
}

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

- Des parenthèses sont obligatoires si le JSX ne commence pas sur la même ligne que le return.
- Un composant doit avoir un seul élément racine (ici), depuis React 16 on peut retourner un tableau et depuis React 16.2 on peut retourner un Fragment (voir doc)

React - Propriétés

- Les propriétés ou props, permettent de passer des valeurs au moment du rendu du composant (syntaxe proche d'un attribut HTML)
- Pour accéder à une propriété depuis le composant on passer par sa propriétés props (ici en JSX {this.props.content})

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
  render() {
    return <h1 className="my-app">{this.props.content}</h1>
  }
}

ReactDOM.render(
  <App content="Hello props"/>,
  document.getElementById('root'),
);
```

React - Propriétés

- Définir la typologie et la validation des propriétés du composants
- Installer prop-types (voir aussi airbnb-prop-types)
  `npm install prop-types`

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

class Contact extends React.Component {
  render() {
    return <p>
      Hello my name is {this.props.name},
      I'm {this.props.age}
      </p>;
  }
}

Contact.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number,
};
```

React - Propriétés

- Validateurs personnalisés :

```
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 higher than 0 and lower than 120`)
    }
  },
};
```

- Autres validateurs possibles :
  https://github.com/facebook/prop-types

- Valeurs par défaut :

```
Contact.defaultProps = {
  name: 'John'
};
```

React - Refs

- Référencer des éléments du DOM avec refs

```
class ContactAdd extends React.Component {
  add(e) {
    e.preventDefault();
    console.log(this.refs.prenom.value);
    console.log(this.refs.nom.value);
  }
  render() {
    return <form onSubmit={this.add.bind(this)}>
      <div>
        Prénom : <input ref="prenom" />
      </div>
      <div>
        Nom : <input ref="nom" />
      </div>
      <button>+</button>
    </form>;
  }
}
```

React - State

- Props permet de communiquer avec le composant, state est son état interne, la méthode render est appelée à chaque modification
- On ne peut pas modifier le state directement, il faut utiliser la méthode setState

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

React - State

- Il n’est pas nécessaire de modifier tout le state à chaque appel de setState
- Pour des questions de performance, les objets et tableaux du state seront de préférence immuables

```
export class Todo extends Component {
  constructor(props) {
    super(props);
    this.state = {
      saisie: '',
      liste: []
    };
  }

  inputHandler(e) {
    this.setState({
      saisie: e.target.value,
    })
  }

  formHandler(e) {
    this.setState((prevState) => ({
      liste: [...prevState.liste, this.state.saisie]
    }))
  }

  // ...
}
```

React - Imbrication de composants

```
class App extends React.Component {
  constructor() {
    super();
    this.state = { count: 0 };
  }
  increment() {
    this.setState({ count: this.state.count + 1 });
  }
  decrement() {
    this.setState({ count: this.state.count - 1 });
  }
  render() {
    return <div className="App">
      <h1>{this.state.count}</h1>
      <CounterButton update={this.increment.bind(this)}>+</CounterButton>
      <CounterButton update={this.decrement.bind(this)}>-</CounterButton>
    </div>;
  }
}

class CounterButton extends React.Component {
  render() {
    return <button onClick={this.props.update}>
      {this.props.children}
      </button>;
  }
}
```

- Lorsque qu'un state doit être accessible par plusieurs composant, il faut le définir sur le plus proche ancêtre commun, les composants imbriqués y accéder au travers de props

React - Formulaires

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

export class ContactAdd extends Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(e) {
    this.setState({
      [e.target.name]: e.target.value
    });
  }

  handleSubmit(e) {
    e.preventDefault();
    // fetch()...
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <div className="form-group">
          <label>Prénom</label>
          <input type="text" className="form-control" name="firstName" onChange={this.handleChange} />
        </div>
        <div className="form-group">
          <label>Name</label>
          <input type="text" className="form-control" name="lastName" onChange={this.handleChange} />
        </div>
        <button type="submit" className="btn btn-default">Add</button>
      </form>
    );
  }
}
```

React - Cycle de vie

- Chaque composant à un certain nombre de méthodes liées à son cycle de vies :
- Chargement
  - constructor()
  - ~~componentWillMount~~() (dépréciée)
  - render()
  - componentDidMount()
- Destruction
  - componentWillUnmount()
- Mise à jour
  - ~~componentWillReceiveProps~~() (dépréciée)
  - shouldComponentUpdate()
  - ~~componentWillUpdate~~() (dépréciée)
  - render()
  - componentDidUpdate()
- https://reactjs.org/blog/2018/03/27/update-on-async-rendering.html

React - Cycle de vie

- constructor()
  - Appelée côté client et serveur
  - constructor sert à initialiser state et props

React - Cycle de vie

- componentDidMount()
  - Le rendu initial du composant a été effectué
  - Il est possible de manipuler le DOM
  - N’existe que côté client
  - Le bon endroit pour charger un plugin jQuery ou tout ce qui ne s’exécute qu’une seul fois et qui a besoin d’accéder au DOM
  - Démarrer des requêtes AJAX, des timers

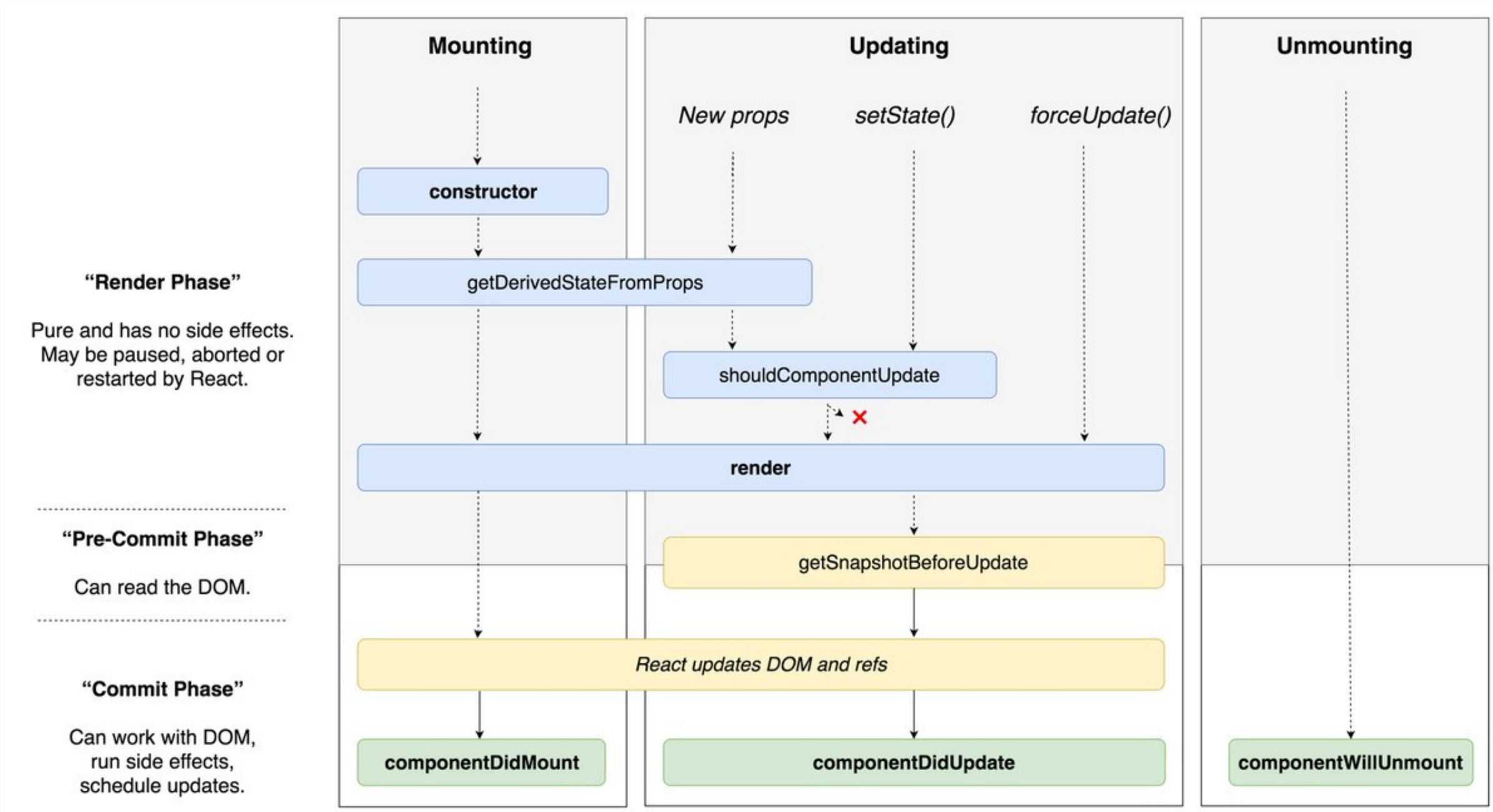
React - Cycle de vie

- shouldComponentUpdate()
  - Permet d'empêcher un render(), doit répondre true ou false
  - Utile pour optimiser une application, ne pas faire de rendu si les props ou le state on été modifié d’une manière qui ne nécessite pas un nouveau rendu (voir aussi PureComponent)

React - Cycle de vie

- componentDidUpdate()
  - Juste après un rendu autre que initial
  - On a accès au DOM
  - Le bon endroit pour un update d’un plugin jQuery (Chosen, Select2…)
- componentWillUnmount()
  - Le composant va être supprimer
  - Permet de supprimer des listeners, libérer la mémoire, appeler clearInterval/ Timeout, sinon l'objet associé au composant ne sera jamais détruit (si ref interne dans un callback)

React - Cycle de vie



React - Higher Order Components

- Un Higher Order Component (HOC) est une fonction qui reçoit un composant en entrée et retourne un nouveau composant (une liste filtrée, remplie, etc…)
- Exemple, connect dans react-redux, qui injecte la fonction dispatch à un composant

```
export default connect()(TodoApp);
```

- Voir Recompose (bibliothèque d'utilitaire HOC)
  https://github.com/acdlite/recompose/

React - Higher Order Components

- Ajouter de nouvelles props via un HOC

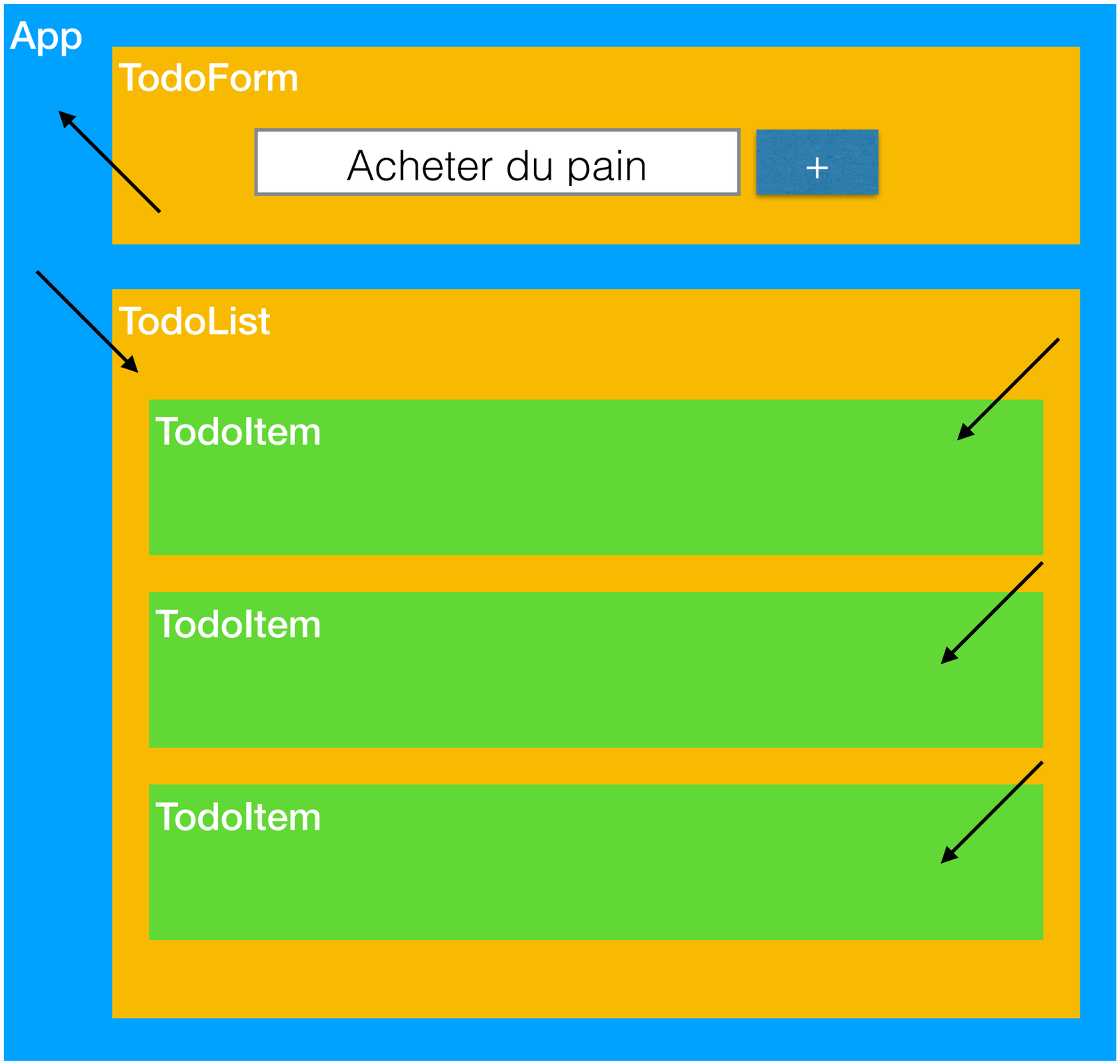
```
render() {
  // Filter out extra props that are specific to this HOC and shouldn't be
  // passed through
  const { extraProp, ...passThroughProps } = this.props;

  // Inject props into the wrapped component. These are usually state values or
  // instance methods.
  const injectedProp = someStateOrInstanceMethod;

  // Pass props to wrapped component
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}
```

React - Higher Order Components

- Renommer le composant retourné (bonne pratique)

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

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}

export const logLifeCycle = (WrappedComponent) => {

  class LogLifeCycle extends Component {
   // ...
  }

  LogLifeCycle.displayName = `LogLifeCycle(${getDisplayName(WrappedComponent)})`;


  return LogLifeCycle;
};
```

React - Exercice



- Créer 3 composants :
  - TodoForm
  - TodoList
  - TodoItem
- Dans le state de App créer une liste de todos (string[])
- Passer ce tableau à TodoList, qui créera autant de TodoItem qu'il y a d'élément dans le tableau
- TodoItem reçoit un élément et l'affiche
- TodoForm Controlled Component, reçoit un callback de App et lui passe la valeur saisie au submit du form
- Le callback de App ajouter l'élément au tableau

formation.tech

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 préfèreront 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
```

Immuabilité - Tableaux

- 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
```

Immuabilité - Tableaux

- 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
```

Immuabilité - Tableaux

- 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
```

Immuabilité - Tableaux

- 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
```

Immuabilité - Objet

- 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
```

Immuabilité - Objet

- 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
```

Immuabilité - Objet

- 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

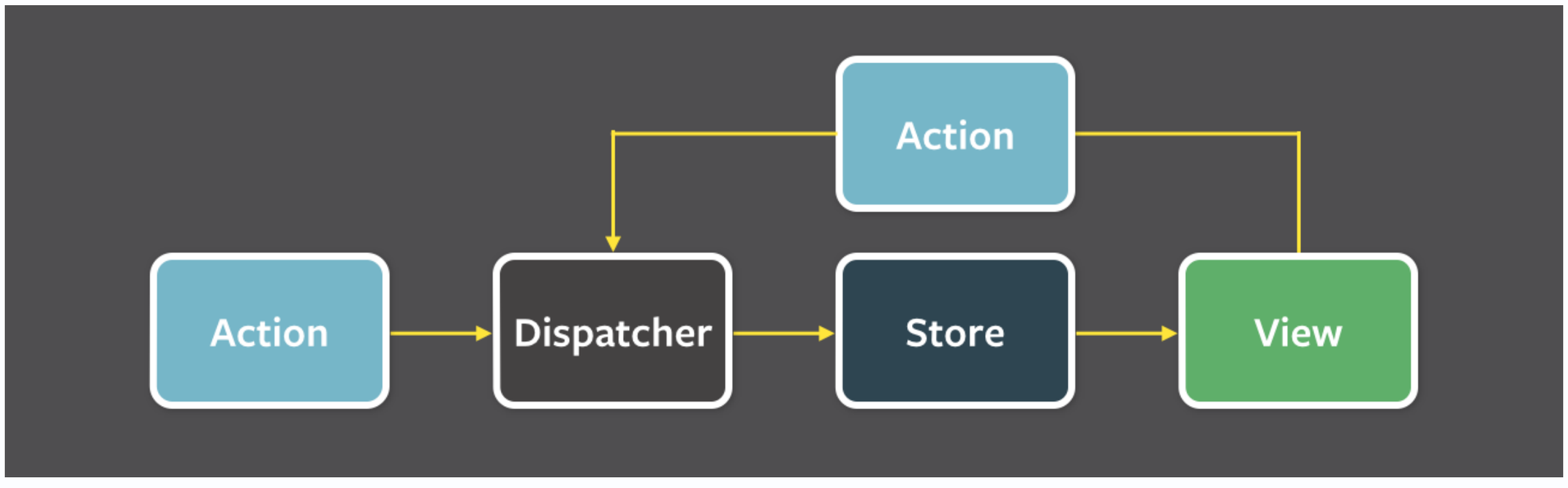
```
const immutable = require('immutable');

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
```

formation.tech

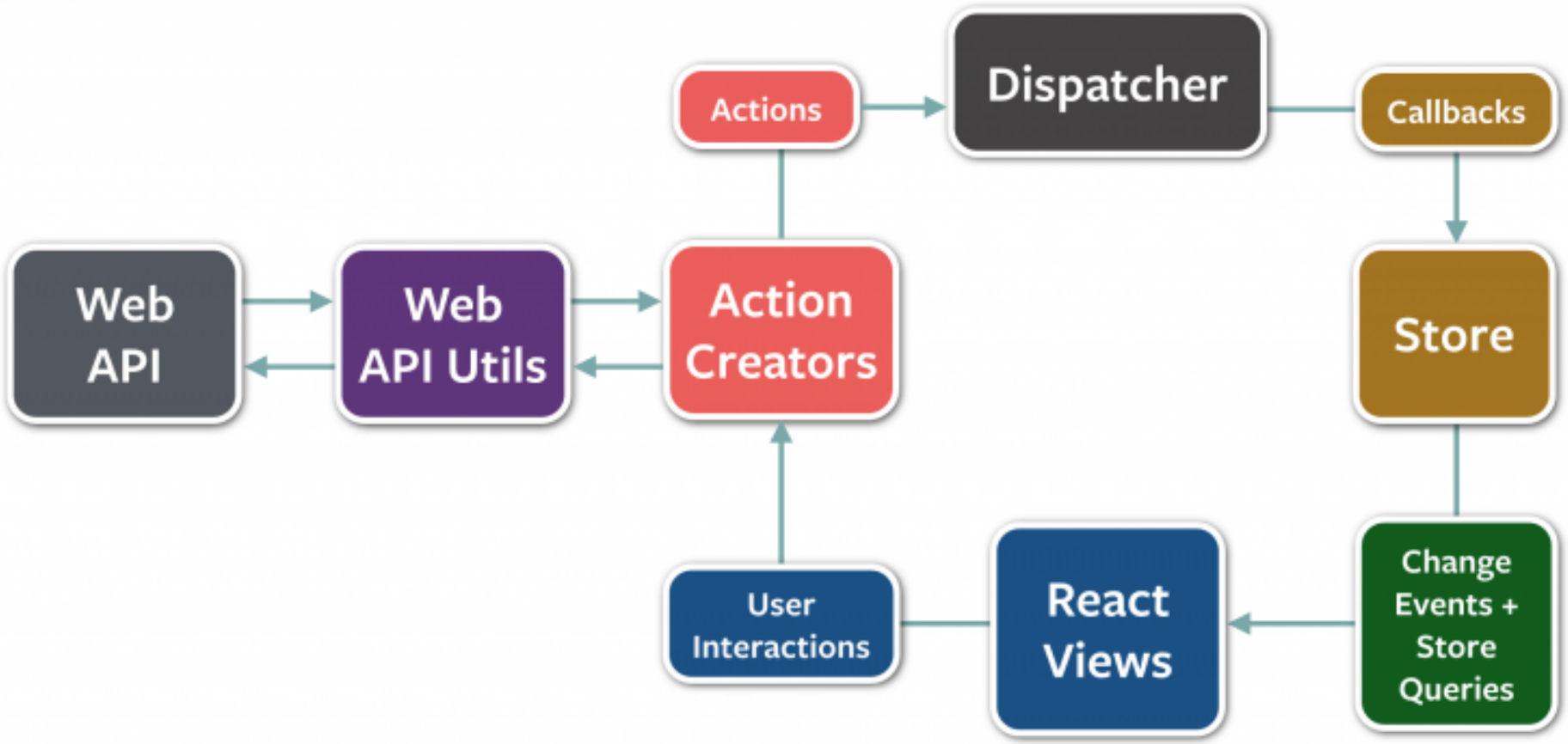
Redux

Redux - Introduction

- Redux est un conteneur d'état (state container)
  Une bibliothèque dont le rôle et de stocker l'état de l'application et ainsi de la réaffirmer dans l'état précédent lorsque l'historique est manipulé.
- Inspiré par Flux (Facebook)
  Redux est inspiré de Flux, une architecture proposée par Facebook pour les applications front-end. Différente bibliothèque propose de simplifier leur mise en place, dont Redux, même si quelques concepts sont différents.



Redux - Introduction



Redux - Quand utiliser Redux ?

- Extrait de la doc du Redux : https://redux.js.org/docs/faq/OrganizingState.html#organizing-state-only-redux-state
  - La même donnée est-elle utilisée pour piloter différents composants ?
  - Avez vous besoin de créer de nouvelles données dérivées de ces données ?
  - Y-a-t'il une valeur que vous souhaiteriez restaurer dans un état précédent (time travel debugging, undo/redo) ?
  - Voulez-vous mettre cette donnée en cache ?
- Si oui à l'une de ces question : Redux
- Voir aussi MobX

Redux - Installation

- Redux
  - `npm install redux`
  - `yarn add redux`
- Intégration avec react
  - `npm install react-redux`
  - `yarn add react-redux`
- Intégration avec l’extension Redux DevTools
  - `npm install redux-devtools-extension --save-dev`
  - `yarn add redux-devtools-extension --dev`

Redux - Quelques principes

- Immutable State Tree
  - Contrairement à Flux, Redux maintient l'ensemble de l'état de l'application dans un arbre unique.
  - Cet arbre stocké sous la forme d'un plain object JavaScript ou bien tout autre structure (voir Immutable.js).
  - Il doit être immuable, une modification doit entrainement la création d'un nouvel objet en mémoire et non la modification de l'objet existant, ceci pour permettre des fonctionnalités plus avancées comme un undo/redo.
- Pas de modification directe du State Tree
  - Il ne faut pas modifier directement le State Tree, au lieu de ça on va « dispatcher » des actions pour indiquer les modifications à apporter à l'arbre.
  - Une action est un objet JS qui décrit le changement à apporter au State Tree.

Redux - Quelques principes

- Actions
  - Une action doit avoir à minima une propriété nommée type.
    Exemple pour un compteur :

```
const incrementAction = {
  type: 'INCREMENT',
};

const decrementAction = {
  type: 'DECREMENT',
};
```

  - Chaque action doit décrire le minimum du changement à effectuer dans l'application Redux.

```
const addTodoAction = {
  id: 123,
  value: 'Apprendre à utiliser Redux',
  type: 'ADD_TODO',
};
```

  - Chaque changement intervenant dans l'application, clic utilisateur, nouvelle données reçues du serveur, texte saisi… devrait être décrit par une action la plus simple possible.

Redux - Quelques principes

- Actions creators
  - Pour faciliter la création d'actions et les pouvoir les réutiliser plus facilement à différents endroit de l'application, on utilise des fonctions appelées actions creators

```
export const setVisibilityFilter = (filter) => ({
  type: 'SET_VISIBILITY_FILTER',
  filter
});
```

Redux - Quelques principes

- Fonction pure
  - Retourne toujours la même valeur lorsque appelée avec les des paramètres identiques.
  - Aucun effet parallèle comme l'écriture dans un fichier
  - Ne modifie par ses paramètres d'origines (objets, tableaux…)

```
// fonction pure
const addition = function(a, b) {
  return Number(a) + Number(b);
};

// fonctions impures
const getRandomIntInclusive = function(min, max) {
  return Math.floor(Math.random() * (max - min + 1)) + min;
};

const validateUser = function(user) {
  localStorage.setItem('user', user);
  return user === 'Romain';
};

const userToUpperCase = function(user) {
  user.prenom = user.prenom.toUpperCase();
  return user;
};
```

Redux - Quelques principes

- Reducers
  - Fonction pure
  - Reçoit l'état précédent et l'action dispatchée
  - Retourne l'état suivant
  - Peut conserver les références vers les objets non-concernés par l'action

```
var counterReducer = function(state, action) {
  if (state === undefined) {
    return 0;
  }

  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
  }

  return state;
};
```

Redux - Quelques principes

- Reducers en ES6
  - Privilégier des constantes plutôt que des chaînes de caractères pour les types d’actions
  - Utiliser une valeur par défaut pour l’état initial
  - Ne pas utiliser Symbol() si le state doit être sérialisé.

```
const Counter = {
  INCREMENT: Symbol(),
  DECREMENT: Symbol(),
};

const counterReducer = (state = 0, action) => {
  switch (action.type) {
    case Counter.INCREMENT:
      return state + 1;
    case Counter.DECREMENT:
      return state - 1;
  }

  return state;
};
```

Redux - Mise en place

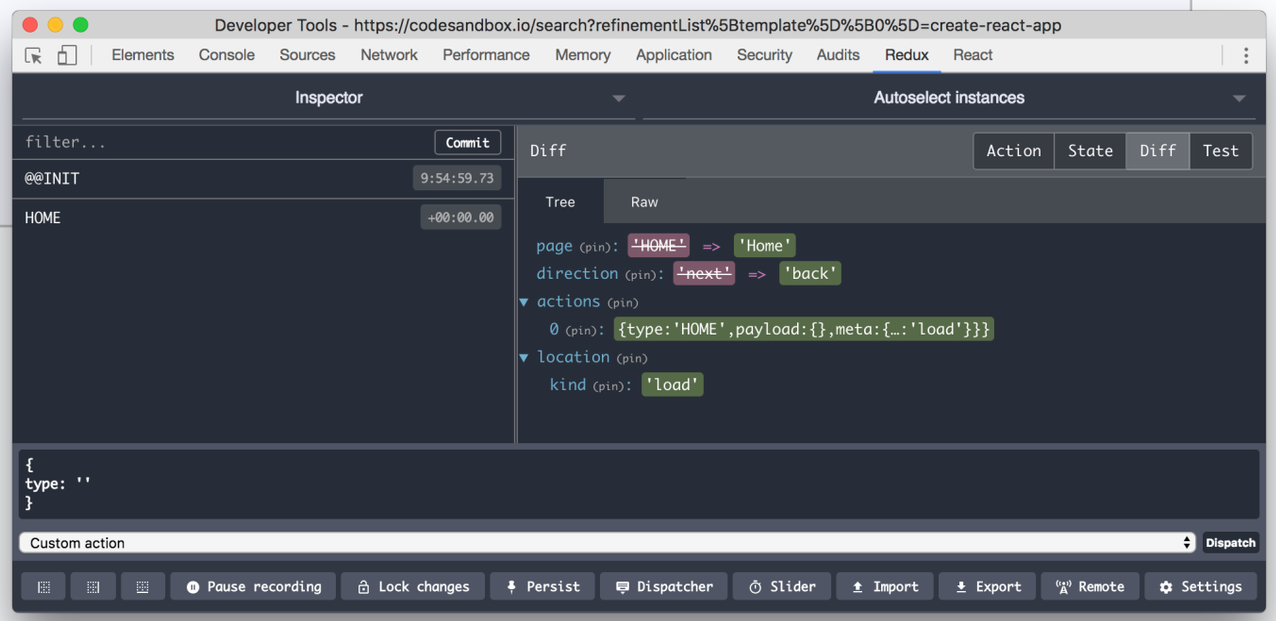
- Store
  - Objet dans lequel est stocké le state (store.getState())
  - Doit être créé à partir d'un reducer ou d'une combinaison de reducers
  - Peut recevoir un state initial, qui pourrait être persisté dans le localStorage par exemple
  - Peut également recevoir des plugins appelées middlewares

```
import { createStore } from 'redux';

const counterReducter = (state, action) => {
  // ...
};

const store = createStore(counterReducter);
```

Redux - Mise en place

- Redux DevTools
  - Installer *redux-devtools-extension*
  - Passer *composeWithDevTools* en 2e paramètres de *createStore*

```
import { createStore, combineReducers } from 'redux';
import { composeWithDevTools } from 'redux-devtools-extension';
import { counter } from './reducers/counter';

export const store = createStore(
  counter,
  composeWithDevTools()
);
```



Redux - Mise en place

- React Redux
  - Pour que le store soit disponible dans l'application React, on utilise le composant Provider de react-redux (il doit être à la racine)

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/App';
import { store } from './store';
import { Provider } from 'react-redux';

ReactDOM.render(
  <Provider store={store}><App /></Provider>,
  document.getElementById('root'),
);
```

Redux - Mise en place

- La fonction connect de react-redux permet d'associer de rendre disponible la méthode dispatch de redux et d'accéder à certaines propriétés du state

```
import React, { Component } from 'react';
import { connect } from 'react-redux'
import { counterIncrement } from '../actions/counter';

export class ButtonCounter extends Component {
  constructor() {
    super();
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.props.dispatch(counterIncrement());
  }
  render() {
    return <button onClick={this.handleClick}>{this.props.count}</button>;
  }
}

const mapStateToProps = (state) => ({
  count: state.count,
});

export const ButtonCounterContainer = connect(mapStateToProps)(ButtonCounter);
```

Redux - Mise en place

- On peut également associer directement des méthodes avec *mapDispatchToProps*

```
import React from 'react';
import { connect } from 'react-redux'
import { counterIncrement } from '../actions/counter';

export const ButtonCounter = (props) => {
  return <button onClick={props.handleClick}>{props.count}</button>;
};

const mapStateToProps = (state) => ({
  count: state.count,
});

const mapDispatchToProps = (dispatch) => ({
  handleClick: () => dispatch(counterIncrement()),
});

export const ButtonCounterContainer = connect(
  mapStateToProps,
  mapDispatchToProps,
)(ButtonCounter);
```

Redux - Code asynchrone

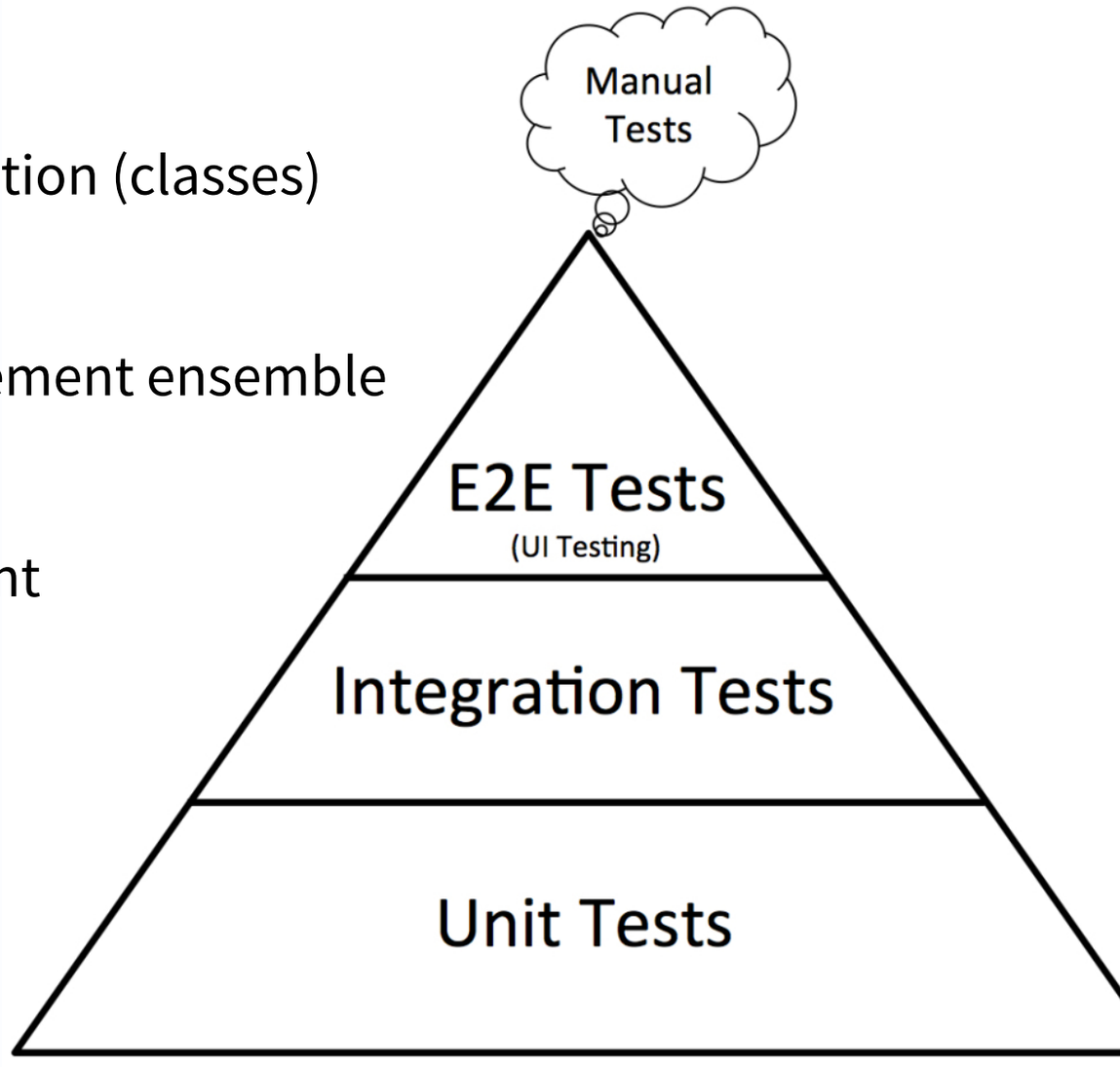
- Redux Thunk
  https://stackoverflow.com/questions/35411423/how-to-dispatch-a-redux-action-with-a-timeout/35415559#35415559
- Redux Saga
  https://stackoverflow.com/questions/35411423/how-to-dispatch-a-redux-action-with-a-timeout/35415559#38574266

formation.tech

Jest

Tests Automatisés - Introduction

- Avec les tests automatisés, les scénarios de test sont codés et peuvent être rejoués régulièrement.
- 4 types de test :
  - Test unitaire
    Permets de tester les briques d'une application (classes)
  - Test d'intégration
    Teste que les briques fonctionnent correctement ensemble
  - Test fonctionnel
    Vérifie l'application du point de vue du client
  - Test End-to-End (E2E)
    Vérifie l'application dans le client



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

Jest - Installation

- Installation

```
npm install --save-dev jest
yarn add --dev jest
```

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 !');
});
```

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.
```

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.
```

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 |                |
----------|----------|----------|----------|----------|----------------|
```

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);
});
```

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');
});
```

Jest - React

- Une application créé 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;
```

Jest - React

- 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

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();
});
```

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/>,
  ]);
});
```

Jest - Enzyme

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

```
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);
});
```

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>
    );
  }
}
```

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');
});
```