

Formation ReactJS : Développement Web



par **Guillaume Schoder**

- Développeur Fullstack React.js/Node.js depuis 7 ans
- Donne des formations depuis 4 ans
- Travaille sur Gros projets et Pour des startups

Tour de table

- Parcours / métier
- Connaissances
- Attentes

J1

- Introduction à React
- Les nouveautés ECMAScript
- Premiers développements avec ReactJS

J2

- Les Composants React
- Styliser ces composants
- Interaction de l'utilisateur avec les composants
- Les possibilités de build

J3

- Gestion centralisée des données
- Application monopage avec ReactJS et un module de store de variables
- Le Router
- Application isomorphe

Savoir utiliser la bibliothèque React.JS pour faciliter la création d'applications web isomorphiques accessibles et performantes.

- Connaître les apports de la bibliothèque React dans le cadre d'un développement JavaScript
- Savoir coupler React.JS avec des modules complémentaires, JSX et ES6
- Concevoir une application web mono-page avec la bibliothèque React et l'architecture Flux
- Comprendre et maîtriser la notion d'immutabilité pour optimiser les performances des applications mises à jour uniquement lors de changement
- Comprendre et mesurer les impacts du choix d'une architecture incluant ce type d'application

Philosophie de React

Le DOM-Virtuel

Approche orientée composant

Cycle de vie

JSX

Le DOM-Virtuel

React est une librairie JS maintenant un **DOM virtuel** ou VDOM (composé d'un ou plusieurs éléments HTML/JSX e.g. **composant**)

DOM 'réel' initial

```
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
  </head>
  <body>
    ...
    <div id="root"></div>
  </body>
</html>
```



Code React

```
ReactDOM.render(
  <h1>Hello World</h1>,
  document.getElementById('root')
);
```



React

Performant > En mémoire navigateur
Compare les rendus DOM pour re-rendre



DOM 'réel' final

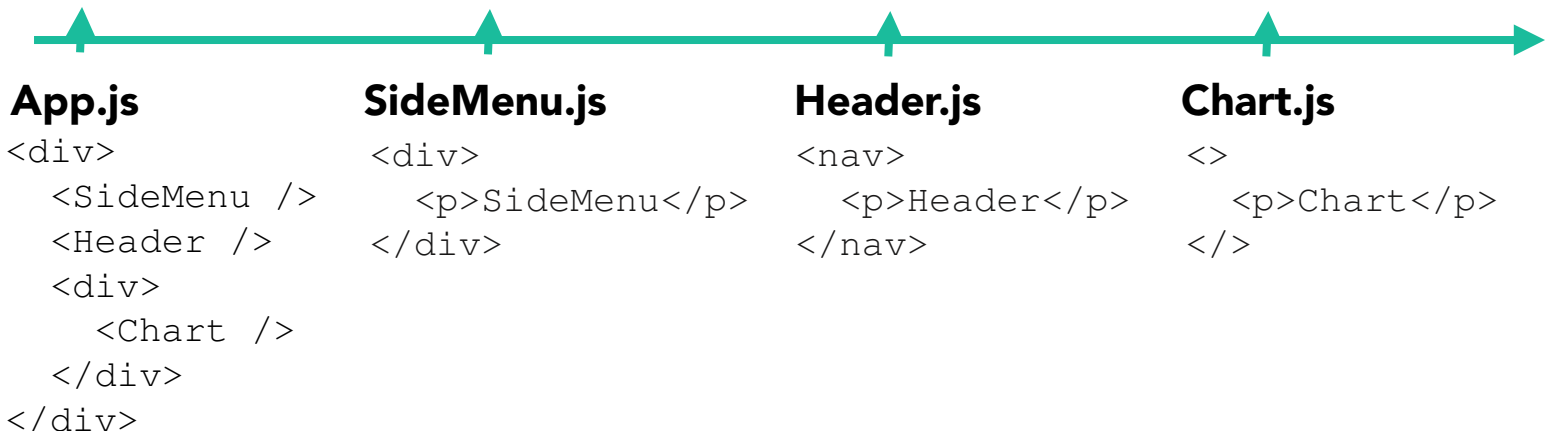
```
<!DOCTYPE html>
<html lang="en">
  <head>
    ...
  </head>
  <body>
    ...
    <h1>Hello World</h1>
  </body>
</html>
```

Approche orientée composant (1)

En React, TOUT est composant : un élément parent du **DOM virtuel** (ex : `<App />`), contenant d'autres éléments du DOM virtuel (**composants**) qui seront **rendus récursivement (top down)**

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

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

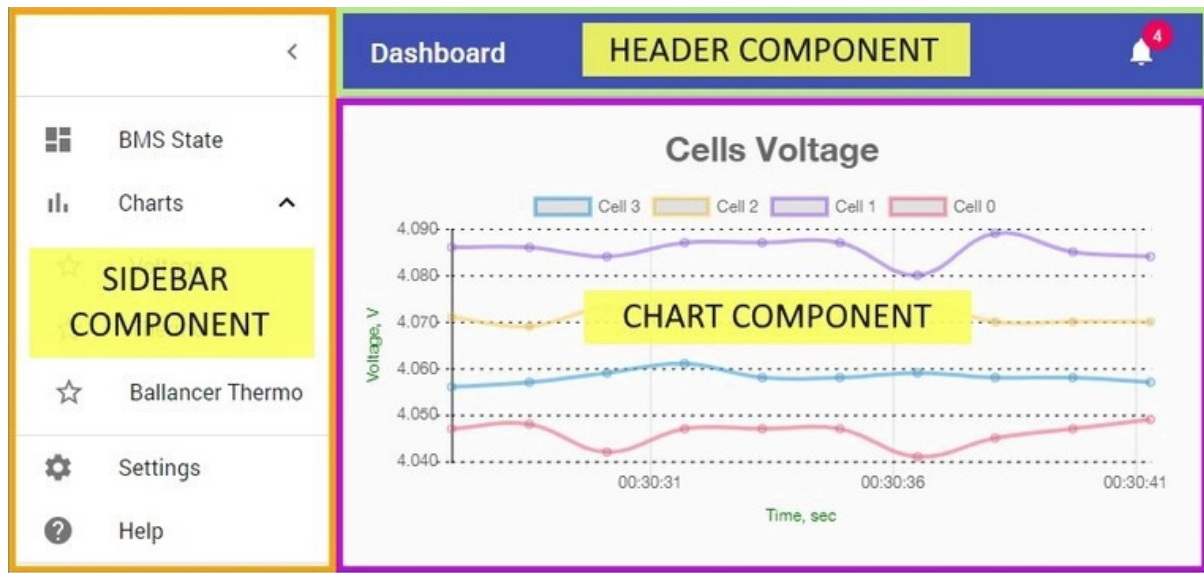


DOM final de App.js

```
<div>
  <div>
    <p>SideMenu</p>
  </div>
  <nav>
    <p>Header</p>
  </nav>
  <div>
    <p>Chart</p>
  </div>
</div>
```


Approche orientée composant (2)

- App React.js : composants imbriqués les uns dans les autres. (Tout élément d'une page est un composant)
- La page est un composant parent constitué de composants enfants.
- Un composant est un fichier Js qui peut être une classe ou une fonction particulière



Cycle de vie

Initialization

setup props and state

Mounting

componentWillMount

render

componentDidMount

Updation

props

componentWillReceiveProps

shouldComponentUpdate

componentWillUpdate

render

componentDidUpdate

states

shouldComponentUpdate

true

false

componentWillUpdate

render

componentDidUpdate

Unmounting

componentWillUnmount

- Le DOM virtuel (et donc le réel) est géré entièrement par React, pas par HTML5 (preventDefault())
- React s'appuie sur des cycles de rendu des composants (fonction render()) et compare le VDOM avant et après chaque cycle de rendu
- Les cycles de rendu sont asynchrones.

JSX

Le JSX est une syntaxe extension du JavaScript qui permet de générer des élément React.js.

Le JSX permet d'intégrer la logique JavaScript et l'UI dans une même expression / fonction / fichier .js.

```
const element = <h1>Je suis cool</h1>;  
function getGender() { return 'M.'; }  
const name = 'José Cool';  
const element = <h1>Je suis {getGender()} {name}</h1>;  
ReactDOM.render(element, document.getElementById('root'))
```

Des fonctions React de création de composant peuvent également générer des éléments.

```
const element = (  
  <h1 className="title">  
    Hello  
  </h1>;  
)  
=   
const element = React.createElement(  
  'h1',  
  { className: 'title' },  
  'Hello'  
)
```

Les autres frameworks Javascript



Angular.Js



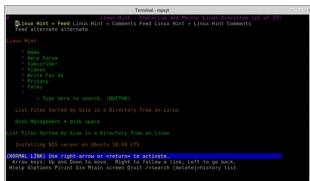
Vue.Js



Environnement de développement et outils de debug



Node.js



Terminal



**Navigateur avec potentiels
plugins de debug**

Présentation

ECMAScript est un ensemble de normes concernant les langages de programmation de type script et standardisées par Ecma International dans le cadre de la spécification ECMA-262. Il s'agit donc d'un standard, dont les spécifications sont mises en œuvre dans différents langages de script, comme JavaScript ou ActionScript. C'est un langage de programmation orienté prototype.

Les nouveautés ES6/ES2015

```
function add(a, b){  
  var x = a || 2;  
  var y = b || 2;  
  
  return x+y;  
}
```

```
function add(a=1, b=1){  
  return x+y;  
}
```

```
(a=1, b=1) => x+y;
```

```
var str = "Lorem ipsum dolor sit amet\n"  
+ "consectetur adipiscing elit,\n"  
+ "In dictum, quam eget tristique facilisis."
```

```
var str = `Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
In dictum, quam eget tristique facilisis.`
```

```
var str = "Hi ${firstname} ${lastname}, Welcome to WayToLearnX!";
```

```
Class Person {}    ...  Class Employee extends Person {}
```

ES2017 : les fonctions « async »

Les fonctions Async permettent de faire des requêtes asynchrones basées sur des promesses d'être écrites de façon plus claire et d'éviter un chinage de promesse.

Avant

```
function test() {  
  createPerson().then(() =>  
    createOrder().catch((error) =>  
      Throw new Error(error)  
    )  
  ).catch((error) =>  
    Throw new Error(error)  
  )  
}
```

Après

```
async function test() {  
  try {  
    await createPerson()  
    await createOrder()  
  }  
  catch(error) {  
    Throw new Error(error)  
  }  
}
```

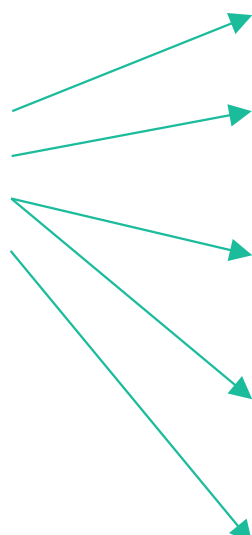

Le composant (1)

Un composant peut être composé :

- de propriétés externes (optionnel) : **props**
- d'états internes (optionnel) : **states**
- d'un cycle de vie (optionnel) : **fonctions**
- d'une fonction de rendu : **JSX**

La fonction render est obligatoire !!

Un composant renvoie toujours du JSX



```
class YourComponentName extends Component {  
  constructor(props) {  
    super(props)  
    this.state = {  
      state1: new Date()  
    }  
  }  
  
  componentDidMount() {  
  }  
  
  componentWillUnmount() {  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>{this.state.state1}</h1>  
      </div>  
    )  
  }  
}
```

Le composant (2)

Depuis React v16 les composants utilisent les 'hooks' pour gérer leurs états et cycle de vie.

```
function YourComponentName(props) {  
  const [state1, setState1] = useState(new Date())  
  
  useEffect(() => {  
    return () => {}  
  }, [])  
  
  return (  
    <div>  
      <h1>{this.state.state1}</h1>  
    </div>  
  )  
}
```

```
class YourComponentName extends Component {  
  constructor(props) {  
    super(props)  
    this.state = {  
      state1: new Date()  
    }  
  }  
  
  componentDidMount() {  
  }  
  
  componentWillUnmount() {  
  }  
  
  render() {  
    return (  
      <div>  
        <h1>{this.state.state1}</h1>  
      </div>  
    )  
  }  
}
```

Props et States

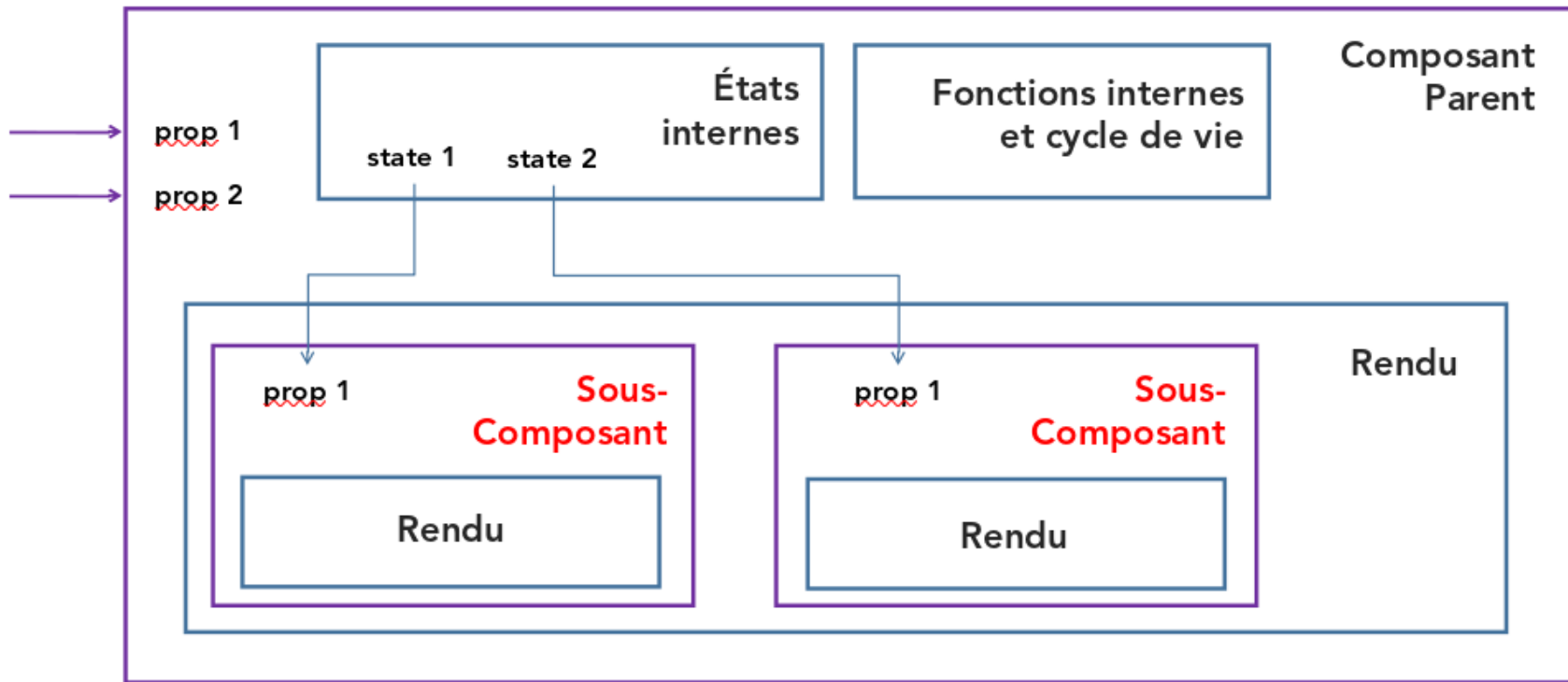
Props

- Paramètre reçus par un composant depuis son parent
- Non modifiable
- Peut contenir une fonction callback du parent

States

- état interne du composant, immutable
- Une modification déclenche un cycle de render
- Pas supprimé en cas de re-render
- Modifiable via `setState`, `useState`
- Peut être initialisé à partir d'une prop
- Ralenti les performances de l'application

Un composant d'un point de vue fonctionnel



Différents type de composant

Composant Pur

- Prédiction
- Affichage
- pas de modification hors du scope

```
const MyList = props =>
  <ul>
    {props.items.map(item => (
      <li key={item.name}>
        {item.name}
      </li>
    ))}
  </ul>
```

Composant Container

- Récupère les données
- Gère le cycle de vie des données

```
const MyListWithData = props => {
  const [items, setItems] = useState([])

  useEffect(() => setItems(getItems()), [])

  return (<MyList items={items}/>)
}
```

Les 'hooks' des composants fonctionnels

- useState : gère le store et la mise à jour d'une variable
- useEffect : hook de render, permet d'effectuer une fonction générale en fonction de la mise à jour d'autres variables
- useMemo : gère la mise à jour d'une variable en fonction d'autres dépendances (performance)
- useCallback : gère la mise à jour d'une fonction en fonction d'autres dépendances (performances)

```
function Counter({ step: 2 }){  
  const [var1, setVar1] = useState(0)  
  const value = useMemo(() => var1 + 1, [var1])  
  const incrementVar1 = useCallback((value) => setVar1((val) => val + value), [])  
  
  useEffect(() => { incrementVar1(step) }, [step])  
  
  return (<p>{value}</p>)  
}
```

Gestion des événements click, mouseover...

React utilise son propre système de SyntheticEvent qui est appliqué sur chaque balise HTML de code

La syntaxe est basée sur le 'on'[monEvent] : exemple : onClick / onMouseOver ... (camel Case)

Il est possible d'utiliser ou non les événements HTML par défaut et d'empêcher le spread d'événement

HTML

```
<a
  href="#"
  onclick="
    console.log('le lien est cliqué');
    return false"
>
  LE LIEN!
</a>
```

React

```
function leLien() {
  function handleClick(e) {
    // e est un événement synthétique (w3C specs)
    e.preventDefault();
    console.log('le lien est cliqué');
  }

  return (
    <a href="#" onClick={handleClick}>LE LIEN!</a>
  )
}
```

Les événements supportés

React.js normalise les événements afin que ceux-ci puissent être interprétés pour chaque navigateur et par le DOM virtuel.

Les événements synthétiques supportés :

- **Clipboard** : onCopy / onCut / onPaste
- **Composition** : onCompositionEnd|Start|Update
- **Keyboard** : onKeyDown|Press|Up
- **Focus** : onFocus / onBlur
- **Form** : onChange / onInput / onInvalid / onReset / onSubmit
- **Generic** : onError / onLoad
- **Mouse** : onClick / onContextMenu / onDoubleClick / onDrag / onDragEnd|Enter|Exit|Leave|Over|Start / onDrop / onMouseDown|Enter|Leave|Move|Out|Over|Up
- **Pointer** : onPointerDown|Move|Up|Cancel|Enter|Leave|Over|Out / onGotPointerCapture / onLostPointerCapture
- **Selection** : onSelect
- **Touch Events** : onTouchCancel|End|Move|Start
- **UI** : onScroll
- **Wheel** : onWheel
- **Media** : onAbort / onCanPlay|CanPlayThrough / onDurationChange / onEmptied / onEncrypted / onEnded / onError / onLoadedData|Metadata / onLoadStart / onPause / onPlay / onPlaying / onProgress / onRateChange / onSeeked / onSeeking / onStalled / onSuspend / onTimeUpdate / onVolumeChange / onWaiting
- **Image** : onLoad / onError
- **Animation** : onAnimationStart|End|Iteration
- **Transition** : onTransitionEnd
- **Other** : onToggle

Gestion des événements listeners

Il suffit de définir un "listener" à l'initialisation d'un composant. En utilisant une classe ES2015, le pattern classique est de définir le gestionnaire d'événement comme une méthode de la classe

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = { toggled: true };
    // Binding pour faire fonctionner "this" dans la callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({ toggled: !state.toggled }));
  }

  render() {
    return (
      <CoolButton onClick={this.handleClick}>
        {this.state.toggled ? 'ON' : 'OFF'}
      </CoolButton>
    )
  }
}
```

Le 'binding' en JSX

Si le bind ne vous plaît pas il existe 2 autres façons :

```
class Toggle extends React.Component {  
  
  handleClick = (e) => {  
    console.log(e)  
  }  
  
  render() {  
    return (  
      <button onClick={this.handleClick}>  
        Click Me  
      </button>  
    )  
  }  
}
```

ou

```
class Toggle extends React.Component {  
  handleClick(e) {  
    console.log(e)  
  }  
  
  render() {  
    return (  
      <button onClick={ (e) => this.handleClick(e) }>  
        Click Me  
      </button>  
    )  
  }  
}
```

Les différentes approches pour styliser un composant

Style classique

```
Import './style.css'

render(
  <div className='my-class'>
    Hello World
  </div>
)
```

inlineStyle React

```
import React from 'react'

render(
  <h1 style={{ fontWeight:400 }}>
    Hello World
  </h1>
)
```

Module css

```
import React from 'react'
Import styles from './css.module.css'

render(
  <div style={styles.container}>
    <h1 style={styles.title}>
      Hello World
    </h1>
  </div>
)

// Css.module.css
.container:{
  display:flex;
  flex-wrap: warp;
}
.title{ color: red; }
```

Styled-Component & Emotion

Emotion est une bibliothèque conçue pour écrire des styles css avec JavaScript.

Il y a deux principales façons de l'utiliser :

```
import { css } from '@emotion/react'
```

```
const color = 'white'
```

```
render(  
  <div  
    style={css`  
      padding: 32px;  
      border-radius: 4px;  
      &:hover {  
        color: ${color};  
      }  
    `}  
  >  
    Hover to change color.  
  </div>  
)
```

```
import styled from '@emotion/styled'
```

```
const Button = styled.button`  
  padding: 32px;  
  background-color: hotpink;  
  font-size: 24px;  
  border-radius: 4px;  
  color: black;  
  font-weight: bold;  
  &:hover {  
    color: white;  
  }  
,
```

```
render(  
  <Button>This my button component.</Button>  
)
```

Gestion d'Ajax

Axios, jQuery AJAX, et le window.fetch intégré dans le navigateur, etc

Pour récupérer des données à l'initialisation du composant : `componentDidMount()`

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.state = { error: null, isLoading: false, items: [] }
  }

  componentDidMount() {
    fetch("https://api.exemple.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          this.setState({ isLoading: true, items: result.items })
        },
        // C'est important de gérer les erreurs ici (au lieu du
        catch) pour ne pas engloutir d'autres erreurs du composant
        (error) => { this.setState({ isLoading: true, error }) }
      )
  }
}
```

```
render() {
  const { error, isLoading, items } = this.state

  if (error) {
    return <div>Error: {error.message}</div>;
  } else if (!isLoading) {
    return <div>...loading</div>
  } else {
    return (
      <ul>
        {items.map(item => (
          <li key={item.name}>
            {item.name}
          </li>
        ))}
      </ul>
    )
  }
}
```

Gestion d'Ajax avec des hooks

```
function MyComponent() {  
  const [error, setError] = React.useState(null)  
  const [isLoading, setLoading] = React.useState(false)  
  const [items, setItems] = React.useState([])  
  
  React.useEffect(() => {  
    (async () => {  
      setLoading(true)  
      try {  
        const response = await fetch("https://api.exemple.com/items")  
        const data = await response.json()  
        setItems(data)  
      }  
      catch(error) {  
        setError(error)  
      }  
      setLoading(false)  
    })()  
  
  }, [])
```

```
    if (error) {  
      return (<div>Error: {error.message}</div>);  
    } else if (!isLoading) {  
      return <div>...loading</div>  
    } else {  
      return (  
        <ul>  
          {items.map(item => (  
            <li key={item.name}>  
              {item.name}  
            </li>  
          ))}  
        </ul>  
      )  
    }  
  }  
}
```

Gestion des formulaires (basique)

Les formulaires gardent naturellement des états internes.

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

Ce formulaire amène l'utilisateur sur une nouvelle page quand l'utilisateur soumet le formulaire.

Gestion des formulaires (avec React)

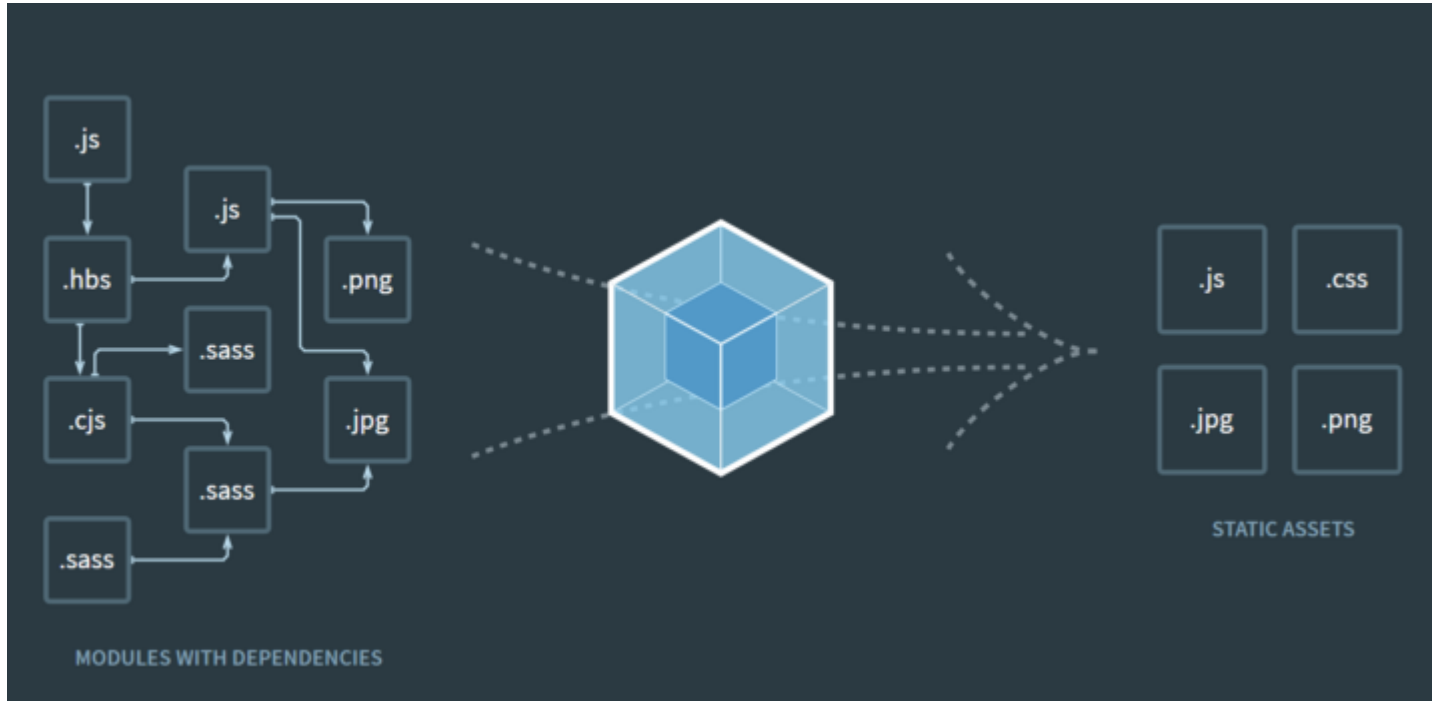
Il est préférable de garder et gérer les valeurs des entrées du formulaire via JavaScript et React.js.

```
class MyForm extends React.Component {
  constructor(props) {
    super(props)
    this.state = { value: '' }
    this.handleChange = this.handleChange.bind(this)
    this.handleSubmit = this.handleSubmit.bind(this)
  }

  handleChange(event) { this.setState({ value: event.target.value }) }

  handleSubmit(event) {
    event.preventDefault();
    axios.post('/test', { body: this.state })
  }

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

Build complet avec webpack

- Webpack est un bundler de modules statiques
- Construit en interne un graphique de dépendance
- Combine chaque module dont votre projet a besoin en un ou plusieurs bundles

```
module.exports = {
  entry: './src/index.tsx',
  mode: production ? 'production' : 'development',
  output: {
    filename: BUNDLE_NAME,
    chunkFilename: CHUNK_NAME,
    path: DIST_PATH,
    publicPath: PUBLIC_PATH,
  },
  plugins,

  // Enable sourcemaps for debugging webpack's output.
  devtool: production ? 'source-map' : 'eval-source-map',

  resolve: {
    extensions: ['.ts', '.tsx', '.js', '.json'],
    aliasFields: ['browser'],
    alias: {
      '#ui': path.resolve(__dirname, 'src/ui/'),
    },
  },
}
```

```
module: {
  rules: [
    {
      exclude: /\.test\.ts$/,
    },
    {
      test: /\.?(?:le|c)ss$/,
      use: [
        { loader: 'style-loader' },
        { loader: 'css-loader' },
        { loader: 'less-loader' },
      ],
    },
    {
      test: /\. (png|jpg|gif|woff|woff2|eot|ttf|otf) $/,
      use: [
        {
          loader: 'file-loader',
          options: {},
        },
      ],
    },
  ],
}
```

Build managé avec la cli create-react-app

- Configuration webpack basique déjà intégrée
- Possibilité de configuration manuelle en ajoutant un fichier webpack.config.js / utiliser autre config dans package.json (react-app-rewired / react-script)

Les différents packages de bootstrapping React

Le bootstrapping permet la mise en place rapide d'une app React avec les dépendances et les fichiers minimum pour démarrer le développement du projet.

```
npx create-react-app my-app --template typescript
```

- Routing : React-Router-Dom
- UI : Material-UI / React-Bootstrap / Tailwind
- Communication : Axios / GraphQL
- (optionnel) Store : Redux

Autre Package / Framework : Next.JS / Gatsby

Environnement et build

- Jouer sur le .env de build avec NODE_ENV
- Feature flagging : exemple : <http://localhost:3000/?filters=true>

Présentation des différentes technologies de gestion (Redux, MobX, context, Recoil etc.)

Le principe de ces libs est de Store des variables de façon globale afin qu'elles puissent être utilisées par tous les composants sur les différentes pages de l'application (**machine d'état**)

Les systèmes sont basés sur le principe de « Pure Function » et de hook

2 types d'approche :

- Afin que les données et les fonctions puissent être accessibles par les composants ceux-ci doivent être wrap dans un Provider
- Utiliser une librairie qui gère tout et qui renvoie ses fonctions d'accessibilité

Présentation de Redux

Les états sont immutables et ils ne peuvent être changés que par des "pure functions"

Attention la machine d'état est initialisée à chaque premier chargement de page, ainsi si l'utilisateur fait un refresh de la page la machine d'état revient à son état initial (Single Page App).

Peut être adapté pour faire de la persistance (sauvegarde d'état sur le storage ou dans app mobile)

Single source of truth

Le Store et les Actions

Le store représente la machine d'état de l'application :

- il est initialisé avec des valeurs
- il détient la vérité à tout moment de la valeur des états

Les actions représentent des fonctions liées à une seule action (d'édition) sur la machine d'état :

- elles sont composées :
 - un type représentant une action sur la machine d'état
 - une payload représentant la donnée de l'état modifié

```
import { EDIT_GAME_HISTORY } from "../actionTypes";

export const editHistory = content => ({
  type: EDIT_GAME_HISTORY,
  payload: {
    gameHistory: content
  }
});
```


Création de Reducteurs pour les Actions

Un REDUCER est une fonction associée à un modèle de donnée (exemple : todoList) représentant la valeur initiale de l'état et l'ensemble des actions possibles (défini par un type et la payload : la donnée à mettre dans la machine d'état).

```
import { EDIT_GAME_HISTORY } from "../actionTypes";

const initialState = {
  gameHistory: [],
};

export default function(state = initialState, action) {
  switch (action.type) {
    case EDIT_GAME_HISTORY: {
      return {
        ...state,
        gameHistory: [...state.gameHistory, action.payload.gameHistory],
      };
    }
    default:
      return state;
  }
}
```

Utilisation avec React

on Wrap l'element

```
import { connect } from "react-redux";
import { editHistory } from "../redux/actions";
```

```
...MyComponent
```

```
const mapStateToProps = state => {
  return state.gameHistory
};
```

```
export default connect(
  mapStateToProps,
  { editHistory }
)(MyComponent);
```

Ou de façon globale :

```
ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </React.StrictMode>
);
```

on utilise des hooks

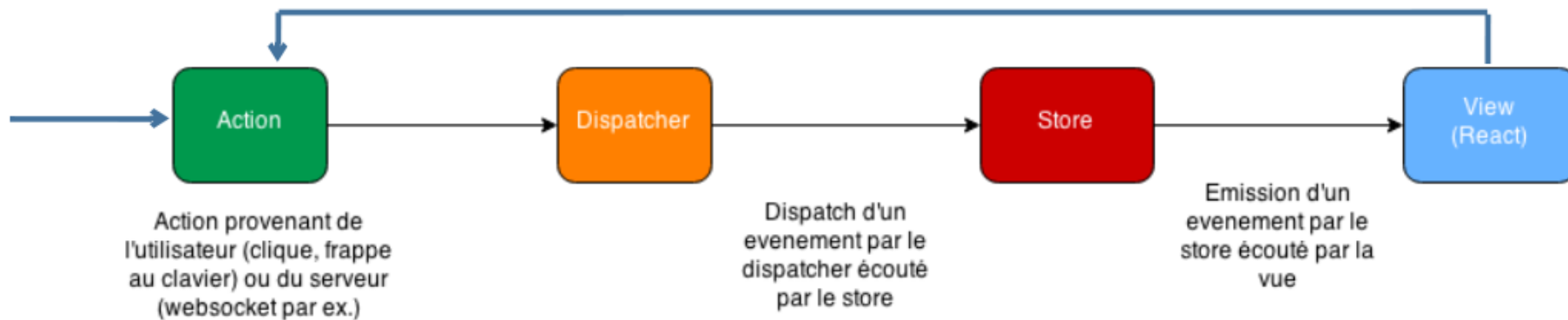
```
import { useDispatch, useSelector } from 'react-redux'
```

```
function MyComponent(props) {
  const dispatch = useDispatch()
  const myReduxState = useSelector(state => state.myReduxState)
  // ou
  Définir une fonction selector en amont ex :
  const myState = useSelector(myfunction)

  return (
    <button
      onClick={() => dispatch({ type: 'increment-counter' })}
    >
      Increment counter : {myReduxState}
    </button>
  )
}
```

Modèle Flux

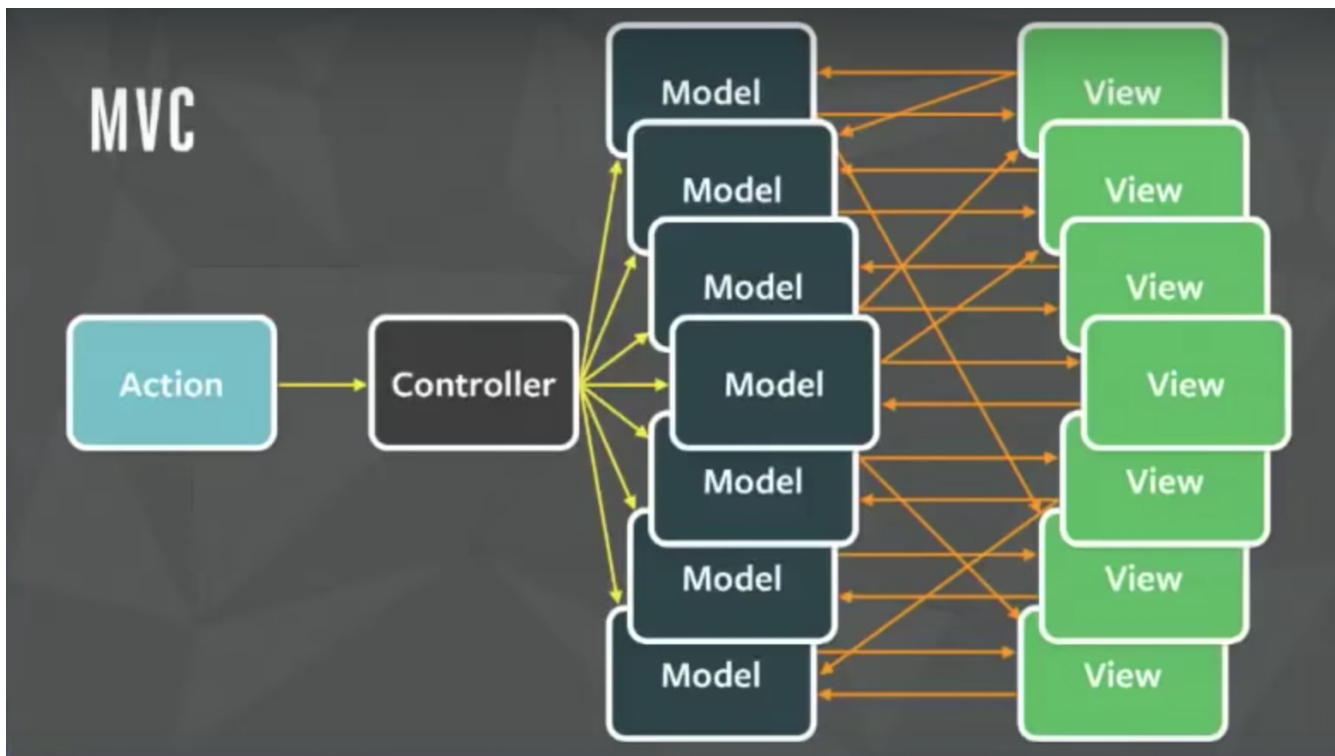
properties flow down, actions flow up



Cycle de vie : render récursif et conditionnel des composants

limitation du MVC : moins de ré-utilisabilité, pas de découpage fonctionnel de l'UI

Modele MVC dans React



Flux/Redux : présentation. Propagation de données.

Redux

- les actions peuvent également être des fonctions et des promesses.
- Un 'store' par application
- Redux 'store' expose des fonctions API simples
- La logique des données est gérée par le 'reducer'.

Flux

- Plusieurs 'store' par application
- Une modification déclenche un cycle de render
- Un seul 'dispatcher' où passent les actions
- La logique des données est contenue dans le store

Création des contrôleurs

On crée un context

```
export const DataContext = createContext({
  data: [],
  fetchData: async () => {},
});
```

On crée un Provider pour gérer la logique

```
export const DataProvider = ({ children }) => {
  const [data, setData] = useState([])

  const fetchData = {...}

  useEffect(() => {
    (async () => await fetchData())()
  }, [])

  return (
    <DataContext.Provider
      value={{
        data,
        fetchData,
      }}
    >
      {children}
    </DataContext.Provider>
  );
};
```

On crée un hook relie au contexte

```
export const useData = () => useContext(DataContext)
```

Autre lien pour la culture : <https://www.freecodecamp.org/news/how-to-use-flux-in-react-example/>

Création de vues

on Wrap l'element auquel on veut passer les etats

```
Return (  
  <DataProvider>  
    <App />  
  </DataProvider>  
)
```

App :

```
<div>  
  <DataList />  
  <DataList2 />  
  <AutreComponent />  
</div>
```

On utilise le hook pour recuperer les actions et etats du context

```
DataList:  
  
function DataList() {  
  Const {data, fetchData } = useData()  
  
  ...  
}
```

Grace a ça nous évitons le props drilling !

Les différents Routers

MemoryRouter

- Ne change pas l'URL dans votre navigateur
- Conserve les changements d'URL en mémoire RAM
- Pour environnements de test
- Pas d'historique.

HashRouter

- Utilise le «hash routing » côté client.
- Chaque fois qu'une nouvelle route est rendue, elle met à jour l'URL du navigateur avec des routes de hachage.
- Hachage de l'URL pas gérée par le serveur.
- La valeur de hachage sera gérée par le routeur React.
- Prendre en charge les navigateurs hérités*.
- Aucune configuration dans le serveur.
- Pas recommandé par l'équipe React.

BrowserRouter

- Basé sur HTML pushState API
- fonctionne comme une URL normale
- Serveur gère toute l'URL de la requête
- BrowserRouter s'occupe du routage de la page concernée.
- forceRefresh pour prendre en charge les navigateurs hérités*.

HashRouter

```
import React from 'react';
import { HashRouter, Route } from 'react-router-dom';
import Posts from './Pages/Posts';
import LandingPage from './Pages/LandingPage';

export const App = () => (
  <HashRouter>
    <Route
      exact
      path="/"
      component={LandingPage}
    />
    <Route
      exact
      path="/posts"
      component={Posts}
    />
  </HashRouter>
)
```

Output

- http://localhost:3000/#/
- http://localhost:3000/#/posts

Conséquences sur le déploiement

Le serveur doit gérer l'url '/'

BrowserRouter

```
import React from 'react';
import { HashRouter, Route } from 'react-router-dom';
import Posts from './Pages/Posts';
import LandingPage from './Pages/LandingPage';

export const App = () => (
  <BrowserRouter>
    <Route
      exact
      path="/"
      component={Home}
    />
    <Route
      exact
      path="/posts"
      component={Posts}
    />
  </BrowserRouter>
)
```

Output

- http://localhost:3000/
- http://localhost:3000/posts

Conséquences sur le déploiement

Le serveur doit gérer chaque url rendre chaque
Plusieurs routes (Amélioration SEO)

Node.Js

Il s'agit d'un environnement serveur pour le JavaScript, construit en utilisant le moteur JavaScript de Google V8.

Node est conçu pour gérer un très grand nombre de processus concurrents : par exemple des milliers de connexion HTTP et/ou Websocket, de l'écriture dans des fichiers en parallèle etc.

Node est également conçu pour gérer des flux très importants de données.

Node est donc excellent pour le Web et possède d'excellente performance.

Next.Js

Next.js est un framework construit avec React, Node.js, Babel et Webpack.

Next.js vous permet d'exporter votre site Web de manière statique au format HTML ou de le rendre sur le serveur. Il divise automatiquement votre code, ce qui réduit la taille de votre bundle et rend votre application rapide, car, à la fin, seul le JavaScript nécessaire sera chargé sur le navigateur.

Next.js active le routage dans votre application à l'aide du routage basé sur le système de fichiers. Il utilisera automatiquement les fichiers situés dans le dossier pages comme route, ce qui signifie que vous n'avez même pas besoin d'ajouter une bibliothèque supplémentaire pour créer une application multi-pages avec Next.

Formation ReactJS : Développement Web

Coordonnées

formateur@plb.fr

www.plb.fr

par Guillaume Schoder