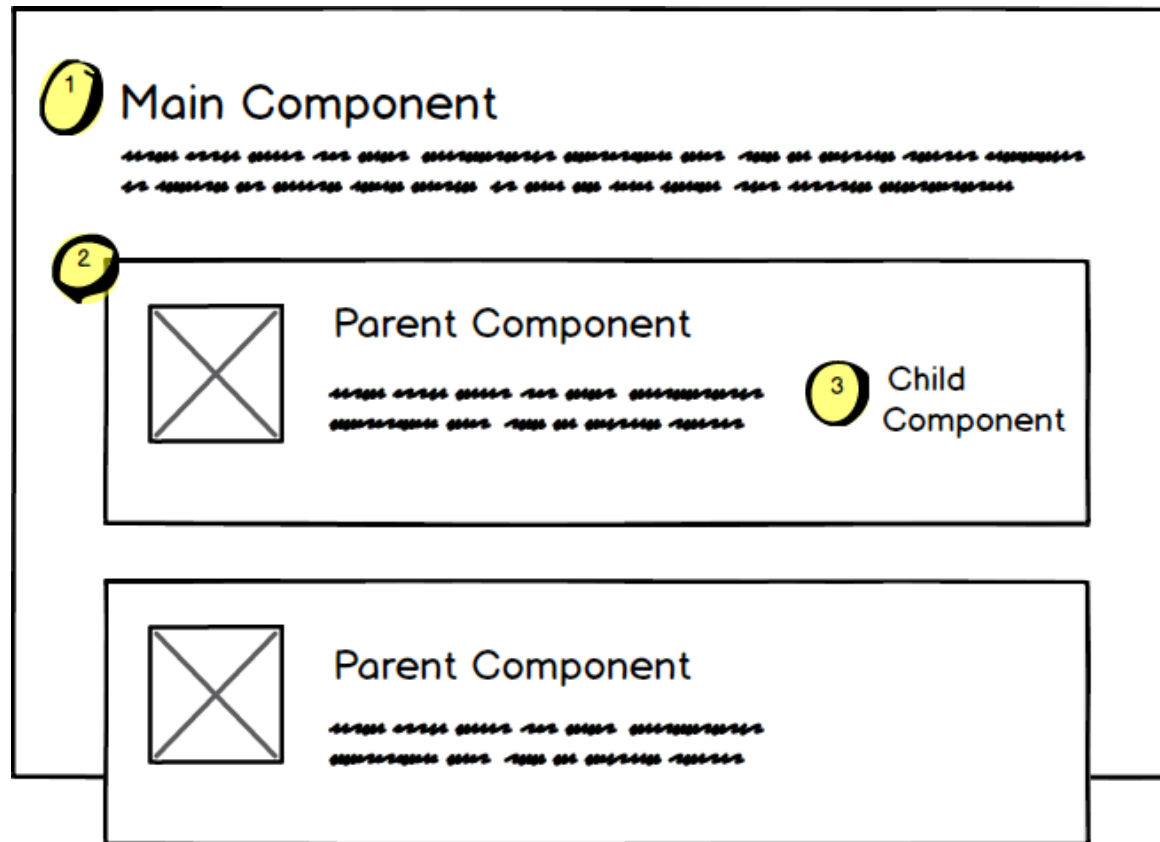


Manipuler les composants

Animé par Mazen Gharbi

Comprendre les composants

“ Not only do React components map cleanly to UI components, but they are self-contained. The markup, view logic, and often component-specific style is all housed in one place. This feature makes React components reusable. ”

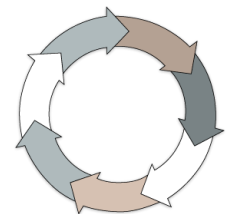


Cycle de vie des composants

► Un composant bénéficie automatiquement de méthodes de gestion de son cycle de vie :

- › [componentDidMount](#): le composant a été créé et son contenu a été ajouté dans le DOM
- › [shouldComponentUpdate](#): permet d'éviter (ou pas) le ré-interpréter du composant. Pratique si les props ou le state sont modifiés de manière intensive
- › [componentDidCatch](#): Un composant enfant a déclenché une erreur (et ne peut s'afficher)

[Plus d'infos ici](#)



Functional component

- ▷ Un composant fonctionnel est une fonction qui prend les props en paramètres et retourne du JSX ;
- ▷ Aucun state ;
- ▷ Ne bénéficie pas des méthodes de gestion de cycle de vie ;
- ▷ Très léger !

```
import React from 'react';  
  
export default function Hello() {  
  return <div>Bonjour tout le monde !</div>;  
}
```

<Hello />

On l'appelle comme un
composant habituel

Les bonnes pratiques ont changées

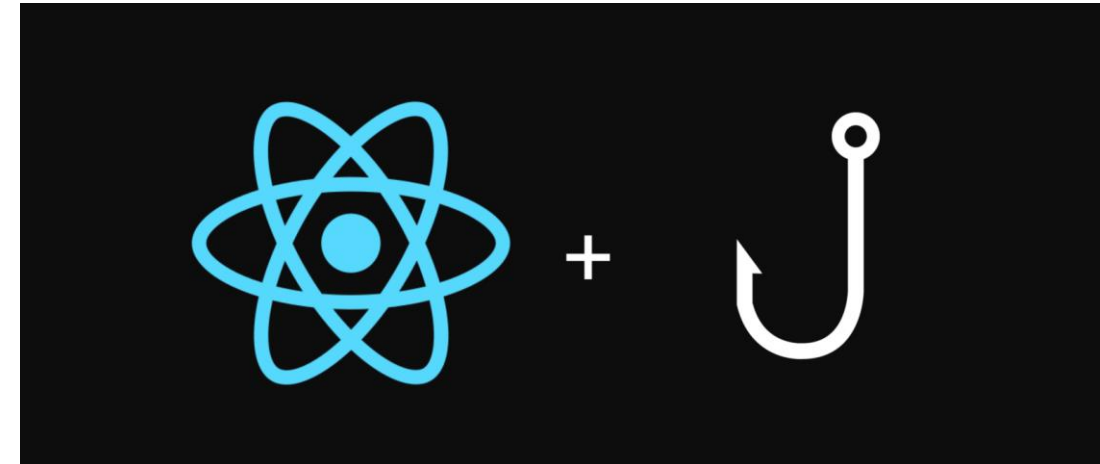
- ▷ Désormais, il s'agira d'implémenter des « functional component » **TOUT le temps !**
- ▷ Simplifie le développement
 - › Evite les méthodes alambiquées (componentDidMount / etc.)
 - › Solutionne le problème du binding this
 - › Plus accessible pour un débutant React
- ▷ Mais la **gestion du state évolue.**

Un functional component

```
export default function MonComposant() {  
  const b = 'Bonjour';  
  
  function reagir() {  
    console.log("b : " + b);  
    console.log("Vous avez cliqué sur le bouton");  
  }  
  
  return (  
    <div>  
      <p>Ce composant est fonctionnel !</p>  
      <p>Valeur de b = {b}</p>  
      <button onClick={reagir}>Cliquez ici</button>  
    </div>  
  )  
}
```

Les hooks

- ▷ Avec les composants fonctions, plus de « `this.state` »
 - › Et plus de mot-clé `this` avec le comportement étudié précédemment !
- ▷ Pour les remplacer, l'équipe React a intégrée les « Hooks »
- ▷ Un hook permet de **gérer un état** localisé
- ▷ On peut avoir **plusieurs hooks** !
- ▷ Nouvelle méthode : « `useState` »
 - › Renvoie 2 valeurs !



Exemple de hook

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

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```


Composition

- ▷ Avec les **classes**, on réfléchissait notre application comme une **imbrication de composants**
- ▷ Avec les **fonctions**, on réfléchit notre application comme une **imbrication de fonctionnalités**
- ▷ Logiquement plus modulaire – L'objectif principal des hook a été de simplifier l'architecture d'un projet React conséquent
- ▷ Avec les hooks, même le state peut être extrait d'un composant !
 - › Permet donc la factorisation de bien plus de chose qu'auparavant

Code plus lisible

► Avec les hooks, le code suit un cheminement bien plus logique

```
import * as React from "react";
import { Card, Row, Input, Text } from "../components";
import ThemeContext from "../ThemeContext";

export default class Greeting extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "Harry",
      surname: "Potter",
      width: window.innerWidth
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleResize = this.handleResize.bind(this);
    this.handleSurnameChange = this.handleSurnameChange.bind(this);
    window.addEventListener("resize", this.handleResize);
    document.title = this.state.name + ' ' + this.state.surname;
  }

  componentDidMount() {
    document.title = this.state.name + ' ' + this.state.surname;
  }

  componentWillUnmount() {
    window.removeEventListener("resize", this.handleResize);
  }

  handleChange(name) {
    this.setState({ name });
  }

  handleSurnameChange(surname) {
    this.setState({ surname });
  }

  handleResize() {
    this.setState({ width: window.innerWidth });
  }

  render() {
    let { name, surname, width } = this.state;
    return (
      <ThemeContext.Consumer>
        {theme => (
          <Card theme={theme}>
            <Row label="Name">
              <Input value={name} onChange={this.handleChange} />
            </Row>
            <Row label="Surname">
              <Input value={surname} onChange={this.handleSurnameChange} />
            </Row>
            <Row label="Width">
              <Text>{width}</Text>
            </Row>
          </Card>
        )}
      </ThemeContext.Consumer>
    );
  }
}
```

Class VS Hooks

```
import React, { useState, useContext, useEffect } from "react";
import { Card, Row, Input, Text } from "../components";
import ThemeContext from "../ThemeContext";

export default function Greeting(props) {
  let theme = useContext(ThemeContext);

  let [name, setName] = useState("Harry");
  let [surname, setSurname] = useState("Potter");
  useEffect(() => {
    document.title = name + " " + surname;
  });

  let [width, setWidth] = useState(window.innerWidth);
  useEffect(() => {
    let handleResize = () => setWidth(window.innerWidth);
    window.addEventListener("resize", handleResize);
    return () => {
      window.removeEventListener("resize", handleResize);
    };
  });

  return (
    <Card theme={theme}>
      <Row label="Name">
        <Input value={name} onChange={setName} />
      </Row>
      <Row label="Surname">
        <Input value={surname} onChange={setSurname} />
      </Row>
      <Row label="Width">
        <Text>{width}</Text>
      </Row>
    </Card>
  );
}
```

useState

▷ « useState » est notre nouvelle méthode pour **manipuler l'état** du composant actuel

› Nous ne manipulons plus d'objet state global désormais et tant mieux !

```
const [val, setVal] = useState("Valeur initiale");
```

▷ Renvoie un tableau contenant la valeur initiale et un setter

▷ Appeler le « setVal » provoque la réinterprétation de la vue

▷ Fonctionne également pour un objet

```
const [val, setVal] = useState({ prop: "Valeur initiale" });
```

Hooks d'effets

- ▶ Avec les functional components, plus les méthodes pour interagir avec le cycle de vie d'un composant

```
class MyComp extends React.Component {  
  componentDidMount() {}  
  
  componentDidUpdate() {}  
  
  componentWillUnmount() {}  
}
```

- ▶ Cette logique a été remplacée par les hook d'effet :

```
const [val, setVal] = useState("Ma valeur initiale");  
  
// Permet de réagir à la création du composant ET à chaque modification des variables  
useEffect(() => {  
  console.log("Hello ! Vue réinterprétée :)");  
});
```

Hooks d'effets - Nettoyer la mémoire

▷ En retournant une fonction, nous pouvons appliquer un comportement à la « mort » du composant :

```
useEffect(() => {  
  return () => {  
    // Code à exécuter lors de la mort du composant  
  };  
});
```

▷ Un second paramètre permet de spécifier à quelles variables on souhaite réagir

```
export default function App({ color }) {  
  useEffect(() => {  
    return () => {  
      console.log('La couleur a changée !');  
    };  
  }, [color]);  
}
```

useEffect est réexécuté à chaque fois !

▷ Ce qui signifie qu'un code comme celui-ci :

```
useEffect(() => {  
  UsersAPI.connect(onUserUpdate, props.user.id);  
  return () => {  
    UsersAPI.disconnect(onUserUpdate);  
  };  
});
```

▷ Va provoquer l'abonnement et le désabonnement au service à chaque réinterprétation... Pas toujours le meilleur choix !

› React évite ainsi certains bugs si les props changent entre temps

```
useEffect(() => {  
  UsersAPI.connect(onUserUpdate, props.user.id);  
  return () => {  
    UsersAPI.disconnect(onUserUpdate);  
  };  
}, [props.user.id]);
```

→ Bien mieux pour les performances !

Construire nos propres Hooks

▷ Comme énoncé au départ, objectif : réutilisabilité !

```
function useLoadUsers() {  
  const [users, setUsers] = useState(null);  
  
  useEffect(() => {  
    const subscription = UserAPI.loadUsers().subscribe((users) => {  
      setUsers(users); // Provoque la réinterprétation de la vue  
    });  
  
    return () => {  
      subscription.unsubscribe();  
    };  
  }, []); // [] => Permet d'enclencher le hook uniquement 1 fois au chargement du composant  
  
  return users;  
}
```

```
function DisplayUsers(props) {  
  const users = useLoadUsers();  
  
  return users ? 'Aucun utilisateur' : `Il y a ${users.length} utilisateurs`;  
}
```

PropTypes

- ▷ Permettent de définir les types des propriétés attendues
 - › Non obligatoires mais vivement recommandées ;
- ▷ Vérification qui facilite la collaboration entre différents développeurs
 - › Affiche un **warning** dans la console si une propriété ne respecte par la propTypes définie ;
- ▷ N'impactent pas les performances de votre application en prod
 - › Car effacées lors de la création de la phase de build

[En savoir plus ici](#)

PropTypes

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

export default function Hello(props) {
  const { firstname, lastname } = props;
  return (
    <div>
      Bonjour {firstname} {lastname}, comment tu vas ?
    </div>
  );
}

Hello.propTypes = {
  firstname: PropTypes.string.isRequired,
  lastname: PropTypes.string
};
```

Destructuration d'objet

Permet d'indiquer que ce champs est requis

Children

Souvenez-vous

- ▷ « *Seulement trois props sont réservées par React :*
key: différencie les composants dans une liste à l'aide d'un identifiant unique ([en savoir plus](#))
ref: permet de manipuler directement l'élément dans le DOM
children: liste des composants enfants »
- ▷ Il est possible d'entrer du contenu dans le corps d'appel d'un component afin de le récupérer par la suite via la propriété children

Children

```
function FancyBorder(props) {  
  return (  
    <div className={"FancyBorder FancyBorder-" + props.color}>  
      {props.children}  
    </div>  
  );  
}  
-----  
function Dialog(props) {  
  return (  
    <FancyBorder color="blue">  
      {  
        <h1 className="Dialog-title">{props.title}</h1>  
        <p className="Dialog-message">{props.message}</p>  
      }  
    </FancyBorder>  
  );  
}
```

Nos childrens

Construire son projet

- ▷ On va essayer de créer des composants de type « **data-driven** » :
 - › Pour y arriver, on maximisera l'utilisation des « props » ;
 - › Facilitera ainsi la **ré-utilisabilité**
- ▷ Les props vous permettent également de passer des références de fonction

The diagram illustrates a transformation of a React component. On the left, a component is shown in a data-driven state where all data and logic are passed as props. On the right, the same component is shown in a stateful state where internal state and logic are managed within the component, and only the necessary props are passed. A vertical dashed line and a horizontal arrow indicate the transition between these two states.

```
<Product
  key={"product-" + product.id}
  id={product.id}
  title={product.title}
  description={product.description}
  url={product.url}
  votes={product.votes}
  submitterAvatarUrl={product.submitterAvatarUrl}
  productImageUrl={product.productImageUrl}
  onVote={this.handleProductUpVote}
/>
```

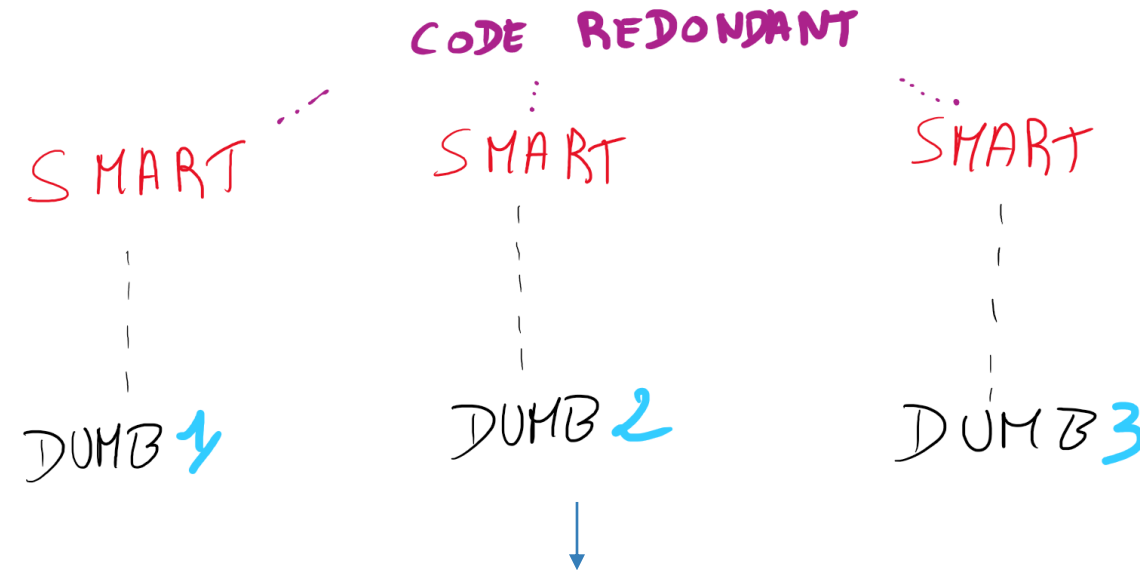
→

```
<Product
  key={"product-" + product.id}
  {...product}
  onVote={this.handleProductUpVote}
/>
```

High Order Components

- ▷ Technique avancée pour optimiser la réutilisabilité
- ▷ Pour rappel, un composant permet de générer une vue réutilisable
- ▷ Un HOC est un composant **transforme un composant en... un autre composant** ! Cela nous permettra de factoriser des fonctionnalités entre différents composants
- ▷ Il y a pas les mixins pour ça ?
 - › **Mauvaise pratique ! En savoir plus [ici](#)**

High Order Components



- Dans ce scénario, le H.O.C. permettrait d'automatiser la « génération » des **Smart** en prenant en paramètre leurs **Dumb** respectifs

Un exemple très simple

```
export default function HOC(Component) {  
  return (color, { ...props }) => {  
    let colorToApply = color;  
    if (color === 'red') {  
      colorToApply = 'blue';  
    }  
  
    return <Component color={colorToApply} {...props} />;  
  };  
}
```

```
export default function App() {  
  const NewList = HOC(MaListe);  
  const NewSayHi = HOC(SayHi);  
  return (  
    <div>  
      <NewList />  
      <SayHi />  
    </div>  
  );  
}
```



Routing

Deep Linking

https://www.macademia.fr/admin/users.php

Protocole Nom de domaine Chemin vers la ressource

- ▶ Habituellement, le « **pathname** » correspond au chemin vers la ressource que l'on souhaite récupéré du serveur ;
- ▶ Nous travaillons actuellement sur une Single Page Application ;
 - › Nous n'accéderons plus à des ressources serveur directement ;
- ▶ Mais un système de routing est tout de même nécessaire
 - › Partager une page du site ;
 - › Recharger la page et rester sur la même vue *etc.* ;

Routes

- ▷ Le système de routing en React est une sorte de « *mémorisateur d'état externe* »
 - › Il sert par exemple à ce qu'un utilisateur enregistre la page actuel en favori
- ▷ Comme toute nouvelle fonctionnalité, il est nécessaire d'installer une librairie pour manipuler le routing :

```
> npm install --save react-router react-router-dom
```

- ▷ Meilleur librairie du moment pour la gestion des routes

Le serveur n'est pas requêté aux changements de pages en React !

Mise en place

- Pour mettre en place les routes, il va être nécessaire de définir un **Context Route** autour de composant principal :

```
import { BrowserRouter as Router } from 'react-router-dom';  
  
<Router>  
  <App />  
</Router>
```

Alias utilisé

Composant racine

- Ainsi, le système de routing est initialisé et on pourra :
 - › Définir nos routes ;
 - › Naviguer entre nos pages.

Déclaration des pages

► En React, chaque page est définie par un composant

```
// On est actuellement dans le composant App  
<Route path='/accueil' element={<HomeComponent />} />  
<Route path='/utilisateurs' element={<UsersComponent />} />  
<Route path='/details' element={<DetailsComponent />} />
```

Composants créés
au préalable



► Le mot-clé « **exact** » permet d'indiquer que l'on souhaite que la route soit EXACTEMENT celle-ci

- › Par défaut, le système de routing est très permissif, et une route comme « **/detailsblabla** » OU « **/details/1/2** » est valide pour afficher DetailsComponent

Naviguer entre les routes

- ▷ Pour la redirection, « react-router-dom » fournit un composant « **Link** »
- ▷ Un attribut « to » permet de spécifier vers quel route on souhaite rediriger :

```
<Link to="/utilisateurs">Voir les utilisateurs du site :)</Link>
```

- ▷ Peut également être un objet :

```
<Link  
to={{  
  pathname: '/courses',  
  search: '?sort=name',  
  hash: '#the-hash',  
  state: { fromDashboard: true }  
}}  
>;
```

The diagram shows four blue arrows pointing from text labels to specific parts of the code above:

- An arrow from "Route visé" points to the `pathname: '/courses'` line.
- An arrow from "Paramètres GET" points to the `search: '?sort=name'` line.
- An arrow from "Ancre" points to the `hash: '#the-hash'` line.
- An arrow from "Mise à jour du state dans le composant de la route sur laquelle on arrive" points to the `state: { fromDashboard: true }` line.

Mise en place du routing

- ▷ En React, chaque route est représentée par un Composant ;

<https://stackblitz.com/edit/macademia-react-router-basic>

- Route imbriquée : <https://stackblitz.com/edit/macademia-react-router-imbrique>

Routes

app.js

```
class App extends Component {
  constructor(props) {
    super(props);
    this.state = { name: 'React' };
  }

  render() {
    return (
      <Router>
        <Routes>
          <Menu />
          <Route path='/accueil' element={<Accueil />} />
          <Route path='/contact' element={<Contact />} />
          <Route path='/forum' element={<Forum />} />
        </Routes>
      </Router>
    );
  }
}
```

Routes imbriquées

```

<Routes>
...
  <Route path='/accueil' element={<Accueil />} />
    <Route path='enfant1' element={<RouteEnfant1 />} />
    <Route path='enfant2' element={<RouteEnfant2 />} />
  </Route>
</Routes>

```

App.js

On déclare toutes les routes enfants comme children de la route parent

```

render() {
  return (
    <div>
      Contenu de le page Accueil -
      <Link to='/accueil/details'>
        <a> Détails</a>
      </Link>
      <Outlet />
    </div>
  );
}

```

Accueil.js

Provoque l'affichage des routes enfants

Paramètres de route

- ▷ Les paramètres d'URL sont des paramètres dont les valeurs sont dynamiques dans l'URL ;
- ▷ Un exemple pratique serait les pages de profil de Twitter. Si rendu par React Router, cette route peut ressembler à cela :

```
import Profile from './pages/profile.component';
```

```
<Route path('/:handle' element={Profile} />
```



Le « : » indique que c'est un paramètre de route qui porte le nom « handle »

Paramètres de route

```
import React, { Component } from 'react';
import { useParams } from 'react-router';

class Profile extends Component {
  constructor(props) {
    super(props);
    this.state = {
      user: null
    };
  }

  componentDidMount() {
    // Récupération du paramètre de route
    const { handle } = useParams();

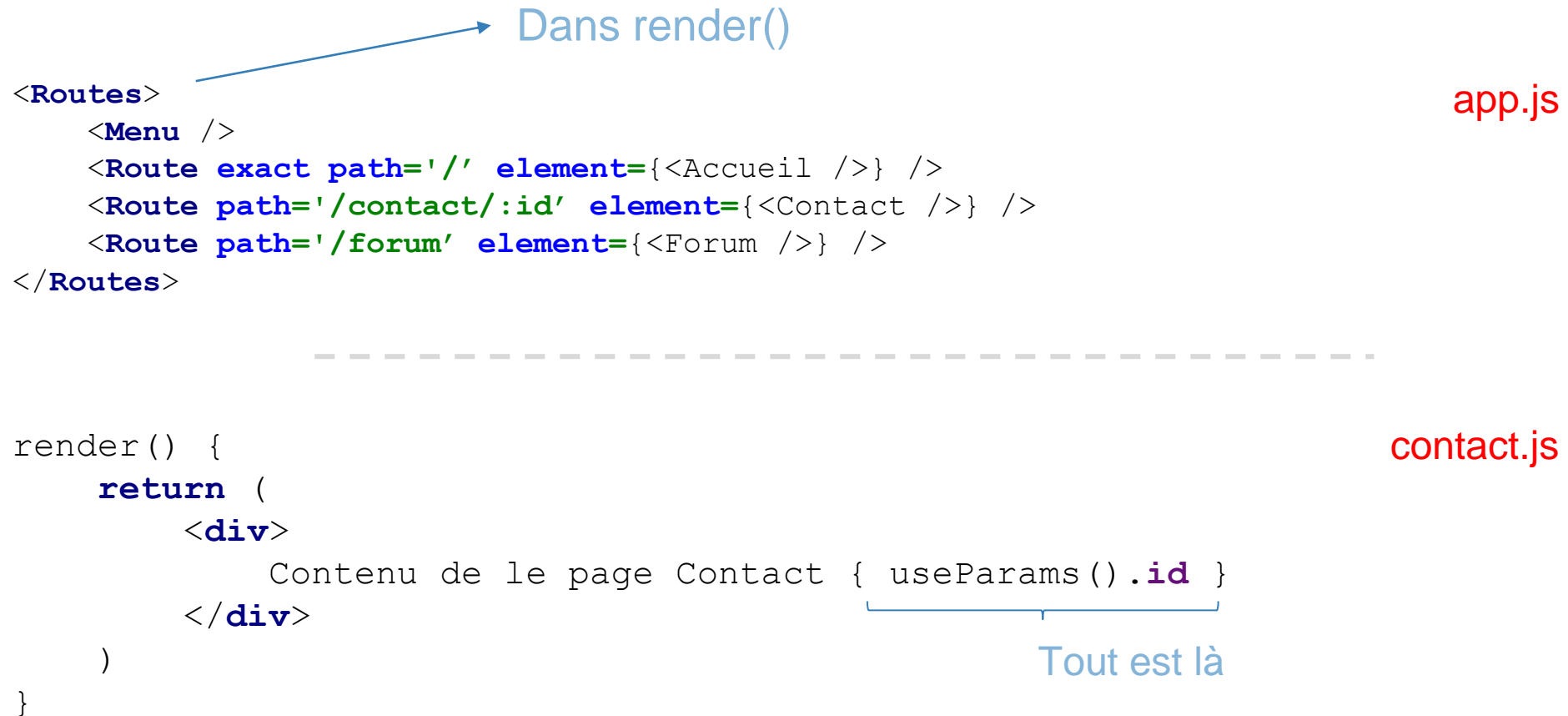
    // fetch permet de requêter un serveur
    fetch(`https://api.twitter.com/user/${handle}`).then((user) => {
      this.setState((previousState) => {
        return {
          user // remplacé par user: user
        };
      });
    });
  }

  render() {
    ...
  }
}
```

C'est ici que tout se joue

Paramètres URL

▷ Voici un exemple : <https://stackblitz.com/edit/macademia-react-router-params>



app.js

```
<Routes>
  <Menu />
  <Route exact path="/" element={<Accueil />} />
  <Route path="/contact/:id" element={<Contact />} />
  <Route path="/forum" element={<Forum />} />
</Routes>
```

contact.js

```
render() {
  return (
    <div>
      Contenu de le page Contact { useParams().id }
    </div>
  )
}
```

Tout est là

Rediriger avec les hooks !

▷ <https://stackblitz.com/edit/react-router-avec-hooks>

```
export default function App() {  
  const nav = useNavigate();  
  const location = useLocation();  
  
  return (  
    <div>  
      <p>Route actuelle : {location.pathname}</p>  
    <ul>  
      <button onClick={() => nav('/home')}>Home</button>
```

```
<Route path="/home" component={Home} />  
<Route path="/contact/:name" component={Contact} />
```

```
export default function Contact() {  
  const { name } = useParams();  
  ...
```



Formulaire

Formulaires en React

- ▷ Nos formulaires en React seront gérés à travers nos composants ;
- ▷ Il existe 2 types de composants pour les gérer :
 - › **Controlled** components ;
 - › **Uncontrolled** components.
- ▷ Quelle différence ?

Controlled component

▷ Un « **Controlled Component** » est un composant sur lequel nous choisissons nous-même la valeur qui apparaît :

```
class ControlledInput extends React.Component {  
  render() {  
    return <input type="text" value="valeur de l'input"/>  
  }  
}
```

ControlledInput.js

▷ Si l'utilisateur insère une nouvelle valeur dans l'input, rien ne se passera. Afin de contourner ce problème, on fait en sorte que le composant React écoute les changements de l'utilisateur pour mettre à jour la valeur

Controlled component

▷ Voici ce qu'on obtiendra :

ControlledInput.js

```
function ControlledInput {  
  const [value, setValue] = useState('');  
  
  function onChange(event) {  
    setValue(event.target.value);  
  }  
  
  return <input  
    type="text"  
    value={value}  
    onChange={onChange}  
  />  
}
```

Le state décide du contenu de l'input

Evenement React, réagit aux changements sur l'input

Uncontrolled component

- ▷ Un « *Uncontrolled Components* » est un composant sur lequel la mise à jour de la valeur ne change pas l'état de notre composant. La valeur de l'input sera donc toujours celle envoyée par l'utilisateur. (inverse du controlled component)

```
function UncontrolledInput {  
    return <input type="text" defaultValue="valeur de l'input"/>  
}
```

- ▷ Comment récupérer la donnée entrée par l'utilisateur ?

Uncontrolled component

▷ Via une référence :

```
<input type="text" ref={ (ref) => input = ref} />
```

▷ Avec un listener :

```
function Form {  
  const onSubmit = (event) => {  
    event.preventDefault();  
    let data = new FormData(event.target);  
  };  
  
  return (  
    <form onSubmit={onSubmit}>  
      <input type="text" name="input_name"/>  
      <button>Submit</button>  
    </form>  
  )  
}
```

Les références avec les classes

► Comme vu précédemment, il existe un moyen pour relier les éléments du DOM avec le modèle grâce à la propriété « ref » ;

```
componentDidMount() {  
    console.log(this.refs.element) ;  
}  
  
render() {  
    return (<div ref="element"> Contenu de la div </div>) ;  
}
```

Et avec les functional components ?

- ▷ C'est une autre paire de manche
- ▷ Il va être nécessaire de créer une référence au préalable :

```
export default function MiniForm() {  
  const firstnameRef = useRef();  
  
  function onSubmitForm(event) {  
    event.preventDefault();  
    console.log(firstnameRef.current.value);  
  }  
  
  return (  
    <form onSubmit={onSubmitForm}>  
      <input ref={firstnameRef} />  
      <button type="submit">Valider</button>  
    </form>  
  );  
}
```

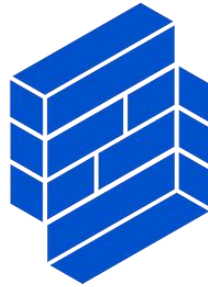
Création de la référence

Annule le rechargement de la page

On applique la référence à l'input

Exemple

<https://stackblitz.com/edit/react-macademia-simple-form>



Formik

Formik – un incontournable

- ▷ L'une des grandes forces de React, c'est sa gestion dynamique de formulaire
- ▷ Mais le code pour implémenter une gestion efficace est **lourd** et **redondant**
- ▷ Formik va nous faire gagner beaucoup de temps et d'énergie

```
> npm install formik --save
```

Un composant Formik très puissant

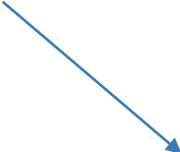
► Formik fournit un composant paramétrable pour englober vos formulaires

› On va mettre en place un **contexte Formik** autour du formulaire

```
const valeursInitiales = {firstname: 'Joe', lastname: 'Dupond'};

const onValidate = values => {
  // Renvoie un objet vide si tout est OK
  // Et contenant des propriétés si des erreurs sont repérées
  console.log(values); // {firstname: '...', lastname: '...'}
  return {firstname: 'Une erreur car je le veux.'};
};

const onSubmit = ({values, meta}) => {
  console.log(values); // {firstname: '...', lastname: '...'}
  // Si ça nous va, alors on envoie l'information au serveur
  setTimeout(() => { // Simule l'appel au back
    meta.setSubmitting(false); // On indique que le chargement est terminé
  }, 1000)
};
```



```
<Formik
  initialValues={ valeursInitiales }
  validate={ onValidate }
  onSubmit={ onSubmit }
  // Et pleins d'autres paramètres optionnels
>
  <!-- Notre formulaire ici -->
</Formik>
```


Formulaire avec les composants Formik

- Une fois le contexte préparé, il s'agit maintenant de créer notre formulaire puis de le relier à Formik
 - › Nous allons pleinement profiter du contexte Formik en mettant en place des composants (consumer) fournis par la librairie

```
import { Formik, Form, Field, ErrorMessage } from 'formik';

// ...

<Formik initialValues={ valeursInitiales } validate={ onValidate } onSubmit={ onSubmit }>
  {{{ isSubmitting, ... }} => (
    <Form>
      <Field type="text" name="firstname" />
      <ErrorMessage name="firstname" component="div" />
      <Field type="text" name="lastname" />
      <ErrorMessage name="lastname" component="div" />
      <button type="submit" disabled={ isSubmitting }>
        Valider
      </button>
    </Form>
  )}
</Formik>
```

Formulaire avec les composants Formik

▷ Sinon, on peut également s'abstraire des composants Formik...

```
import { Formik } from 'formik';
// ...
<Formik initialValues={ valeursInitiales } validate={ onValidate } onSubmit={ onSubmit }>
  ({ values, errors, isSubmitting, handleSubmit, handleChange, handleBlur }) => (
    <form onSubmit={handleSubmit}>
      <input
        name="firstname"
        onChange={handleChange}
        onBlur={handleBlur}
        value={values.firstname}
      />
      { errors.firstname && touched.firstname && <p>errors.firstname</p> }
      /* ... */
      <button type="submit" disabled={isSubmitting}>
        Valider
      </button>
    </form>
  )
</Formik>
```

Validation

- ▷ On gagne effectivement du temps sur la gestion du formulaire..
Mais qu'en est-il de **la validation** ?
 - › **C'est la limite de Formik**
- ▷ Bah... C'est à nous de la mettre en place – manuellement !
- ▷ Pour la gagner du temps, nous partirons plutôt avec une librairie complémentaire très populaire : **YUP**

```
➤ npm install yup --save
```

YUP – Validation des champs

- ▷ Formik possède une configuration adaptée et spécifique pour Yup
- ▷ L'implémentation se fera au travers des **schémas de validation**
- ▷ Et la mise en place correspond à la création du **schéma** :

```
const ConnectionSchema = Yup.object().shape({  
  firstname: Yup.string()  
    .min(2, 'Trop court :')  
    .max(50, 'Trop long cette fois !')  
    .required('Ce paramètre est requis'),  
  lastname: Yup.string()  
    .min(2, '...')  
    .max(50, '...')  
});
```

Mise en place du schéma de validation

► Pour ça, rien de plus simple :

```
<Formik
  initialValues={ valeursInitiales }
  validate={ onValidate }
  onSubmit={ onSubmit }
  validationSchema={ ConnectionSchema }
  // Et pleins d'autres paramètres optionnels
>
  <!-- Notre formulaire ici -->
</Formik>
```

► Et c'est gagné, la gestion d'erreur sera automatiquement relié à Yup

```
{errors.firstname && touched.firstname && <p>errors.firstname</p>}
```



ImmutableJS

ImmutableJS

- ▷ Un state doit être considéré comme **immutable** !
 - › Evite ainsi les erreurs d'affichage dues à l'accès concurrentiel des datas
- ▷ Soit un reducer gérant notre utilisateur :

```
case UPDATE_DATA:
```

```
    const newState = {...state}; // Nouvelle référence certes  
    // Mais ici on modifie l'ancien state  
    newState.users[action.payload.index].firstname = action.payload.newFirstname;  
    return newState;
```

Ce code n'est donc pas bon.

ImmutableJS

▷ Un code correct serait :

C'est un peu verbeux oui.

```
case UPDATE_DATA:
  const { index, newFirstname } = action.payload;

  return {
    ...state,
    users: this.state.users.map((user, indexIteration) => {
      if(indexIteration === index) {
        return {
          ...user,
          firstname: newFirstname
        }
      }
      return user;
    })
  }
}
```

Long et rébarbatif !

ImmutableJS

- ▷ Librairie qui nous simplifiera la vie
- ▷ Permet de mettre à jour un objet sans modifier la référence initiale
 - › Renvoie un **clone**
- ▷ Librairie à installer
 - › Créée et maintenue par Facebook

```
> npm i immutable
```

- ▷ **Obligation de passer par des Objets Immutable désormais !**

Quelques méthodes pratiques

```
setIn<C>(collection: C, keyPath: Iterable<any>, value: any): C
```

- Permet de naviguer dans une collection (Tableau / Objet) afin de mettre à jour une donnée selon un « keypath »

Exemple :

```
const newObj = oldObj.setIn(['user', 'joe', 'firstname'], valueNewFirstname);
```

```
oldObj = {  
  user: {  
    joe: {  
      firstname: 'Ancien prénom'  
    }  
  }  
}
```

Quelques méthodes pratiques

```
fromJS(jsValue: any, [...]): any
```

- ▷ Permet de convertir un objet Javascript vanilla en un objet Immutable

Exemple :

```
let newState = fromJS(this.state);
```

```
[immutableObject].toJS(): any
```

- ▷ A l'inverse, convertit un objet Immutable en JS vanilla

Exemple :

```
this.setState(newState.toJS());
```

ImmutableJS

Avant

```
increment = (i) => {  
  this.setState({  
    ...this.state, // Plus nécessaire depuis plusieurs mois  
    objects: this.state.objects.map((info, index) => {  
      if (index === i) {  
        return {  
          ...info,  
          value: info.value + 1  
        }  
      }  
    })  
  })  
  return info;  
})  
}
```

Après

```
let newState = fromJS(this.state);  
  
newState = newState.setIn(  
  ["objects", i, "value"],  
  this.state.objects[i].value + 1  
);  
  
this.setState(newState.toJS());
```

Questions ?