zenika

Formation sur mesure

React Avancé

INTRODUCTION

SOMMAIRE

Rappels JavaScript et React

React: Immutabilité du state

```
const [value, setValue] = useState()

const handleValueChange = (v) => {
  setValue(v)
}
```

```
const [value, setValue] = useState()

const handleValueChange = (v) => {
  value = v
}
```

- Ne pas utiliser les fonctions *impures* (`.push`, `.pop`, `delete`, etc...)
- Manipuler le state au travers de copie manuelle :
  - Utiliser le spread operator (`...`)
  - Utiliser des fonctions pures (`.map`, `.filter`, `Object.values`, etc...)
- Manipuler le state au travers de copie générée (`immutable.js`, etc...)

React: Uplifting du state

Une donnée partagée entre deux composants frères doit être remontée dans le *state* parent et transmise aux enfant au travers des props.

La modification de ce *state* par les enfants se fera au travers d'une function de rappel (*callback*) définies dans le parent et transmise aux enfants par les props.

React: Props particulière

- **`children`**

Représente le JSX se trouvant entre la balise ouvrante et la balise fermante du composant.

- **`ref`**

Permet au parent de récupérer une référence vers un objet JavaScript dans `ref.current` (le plus souvent, un noeud DOM)

- **`key`**

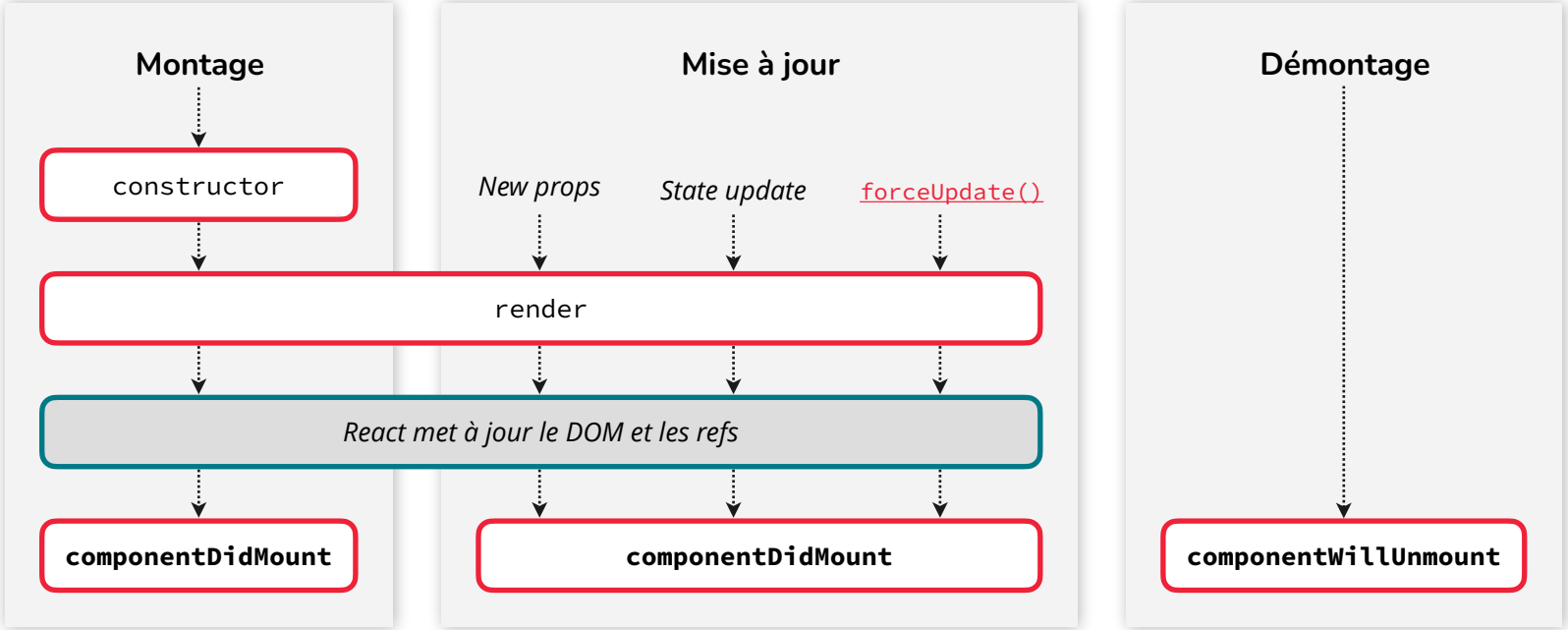
Permet d'assurer l'unicité d'un élément dans une liste à rendre.

L'absence d'unicité mène à une inconsistance du rendu (ex. : mise à jour partielle du rendu, dégradation des performances).

React: Cycle de vie des composants

Vue d'ensemble

https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram



React: Cycle de vie des composants

Cas particuliers

- **`getDerivedStateFromProps`**

Permet de modifier le *state* avant le rendu du composant tout en ayant accès au nouvelles props

- **`shouldComponentUpdate`**

Permet d'optimiser le rendu du composant en évitant les rendu inutiles

- **`getSnapshotBeforeUpdate`**

Permet de lire l'état du DOM avant sa modification pendant le rendu.

- **`componentDidCatch`**

Permet de capturer une exception JavaScript survenu dans un des enfants du composant courant.

JS: Promesses et traitements asynchrones

```
new Promise((resolve, reject) => {
  /* async process */

  if (success) {
    resolve(data)
  } else {
    reject(error)
  }
})
  .then((data) => { /* success handling */ })
  .catch((error) => { /* error handling */ })
```

```
const asyncFun = async () => {
  /* sync process */
  const asyncRes = await otherAsyncFunc()
  /* sync process */

  return data
}

asyncFun()
  .then((data) => { /* success handling */})
  .catch((error) => { /* error handling */})
```

A voir: Jack Archibald : In the loop — JSConf.Asia 2018

JS: Programmation fonctionnel

Curryfication

Transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments.

```
const myFct = (a, b, c) => {
  /* use a, b, and c */
}

const result = myFct(1, 2, 3)
```

```
const myFct = (a) => (b) => (c) => {
  /* use a, b, and c */
}

const myFctWithA = myFct(1)
const myFctWithAnB = myFctWithA(2)
const result = myFctWithAnB(3)

const sameResult = myFct(1)(2)(3)
```

JS: Programmation fonctionnel

Mémoïsation

Une fonction mémoïsée stocke les valeurs renvoyées par ses appels précédents dans une structure de données adaptée et, lorsqu'elle est appelée à nouveau avec les mêmes paramètres, renvoie la valeur stockée au lieu de la recalculer.

```
function memo(fn) {
  const result = {}

  return (...args) => {
    const key = JSON.stringify(args)

    if (!(key in result)) {
      result[key] = fn(...args)
    }

    return result[key]
  }
}
```

```
function doHardStuff(params) {
  /*
    Do some hard computation on
    params to get usable data
  */

  return data
}

const doStuff = memo(doHardStuff)

doStuff({ hello: `world` }) // Do the hard job
doStuff({ hello: `world` }) // got the prev. result
```

JS: Syntax des classes (ES2022)

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

export default class MyComponent extends Component {
  lang = `en`

  static #sayHi = {
    fr: (name) => `Bonjour ${name} !`,
    en: (name) => `Hello ${name}!` }
  }

  sayHi = (name) => MyComponent.#sayHi?.[this.lang]?.(name) ?? name

  render() {
    const { user: name = `john` } = this.props

    const greeting = () => alert(this.sayHi(name))

    return <button onClick={greeting} />
  }
}
```

Les Hooks React

Les règles des Hooks

Les hooks de React sont le moyen donné aux composants fonctions d'interagir avec le cycle de vie de React.

Les hooks sont de simples fonctions JavaScript mais qui doivent suivre impérativement les deux règles ci-contre.

01. **Les Hooks doivent êtres appelés uniquement au niveau racine**

Pour pouvoir fonctionner correctement, le nombre et l'ordre des hooks doit être identique à chaque rendu

02. **Les Hooks doivent êtres appelés uniquement depuis des fonctions React**

Les hooks ne fonctionnerons correctement que s'ils sont appelés depuis un composant fonction ou depuis un hook personnalisé.

Basic hook: useState

useState renvoie une valeur d'état local et une fonction pour la mettre à jour.

```
function MyComponent() {
  const [count, setCount] = useState(0)

  return (
    <button onClick={
      () => setCount(count + 1)
    }>
      {count}
    </button>
  )
}
```

```
function MyComponent() {
  const [fruits, setFruits] = useState({
    apple: 0, banana: 0
  })

  const setApple = () => setFruits(
    (state) => ({ ...state, apple: apple + 1 })
  )

  const setBanana = () => setFruits(
    (state) => ({ ...state, banana: banana + 1})
  )

  return (
    <>
      <button onClick={setApple}>
        Apple: {fruits.apple}
      </button>
      <button onClick={setBanana}>
        Banana: {fruits.banana}
      </button>
    </>
  )
}
```

Basic hook: useEffect

`useEffect` utilise une fonction pour créer des effets de bord.

```
function MyComponent() {
  const [count, setCount] = useState(0)

  useEffect(() => {
    const id = setTimeout(
      () => setCount((n) => n + 1),
      5000
    )

    return () => clearTimeout(id)
  }, [count])

  return (
    <div>{count}</div>
  )
}
```

```
// Component Mount
useEffect(() => { /* … */ })
useEffect(() => { /* … */ }, [])
useEffect(() => { /* … */ }, [data])


// Component Update
useEffect(() => { /* … */ })
useEffect(() => { /* … */ }, [data])


// Clean up on Update & Unmount
useEffect(() => { /* … */ return () => {} })
useEffect(() => { /* … */ return () => {} }, [data])


// Clean up on Unmount
useEffect(() => { /* … */ return () => {} }, [])
```

Uncommon hook: useRef

`useRef` créé une référence vers une valeur qui persistera pendant toute la durée de vie du composant.

```
function MyComponent() {
  const myInput = useRef(null)

  const focus = () => {
    myInput.current?.focus()
  }

  return (
    <>
      <input ref={myInput} type="search" />
      <button onClick={focus}>
        Focus the input
      </button>
    </>
  )
}
```

```
function MyComponent() {
  const timeout = useRef(null)

  const cancel = () => {
    cancelTimeout(timeout.current)
  }

  useEffect(() => {
    timeout.current = setTimeout(
      () => alert(`Boom`),
      60000
    )

    return cancel
  })

  return (
    <button onClick={cancel}>
      Cancel Timeout
    </button>
  )
}
```

Uncommon hook: useReducer

useReducer est une alternative à useState pour mettre a jours des états complexes.

```
function MyComponent() {
  const [{
    apple, banana
  }, dispatch] = useReducer(myReducer, initialState)

  const buyApple = () => dispatch(actions.apple(apple + 1))
  const eatApple = () => dispatch(actions.apple(apple - 1))
  const buyBanana = () => dispatch(actions.banana(banana + 1))
  const eatBanana = () => dispatch(actions.banana(banana - 1))

  return (
    <div>Apple: {apple} :
      <button onClick={buyApple}>Buy</button>
      <button onClick={eatApple}>Eat</button>
    </div>
    <div>Banana: {banana} :
      <button onClick={buyBanana}>Buy</button>
      <button onClick={eatBanana}>Eat</button>
    </div>
  )
}
```

```
const initialState = {
  apple: 0, banana: 0
}

const reducers = {
  apple: (state, action) => ({
    ...state,
    apple: Number(action.payload) || 0
  }),
  banana: (state, action) => ({
    ...state,
    banana: Number(action.payload) || 0
  })
}

const actions = {
  apple: (payload) => ({
    type: `apple`, payload
  }),
  banana: (payload) => ({
    type: `banana`, payload
  })
}

function myReducer(state, action) {
  return reducers[action.type]?.(state, action)
}
```

Optimize hook: useMemo

useMemo retourne la valeur mémoïsé d'une fonction de création.

```
// Un composant mémoïsé qui ne sera re-rendu que si ses props changent
const MyOtherComponent = React.memo(
  ({ options }) => <pre>{JSON.stringify(options, null, 2)}</pre>
)

function MyComponent({ data }) {
  const options = useMemo(
    // Génère un objet d'options horodaté
    () => ({ data, date: new Date().toISOString() }),
    // L'objet d'options ne sera régénéré que si data change
    [data]
  )

  return (
    <>
      <MyOtherComponent options={options} />
      Last Update: <time datetime={options.date}>
        {Date(options.date)}
      </time>
    </>
  )
}
```

Optimize hook: useCallback

`useCallback` retourne une fonction de rappelle mémoïsée.

```
function MyComponent({ name }) {
  const sayHi = useCallback(
    // La fonction de rappelle à mémoïser
    () => alert(`Hi ${name}!`),

    // La fonction de rappelle ne sera mise à jour que si name change
    [name]
  )

  // Le bouton ne sera re-rendu que si sayHi change
  return (
    <button onClick={sayHi}>Say Hi.</button>
  )
}
```

Custom hooks

Toute fonction commençant par **use** est un hook personnalisé qui vous autorise à utiliser n'importe quel hook et qui pourra lui-même être utilisé par d'autres hooks ou par des composants React.

```
function useData(url) {
  const [data, setData] = useState(null)

  const load = useCallback(() => {
    fetch(url)
      .then((res) => res.json())
      .then(setData)
  }, [url])

  useEffect(() => load, [])

  return [data, load]
}
```

```
function MyComponent({ url }) {
  const [data, reload] = useData(url)

  if (!data) {
    return <div>Loading…</div>
  }

  return (
    <pre>{JSON.stringify(data, null, 2)}</pre>
    <button onClick={reload}>
      Reload data
    </button>
  )
}
```

React Router V6

Les nouveautés de React Router V6

- **`<Routes>`** est le remplaçant de **`<Switch>`**
- **`<Route element>`**
- Disparition des props **`strict`** et **`exact`**
- Simplification des routes relatives et des routes imbriquées
- **`useNavigate`** remplace **`useHistory`**

A simple example

```
// V5 example

import {
  BrowserRouter, Switch, Route
} from 'react-router-dom';

const App = () => (
  <BrowserRouter>
    <Switch>
      <Route
        exact path="/users"
        component={UserList}
      />
      <Route
        path="/users/:id"
        render={({ match }) => (
          <User id={match.param.id} />
        )}
      />
    </Switch>
  </BrowserRouter>
)
```

```
// V6 example

import {
  BrowserRouter, Routes, Route
} from 'react-router-dom';

const App = () => (
  <BrowserRouter>
    <Routes>
      <Route
        path="/users"
        element={<UserList />}
      />
      <Route
        path="/users/:id"
        {/* User doit utiliser
            le hook useParam */}
        element={<User />}
      />
    </Routes>
  </BrowserRouter>
)
```

Routes relatives et <Outlet>

```
// V6 example

import {
  BrowserRouter, Routes, Route
} from 'react-router-dom';

const App = () => (
  <BrowserRouter>
    <Routes>
      <Route
        path="/users"
        element={<UserList />}
      >
        <Route
          path=":id"
          element={<User />}
        />
      </Route>
    </Routes>
  </BrowserRouter>
)
```

```
// V6 example

import {
  Link
  Outlet
} from 'react-router-dom';

function UserList({ users }) {
  const outlet = <Outlet />

  const list = <ul>{
    users.map({ id } => (
      <Link key={id} to={id}>
        User: {id}
      </Link>
    ))
  }</ul>

  return outlet || list
}
```

```
// V6 example

import {
  useNavigate, useParams
} from 'react-router-dom';

function User() {
  const navigate = useNavigate()
  const { id } = useParams()

  const back = () => navigate(-1)

  return (
    <>
      <h1>Display user: {id}</h1>
      <Button onClick={back}>
        Back
      </Button>
    </>
  )
}
```

React & TypeScript

Les types de React

- **`JSX.Element`** ou **`React.ReactElement`**

Représente le retour de `React.createElement`

- **`React.ReactNode`** ❤️

Tous les types valides que peut renvoyer un composant react. C'est équivalent à:

```
React.ReactChild | React.ReactFragment | React.ReactPortal | boolean | null | undefined
```

- **`React.ReactChild`**

Toutes les valeur valide pour la propriété children. C'est équivalent à:

```
JSX.Element | string | number
```

Pour plus de détails lisez:

- React TypeScript Cheatsheet
- La documentation JSX de TypeScript

Typer les props (et les states)

```
npm i -D @types/react
```

```tsx
import { useState } from 'react'

interface MyProps {
  id: number
  name: string
  // ...
}

function MyComponent({ id, name }: MyProps) {
  const [data, setData] = useState(
    null as string | null
  )

  return (
    <>
      <div>id: {id}</div>
      <div>name: {name}</div>
      <div>data: {data}</div>
    </>
  )
}
```

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

interface MyProps {
  id: number
  name: string
  // ...
}

interface MyState {
  data: string | null
}

class MyComponent extends Component<MyProps, MyState> {
  state = { data: null }

  render() {
    return (
      <>
        <div>id: {this.props.id}</div>
        <div>name: {this.props.name}</div>
        <div>data: {this.state.data}</div>
      </>
    )
  }
}
```

Stockage d’état léger

Stockage d’état léger

React Context

Le Contexte React

Le Contexte est conçu pour partager des données qui peuvent être considérées comme « globales » pour une arborescence de composants React.

L'utilisation des contextes se déroule en trois étapes :

1. La création d'un objet context avec **`React.createContext`**
2. Le partage de l'état du context avec le **`Provider`** dédié
3. La récupération de l'état du context avec le hook **`useContext`**

```
import { createContext } from 'react'

// Créer un context avec une valeur par défaut
const ThemeContext = createContext(`dark`)

// Partage l'état du context avec une arborescence de composant
function App({ theme }) {
  const { Provider: ThemeProvider } = ThemeContext

  return (
    <ThemeProvider value={theme}>
      <MyComponent />
    </ThemeProvider>
  )
}

// Récupère l'état courant du context
function MyComponent() {
  const theme = useContext(ThemeContext)

  return (
    <span className={theme}>
      Use the theme: {theme}
    </span>
  )
}
```

Contexte et stockage d'état

On peut utiliser un Context en conjonction avec le hook **`useReducer`** pour créer un sytème de stockage d'état partagé.

```
import { createContext, useContext, useMemo, useReducer } from 'react'
import myReducer, { initialState } from './myReducer'

const StoreContext = createContext({})

// Get Store context content from anywhere
export default function useStoreContext() {
  return useContext(StoreContext)
}

// Provide the Store context to the given subtree
export function StoreContextProvider({ children }) {
  const [state, dispatch] = useReducer(myReducer, initialState)

  const ctx = useMemo(() => (
    { state, dispatch }
  ), [state])

  return (
    <StoreContext.Provider value={ctx}>
      {children}
    </StoreContext.Provider>
  );
}
```

Lire et modifier l'état

```
export function useFruit(fruit) {
  const { state, dispatch } = useStoreContext()

  // Pour des sélecteurs complexes
  // on peut utiliser le module reselect
  const total = state[fruit] ?? 0

  // Pour les actions complexe ou intelligente, vous
  // pouvez créez des fonctions de création d'action
  const eat = useCallback(() => dispatch({
    type: fruit,
    payload: total - 1
  }), [total])

  const buy = useCallback(() => dispatch({
    type: fruit,
    payload: total + 1
  }), [total])

  return [total, buy, eat]
}
```

```
function MyFruit({ fruit }) {
  const [nbr, add, remove] = useFruit(fruit)

  return (
    <p>
      I have {nbr} {fruit}(s)
      <button onClick={remove}>Eat one</button>
      <button onClick={add}>Buy one</button>
    </p>
  )
}

function MyFruitSalad() {
  const fruits = [`apple`, `banana`]

  return fruits.map((fruit) => (
    <MyFruit key={fruit} fruit={fruit} />
  ))
}

function App() {
  return (
    <StoreContextProvider>
      <MyFruitSalad />
    </StoreContextProvider>
  )
}
```

Stockage d’état léger

Zustand

La bibliothèque Zustand

« Une petite solution de gestion d'état, rapide et évolutive, utilisant des principes de flux simplifiés. Elle dispose d'une API simple basée sur les hooks, ni rigide ni dogmatique. »

Documentation :

- Démo en ligne
- Dépôt Github

Les **+** notables par rapport aux contextes React

- Pas de context Provider
- Possibilité de faire du rendering atomic 🤩
- Système de middleware 🤔

```
import create from 'zustand'

const useStore = create(set => ({
  count: 1,
  inc: () => set(state => ({
    count: state.count + 1
  })),
}))

function Controls() {
  const inc = useStore(state => state.inc)
  return <button onClick={inc}>one up</button>
}

function Counter() {
  const count = useStore(state => state.count)
  return <h1>{count}</h1>
}
```

Example d’usage

Zustand melange les données et les fonctions de mise à jour dans l’état.

```
import create from 'zustand'

const useStore = create((set) => ({
  apple: 0,
  banana: 0,

  eat: (fruit) => set((state) => ({
    [fruit]: Math.max(0, state[fruit] - 1)
  })),

  buy: (fruit) => set((state) => ({
    [fruit]: state[fruit] + 1
  }))
}))
```

```
function MyFruit({ fruit }) {
  const nbr = useStore((state) => state[fruit])
  const add = useStore((state) => state.buy)
    .bind(null, fruit)
  const remove = useStore((state) => state.eat)
    .bind(null, fruit)

  return (
    <p>
      I have {nbr} {fruit}(s)
      <button onClick={remove}>Eat one</button>
      <button onClick={add}>Buy one</button>
    </p>
  )
}

function MyFruitSalad() {
  const fruits = [`apple`, `banana`]

  return fruits.map((fruit) => (
    <MyFruit key={fruit} fruit={fruit} />
  ))
}
```

Example d’usage

Zustand est suffisamment souple pour s’adapter aux habitudes/codes qu’on peut déjà avoir.

```
import create from 'zustand'

const useStore = create((set) => ({
  state: {
    apple: 0,
    banana: 0,
  }

  dispatch: (action) => set(
    ({ state }) => ({
      state: myReducer(state, action)
    })
  )
}))
```

```
export function useFruit(fruit) {
  // Un seul changement pour passer de context à zustand
  const { state, dispatch } = useStore()

  // Pour des sélecteurs complexes
  // on peut utiliser le module reselect
  const total = state[fruit] ?? 0

  // Pour les actions complexe ou intelligente, vous
  // pouvez créez des fonctions de création d'action
  const eat = useCallback(() => dispatch({
    type: fruit,
    payload: total - 1
  }), [total])

  const buy = useCallback(() => dispatch({
    type: fruit,
    payload: total + 1
  }), [total])

  return [total, buy, eat]
}
```

Gestion des formulaires

Les formulaires avec React

De manière général React rationalise l'interface des éléments de formulaire en normalisant les évènements qu'ils émettent.

Pour contrôler et accéder au valeur des différents composant des formulaires il existe deux grande méthodes:

1. Les composants controlés
2. Les composants non controlés

```
// Composants controlés
function ControledForm() {
  const [name, setName] = useState(``)
  const change = (evt) => setName(
    evt.target.value
  )
  const submit = (evt) => {
    evt.preventDefault()
    alert(`Send name: ${name}`)
  }

  return (
    <form onSubmit={submit}>
      <label htmlFor="name">
        Name: <input
          id="name"
          onChange={change}
          value={name}
        />
      </label>
      <button>Send</button>
    </form>
  )
}
```

```
// Composant non controlés
function UncontroledForm() {
  const name = React.createRef()

  const submit = (evt) => {
    evt.preventDefault()
    alert(`Send name: ${
      name.current.value()
    }`)
  }

  return (
    <form onSubmit={submit}>
      <label htmlFor="name">
        Name:
        <input
          id="name"
          ref={name}
        />
      </label>
      <button>Send</button>
    </form>
  )
}
```

Les bibliothèques de gestion de formulaire

Il existent beaucoup de bibliothèques de gestion des formulaires pour React. Elles ont pour objectif de simplifier la validation des données et tout le cycle de vie de gestion des erreurs de saisie.

Trois d'entre elles sortent du lot :

- Formik
- React Hook Form
- React Final Form

A voir sur NPM Trends.

Gestion des formulaires

React Hook Form

React Hook Form

Jeune bibliothèque en pleine ascension, elle utilise toute les capacité des Hooks React.

C’est principale caractéristique

- Une API compact et versatile
- Capacité d’observer les changements de valeur
- Pensé pour optimiser les performances

Utilisation de base

useForm est au coeur du fonctionnement de la bibliothèque

```
import { useForm } from "react-hook-form";

function MyForm() {
  const { register, handleSubmit, formState: { errors } } = useForm()
  const mySubmit = (data) => console.log(data)

  return (
    <form onSubmit={handleSubmit(mySubmit)}>
      <label htmlFor="apple">
        Apples: <input
          id="apple" type="number" step="1"
          {...register(`apple`, { required: true, min: 0 })}
          aria-invalid={errors.apple ? "true" : "false"}
        />
        {errors.apple && (
          <span role="alert">This field is required</span>
        )}
      </label>
      <button>Confirm</button>
    </form>
  )
}
```

Smart component

React Hook From utilise les contextes React pour permettre de construire des composants de formulaire réutilisables

```
function MyForm() {
  const defaultProps = {
    type="number",
    min={ value: 0, message: `This field can't be negative.` }
    required="This field is required."
  }
  const defaultValues = { apple: 0, banana: 0 }
  const onSubmit = (data) => console.log(data)

  return (
    <Form {...{ defaultValues, onSubmit}}>
      <Input label="Apple:" name="apple"  {...defaultProps} />
      <Input label="Banana:"name="banana" {...defaultProps} />
    </Form>
  )
}
```

```
function Form({ defaultValues, children, onSubmit }) {
  const methods = useForm({ defaultValues })
  return (
    <FormProvider {...methods}>
      <form onSubmit={methods.handleSubmit(onSubmit)}>
        {children}
        <button>Submit</button>
      </form>
    </FormProvider>
  )
}

function Input({ name, type, label …options }) {
  const { register, formState: { errors } } = useFormContext()
  return (
    <label htmlFor={name}>
      {label}
      <input id={name} type={type}
        {...register(name, options)}
        aria-invalid={errors[name] ? "true" : "false"}
      />
      {errors[name] && (
        <span role="alert">{errors[name]}</span>
      )}
    </label>
  )
}
```