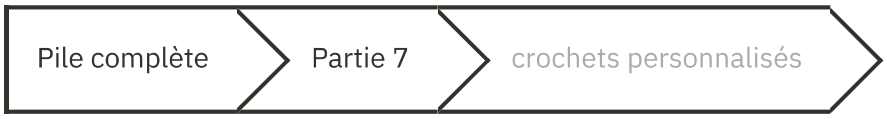
{() => fs}



b crochets personnalisés

Les exercices de la septième partie du cours diffèrent quelque peu des parties précédentes. Comme d'habitude, le chapitre précédent et celui-ci contiennent des exercices liés à la théorie des nombres .

En plus des tâches de ce chapitre et du chapitre suivant, la septième partie comprend une série de tâches de révision et d'application qui étendent l'application de liste de blogs créée dans les parties 4 et 5.

Crochets

React propose un total de 15 hooks différents prêts à l'emploi , dont les plus utilisés sont de loin les déjà familiers useState et useEffect .

Dans la partie 5, nous avons utilisé le hook useImperativeHandle pour exposer le fonctionnement interne d'un composant au monde extérieur. Dans la partie 6 , nous avons utilisé useReducer et useContext pour implémenter une solution de gestion d'état de type Redux.

Ces dernières années, de nombreuses bibliothèques React ont commencé à proposer une interface basée sur des hooks. Dans la partie 6, nous avons utilisé les hooks useSelector et useDispatch de la bibliothèque React Redux pour transmettre les fonctions de stockage et de répartition Redux aux composants qui en ont besoin.

L'API React Router évoquée dans le chapitre précédent repose également en partie sur des hooks , qui permettent d'accéder à la partie paramétrée des routes, et sur *l'* objet de navigation, qui permet de manipuler la barre d'adresse du navigateur à partir du code.

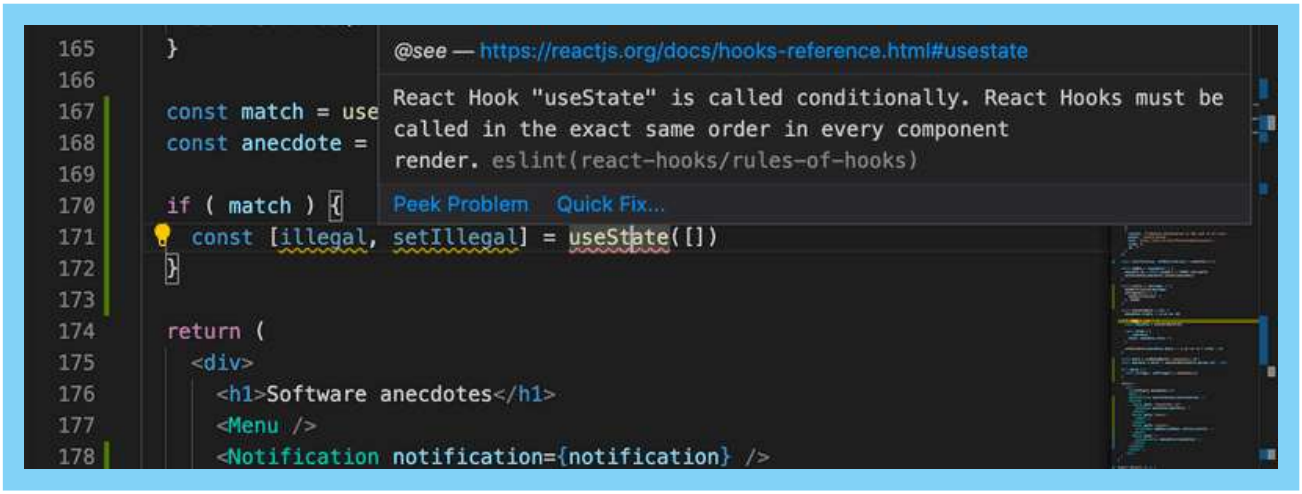
Comme mentionné dans la partie 1 , les hooks ne sont pas n'importe quelles fonctions, mais doivent être utilisés selon certaines règles . Voici les règles d'utilisation des hooks, directement tirées de la documentation React :

N'appelez pas de hooks dans des boucles, des conditions ou des fonctions imbriquées. Utilisez plutôt les hooks au niveau supérieur de votre fonction React.

N'appelez pas de hooks à partir de fonctions JavaScript classiques. Vous pouvez plutôt :

- Appelez les hooks à partir des composants de fonction React.
- Appeler des hooks à partir de hooks personnalisés

Il existe une règle ESLint permettant de garantir que votre application utilise correctement les hooks. Le réglage eslint-plugin-react-hooks , préinstallé dans Create React App , vous avertira si vous essayez d'utiliser un hook de manière incorrecte :



Crochets personnalisés

React offre également la possibilité de définir vos propres hooks, ou des hooks personnalisés . Selon la documentation React, l'objectif principal des hooks personnalisés est de permettre la réutilisation de la logique des composants :

> *Créer vos propres Hooks vous permet d'extraire la logique des composants dans des fonctions réutilisables.*

Les hooks personnalisés sont des fonctions JavaScript standard qui peuvent appeler n'importe quel autre hook, à condition qu'ils respectent les règles des hooks. Le nom d'un hook personnalisé doit commencer par le mot `« use »` .

Dans la partie 1 , nous avons créé un compteur incrémentable, décrémentable et réinitialisable. Le code de l'application est le suivant :

```
import { useState } from 'react'
const App = () => {
  const [counter, setCounter] = useState(0)

  return (
    <div>
      <div>{counter}</div>
      <button onClick={() => setCounter(counter + 1)}>
        plus
```

```
      </button>
      <button onClick={() => setCounter(counter - 1)}>
        minus
      </button>
      <button onClick={() => setCounter(0)}>
        zero
      </button>
    </div>
  )
}
```

Séparons la logique du compteur en un hook personnalisé. Le code de ce hook est le suivant :

copie

```
const useCounter = () => {
  const [value, setValue] = useState(0)

  const increase = () => {
    setValue(value + 1)
  }

  const decrease = () => {
    setValue(value - 1)
  }

  const zero = () => {
    setValue(0)
  }

  return {
    value,
    increase,
    decrease,
    zero
  }
}
```

Le hook utilise donc en interne `le` hook useState pour créer son propre état. Il renvoie un objet contenant la valeur de son état sous forme de champs et de fonctions permettant de modifier la valeur stockée par le hook.

Le composant React utilise le hook de la manière suivante :

copie

```
const App = () => {
  const counter = useCounter()

  return (
    <div>
      <div>{counter.value}</div>
      <button onClick={counter.increase}>
        plus
      </button>
      <button onClick={counter.decrease}>
        minus
      </button>
```

```
      <button onClick={counter.zero}>
        zero
      </button>
    </div>
  )
}
```

De cette façon, l'état du composant `App` et sa manipulation ont été entièrement transférés au hook `useCounter` .

Le même hook pourrait être *réutilisé* dans une application qui compterait les pressions sur les boutons gauche et droit :

copie

```
const App = () => {
  const left = useCounter()
  const right = useCounter()

  return (
    <div>
      {left.value}
      <button onClick={left.increase}>
        left
      </button>
      <button onClick={right.increase}>
        right
      </button>
      {right.value}
    </div>
  )
}
```

L'application crée maintenant *deux* compteurs distincts. L'un, avec ses fonctions de traitement, est stocké dans la variable `left` et l'autre dans la variable `right` .

La gestion des formulaires est un peu complexe dans React. Voici une application qui demande le nom, la date de naissance et la taille de l'utilisateur dans un formulaire :

copie

```
const App = () => {
  const [name, setName] = useState('')
  const [born, setBorn] = useState('')
  const [height, setHeight] = useState('')

  return (
    <div>
      <form>
        name:
        <input
          type='text'
          value={name}
          onChange={(event) => setName(event.target.value)}
        />
        <br/>
        birthdate:
```

```
        <input
          type='date'
          value={born}
          onChange={(event) => setBorn(event.target.value)}
        />
        <br />
        height:
        <input
          type='number'
          value={height}
          onChange={(event) => setHeight(event.target.value)}
        />
      </form>
      <div>
        {name} {born} {height}
      </div>
    </div>
  )
}
```

Chaque champ de formulaire possède son propre état. Pour maintenir cet état à jour avec les données saisies dans le formulaire, un gestionnaire *onChange* approprié est enregistré pour chaque élément *de saisie* .

Définissons un hook personnalisé `useField` qui simplifie la gestion de l'état du formulaire :

```
const useField = (type) => {
  const [value, setValue] = useState('')

  const onChange = (event) => {
    setValue(event.target.value)
  }

  return {
    type,
    value,
    onChange
  }
}
```

La fonction hook prend le type du champ comme paramètre. Elle renvoie tous les attributs requis par *le* champ de saisie, à savoir le type, la valeur du champ et le gestionnaire d'événements onChange.

Le crochet peut être utilisé comme suit :

```
const App = () => {
  const name = useField('text')
  // ...

  return (
    <div>
      <form>
        <input
```

```
          type={name.type}
          value={name.value}
          onChange={name.onChange}
        />
        // ...
      </form>
    </div>
  )
}
```

Attributs de propagation

Nous disposons en fait d'une méthode plus simple. Puisque l' objet `name` contient désormais exactement les champs que *le* composant input attend comme propriétés, nous pouvons transmettre ces propriétés en utilisant la syntaxe spread comme suit :

```
<input {...name} />
```

copie

Ainsi, comme le montre l'exemple dans la documentation React , les deux manières suivantes de transmettre des accessoires à un composant produisent le même résultat :

```
<Greeting firstName='Arto' lastName='Hellas' />
```

copie

```
const person = {
  firstName: 'Arto',
  lastName: 'Hellas'
}

<Greeting {...person} />
```

L'application se réduit à la forme :

```
const App = () => {
  const name = useField('text')
  const born = useField('date')
  const height = useField('number')

  return (
    <div>
      <form>
        name:
        <input  {...name} />
        <br/>
        birthdate:
        <input {...born} />
        <br />
        height:
        <input {...height} />
      </form>
```

copie

```
      <div>
        {name.value} {born.value} {height.value}
      </div>
    </div>
  )
}
```

Le traitement des formulaires est grandement simplifié lorsque les détails fastidieux liés à la synchronisation des états sont encapsulés dans un hook personnalisé.

Les hooks personnalisés ne sont clairement pas seulement un outil de réutilisation, mais ils permettent également une meilleure façon de diviser le code en parties plus petites et modulaires.

On trouve de plus en plus de crochets prêts à l'emploi et d'autres supports utiles sur Internet. Voici quelques exemples :

- Ressources impressionnantes sur les hooks React
- Recettes React Hook faciles à comprendre par Gabe Ragland
- Pourquoi les hooks React s'appuient-ils sur l'ordre d'appel ?

Tâches 7.4.-7.8.

7.4 : application anecdote et accroches étape 1

Continuons avec l' application de tâche React Router .

Simplifiez l'utilisation du formulaire utilisé pour créer une nouvelle anecdote dans votre application avec `le` hook personnalisé useField que vous venez de définir.

Enregistrez le hook dans le fichier */src/hooks/index.js* .

Si vous utilisez une exportation nommée au lieu de l'exportation par défaut que nous utilisons normalement

copie

```
import { useState } from 'react'

export const useField = (type) => {
  const [value, setValue] = useState('')

  const onChange = (event) => {
    setValue(event.target.value)
  }

  return {
    type,
    value,
    onChange
  }
}
```

```
// moduulissa voi olla monta nimettyä eksportia
export const useAnotherHook = () => {
  // ...
}
```

L'importation se fait comme suit :

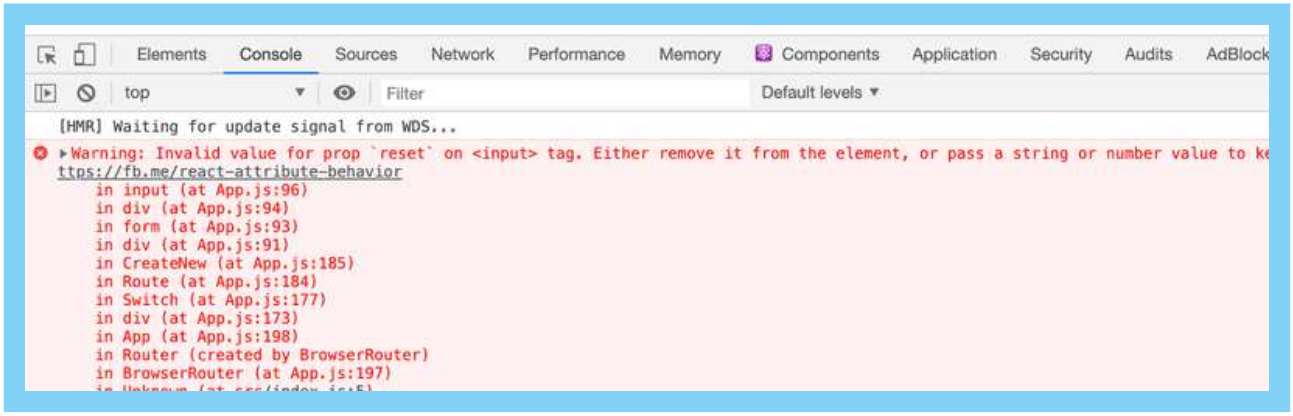
```
import  { useField } from './hooks'

const App = () => {
  // ...
  const username = useField('text')
  // ...
}
```

7.5 : application anecdote et accroches étape 2

Ajoutez un bouton au formulaire qui vous permet d'effacer les champs de saisie :

Étendez le hook afin qu'il fournisse une opération appelée *reset* pour effacer le champ.

Après l'ajout, un avertissement gênant peut apparaître dans la console :



Ne vous inquiétez pas encore de l'erreur dans cette tâche.

7.6 : application anecdote et accroches étape 3

Si votre solution n'a pas provoqué d'avertissement, vous n'avez rien à faire dans cette tâche.

Sinon, apportez un correctif à votre application qui supprime l'avertissement `Valeur non valide pour la propriété 'reset' sur la balise <input> .`

La raison de l'avertissement est qu'après que la tâche précédente a été étendue, la tâche suivante

```
<input {...content}/>
```

copie

signifie la même chose que

```
<input
  value={content.value}
  type={content.type}
  onChange={content.onChange}
  reset={content.reset}
/>
```

copie

Cependant, l' élément *d'entrée* ne doit pas recevoir de *réinitialisation d'accessoire.*

Une solution simple serait bien sûr de ne pas utiliser la syntaxe de propagation et d'écrire toutes les formes comme suit :

```
<input
  value={username.value}
  type={username.type}
  onChange={username.onChange}
/>
```

copie

Dans ce cas, nous perdrions la plupart des avantages du hook *useField* . J'ai néanmoins trouvé une solution simple pour cette tâche, utilisant la syntaxe de propagation pour contourner le problème.

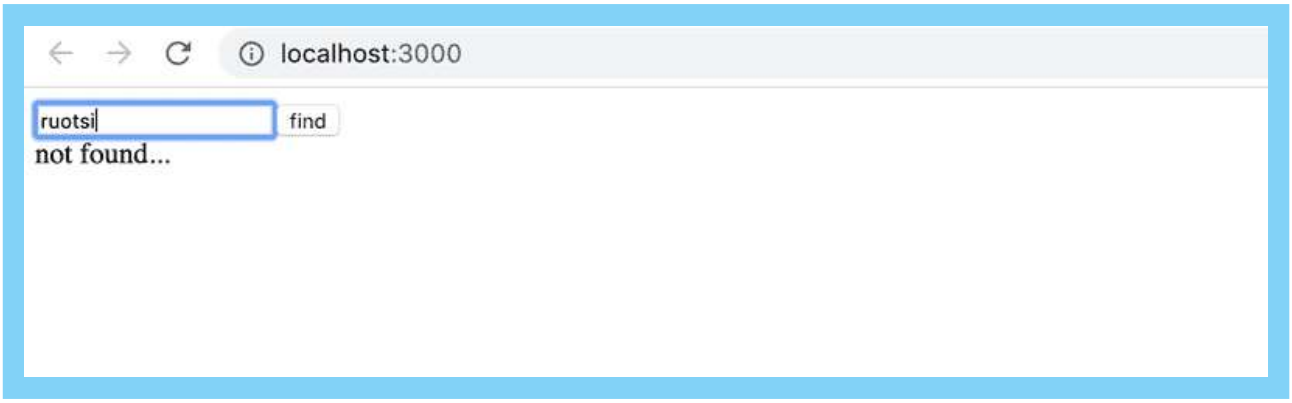
7.7 : crochet de campagne

Revenons un instant à l'atmosphère de la série de tâches 2.12-14 .

Utilisez le code du référentiel https://github.com/fullstack-hy2020/country-hook comme base .

L'application vous permet de rechercher des informations sur un pays depuis l'interface https://studies.cs.helsinki.fi/restcountries/ . Si le pays est trouvé, ses informations de base s'affichent :

Si le pays n'est pas trouvé, l'utilisateur est informé :



L'application est déjà implémentée, mais dans cette tâche, vous devrez implémenter un hook personnalisé, `useCountry` , qui récupérera les informations de pays reçues en paramètre par le hook.

Il est recommandé de récupérer les informations sur le pays à l'aide du nom du point de terminaison de l'API et `du hook useEffect` à l'intérieur du hook .

Notez que dans cette tâche, il est essentiel d'utiliser le tableau comme second paramètre de useEffect . Ce tableau permet de contrôler le moment opportun pour exécuter la fonction effect. Dans la deuxième partie du cours , les principes d'utilisation du second paramètre ont été revus.

7.8 : crochets ultimes

Le code qui communique avec le serveur de l'application de prise de notes développée dans le matériel des sections précédentes ressemble à ceci :

```
import axios from 'axios'
const baseUrl = '/api/notes'

let token = null

const setToken = newToken => {
  token = `bearer ${newToken}`
}
```

```
const getAll = () => {
  const request = axios.get(baseUrl)
  return request.then(response => response.data)
}

const create = async newObject => {
  const config = {
    headers: { Authorization: token },
  }

  const response = await axios.post(baseUrl, newObject, config)
  return response.data
}

const update = (id, newObject) => {
  const request = axios.put(`${baseUrl}/${id}`, newObject)
  return request.then(response => response.data)
}

export default { getAll, create, update, setToken }
```

Nous constatons que le code ne se soucie pas de gérer spécifiquement les notes. Outre la valeur `de la variable baseUrl` , en pratique, le même code peut également gérer la communication entre le frontend et le backend de l'application de blog.

Isolez le code de communication dans un hook `useResource` . Il suffit que la récupération de tous les objets et la création d'un nouvel objet réussissent.

Vous pouvez effectuer cette tâche dans le projet, dans le dépôt https://github.com/fullstack-hy2020/ultimate-hooks . Le composant « *App* » du projet est le suivant :

copie

```
const App = () => {
  const content = useField('text')
  const name = useField('text')
  const number = useField('text')

  const [notes, noteService] = useResource('http://localhost:3005/notes')
  const [persons, personService] = useResource('http://localhost:3005/persons')

  const handleNoteSubmit = (event) => {
    event.preventDefault()
    noteService.create({ content: content.value })
  }

  const handlePersonSubmit = (event) => {
    event.preventDefault()
    personService.create({ name: name.value, number: number.value})
  }

  return (
    <div>
      <h2>notes</h2>
      <form onSubmit={handleNoteSubmit}>
        <input {...content} />
        <button>create</button>
```

```
        </form>
        {notes.map(n => <p key={n.id}>{n.content}</p>)}

        <h2>persons</h2>
        <form onSubmit={handlePersonSubmit}>
          name <input {...name} /> <br/>
          number <input {...number} />
          <button>create</button>
        </form>
        {persons.map(n => <p key={n.id}>{n.name} {n.number}</p>)}
      </div>
    )
}
```

Le hook personnalisé `useResource` renvoie (comme les hooks d'état) un tableau à deux éléments. Le premier élément contient tous les objets de la ressource, et le second est un objet permettant de manipuler la ressource, par exemple en ajoutant de nouveaux objets.

Si vous avez implémenté le hook correctement, l'application vous permettra de traiter les notes et les numéros de téléphone simultanément (démarrez le backend sur le port 3005 avec la commande `npm run server`):

Proposer une modification du contenu du matériel

Partie 7a
Partie précédente

Partie 7c
Prochaine partie

À propos du cours

Contenu du cours

FAQ

Inclus dans le cours

Défi

UNIVERSITY OF HELSINKI

HOUSTON