ALESSANDRO GIANNINI
Sviluppatore Software

React: pratiche per scrivere un'applicazione "moderna"

Develer webinar

21/10/2020

develer

CHI SIAMO

Develer un’Azienda che che progetta e realizza soluzioni hardware e software in ambiti industriali innovativi.

COSA FACCIAMO

- Sviluppo software
- Sistemi embedded
- Corsi
- Eventi

Hai bisogno dei nostri servizi?

www.develer.com/servizi

Vuoi lavorare con noi?

www.develer.com/lavora-con-noi

OVERVIEW

- Brevissima introduzione a React
- Versioni di React disponibili e future
- Feature di React 16 che vedremo
- Spiegazione feature ed esempi di utilizzo
- Repo: https://github.com/ag7/webinar-react-2020
- Q&A

REACT FRAMEWORK

- React è una libreria Javascript per creare interfacce utente
- L'interfaccia grafica viene definita per mezzo di componenti
- Il framework ruota intorno a pochi concetti chiave (componenti, state, props)

REACT VERSIONS

- React 17 è stato rilasciato ieri (20 ottobre 2020)
- Non ci sono nuove feature (previste invece per React 18), ma sono stati introdotti i “gradual upgrades”
- Noi ci focalizziamo su React 16
- Parleremo di un subset di feature introdotte da metà 2017 in poi

REACT 16

- React Hooks
- Fragments
- Lazy Loading
- Portals
- Context API
- Error Boundaries
- React Profiler API

REACT COMPONENTS

- I componenti sono i “mattoncini” con i quali viene costruita la UI
- Sono descritti (UI) per mezzo di sintassi JSX
- Il rendering può essere parametrizzato in base a uno stato interno (state) o esterno (props) al componente

RENDERING

- Il rendering di un componente è eseguito..
- Al variare dello stato interno (state)
- Al variare di una proprietà esterna (props)
- Al variare del valore di un contesto (in caso di sottoscrizione a un contesto)

COMPONENT TYPES

- Functional components
- Class-based components
- “Pure” components

Esempio - “Classic” counter

```
export default class Counter extends Component {
    constructor(props) {
        super(props)
        this.state = { counter: 0 }
        this.onCounterClick = this.onCounterClick.bind(this)
    }

    onCounterClick() {
        this.setState(function (state) {
            return { counter: state.counter +1 }
        })
    }

    render() {
        return (
            <div onClick={this.onCounterClick}>{this.state.counter}</div>
        )
    }
}
```

- Un semplice counter con incremento al click di un div
- Dobbiamo usare l’update asincrono dello stato se esiste una dipendenza tra stato attuale e desiderato
- Possiamo semplificare..

Esempio - “Classic” counter simplified

```
export default class Counter extends Component {
    constructor(props) {
        super(props)
        this.state = { counter: 0 }
    }

    onCounterClick = () => {
        this.setState((state) => ({ counter: state.counter +1 }))
    }

    render() {
        return (
            <div onClick={this.onCounterClick}>{this.state.counter}</div>
        )
    }
}
```

- Possiamo omettere i “bind” utilizzando le arrow function per definire i metodi pubblici della classe
- Possiamo utilizzare le arrow function per “snellire” la setState di incremento

ERROR BOUNDARIES

- I “contenitori di errori” permettono di gestire in modo graceful errori (eccezioni) in fase di rendering
- Intercettano le eccezioni lanciate dal sotto-albero del Virtual DOM a cui sono applicati
- Permettono di mostrare una UI alternativa e gestire la recovery (manualmente)

Esempio - “Bugged” counter

```
export default class Counter extends Component {
    constructor(props) {
        super(props)
        this.state = { counter: 0 }
    }

    onCounterClick = () => {
        this.setState((state) => ({ counter: state.counter +1 }))
    }

    render() {
        const { counter } = this.state
        if (counter === 3) throw new Error('Crashed!')
        return (
            <div onClick={this.onCounterClick}>this.state.counter}</div>
        )
    }
}
```

- Implementiamo un contatore “buggato”
- Scriviamo adesso un boundary per gestire l’errore..

Esempio - Counter Boundary

```
export default class CounterBoundary extends Component {
   constructor(props) {
       super(props)
       this.state = { hasError: false }
   }

   componentDidCatch(error, info) {
       this.setState({ hasError: true })
       logToExternalService(error)
       console.info(info.componentStack)
   }

   render() {
       if (this.state.hasError) return <div>Cannot render counter</div>
       return this.props.children
   }
}
```

- “componentDidCatch” intercetta le eccezioni del sotto-albero a cui è applicato “CounterBoundary”
- Possiamo utilizzare “CounterBoundary” come qualsiasi altro componente React..

Esempio - Utilizzo di un error boundary

```
...
render() {
    <CounterBoundary>
        <Counter />
    </CounterBoundary>
}
...
```

REACT HOOKS

- Sono stati introdotti in React 16.8
- Permettono di “agganciare” alcuni internals di React (state, effects, ..) da componenti funzionali
- Permettono di scrivere codice più pulito rispetto all’equivalente “class-based”
- Vedremo l’hook “useState”

Esempio - “Bugged” counter with “useState” hook

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

export default function BuggedCounter() {
   const [counter, setCounter] = useState(0)
   if (counter === 3) throw new Error('Crashed!')
   return <div onClick={() => setCounter(counter + 1)}>{counter}</div>
}
```

- Usiamo lo stato React per mezzo di “useState” da un componente funzionale
- “useState” riceve il valore iniziale dello stato e fornisce lo stato corrente e un setter
- Il codice finale è molto più semplice dell’equivalente class-based

USE STATE HOOK

- Posso avere più chiamate a “useState” nello stesso componente funzionale
- Lo stato da passare al setter deve essere completo (a differenza di “setState”, non viene mergiato col precedente)
- Per mezzo di hook personalizzati posso estrarre logica stateful e riusarla in più componenti

ALTRI HOOKS

- “useEffect” - combina vari lifecycle methods in una funzione unica, richiamata ad ogni rendering
- “useContext” - “aggancia” il contesto specificato e fornisce il valore corrente
- “useReducer” - implementa il reducer pattern per la gestione dello stato
- … (infinite possibilità con gli hook personalizzati)

FRAGMENTS

- Consentono il raggruppamento di nodi figli senza un nodo contenitore nel DOM
- Evitano il proliferare di nodi inutili nel DOM

Esempio - Esempio di React Fragments

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

export default function CounterList() {
   const [counter, setCounter] = useState(0)
   if (counter === 3) throw new Error('Crashed!')
   return (
       <>
           <div>Do not click thrice on this counter!!</div>
           <div onClick={() => setCounter(counter + 1)}>{counter}</div>
       </>
   )
}
```

- L'uso di "<>" e "</>" identifica un React fragment
- Si può usare anche il tag "<React.Fragment>" per essere più verbosi
- La lista di children verrà creata nel DOM "saltando" il nodo fragment

CONTEXT API

- Consente di definire un contesto (di fatto, uno stato) diverso da state e da props
- Per mezzo di un provider, il contesto viene iniettato a un sotto-albero di componenti
- Si evita così il passaggio manuale di props nella gerarchia di componenti, ottenendo un codice più pulito

Esempio - Creazione di un contesto

```
import {createContext} from 'react';

export const themes = {
 light: {panelBackground:'#DDDDDD', textForeground:'#555555'},
 dark: {panelBackground:'#555555', textForeground:'#DDDDDD'}
}

export const ThemeContext =
   createContext({name: 'dark', theme: themes.dark,toggleTheme: () => {}})
```

- Un contesto viene creato con un valore di default per mezzo di “createContext”
- Il valore di default deve essere istanziato all’atto della definizione del provider
- Il contesto creato è un componente React

Esempio - Applicazione di un contesto

```
import React, { useState } from 'react';
import RootComponent from './RootComponent'
import { ThemeContext, themes } from './ThemeContext'

export default function App() {
 const [themeName, setThemeName] = useState('dark');
 const toggleTheme = () => setThemeName(themeName === 'light' ? 'dark' : 'light')

 return (
   <ThemeContext.Provider value={{ name: themeName, theme: themes[themeName], toggleTheme }}>
     <RootComponent />
   </ThemeContext.Provider>
 );
}
```

Esempio - Utilizzo del valore di un contesto da un componente funzionale

```
import React, { useContext } from 'react';
import { ThemeContext } from './ThemeContext'

export default function Todo(props) {
   const { text, onClick } = props;
   const { theme } = useContext(ThemeContext)

   return (
       <div className="todo-container">
           <div className="todo-element" style={{ color: theme.textForeground }}>{text}</ div>
           <input type="submit" value="X" onClick={onClick} />
       </div>
   )
}
```

Esempio - Utilizzo del valore di un contesto da un componente class-based

```
import React, { Component } from 'react';
import { ThemeContext } from './ThemeContext'

export default class TodoList extends Component {
    ...

    render() {
        // the context current value is accessible with `this.context`
        const { theme: { panelBackground, textForeground } } = this.context;
        return ...
    }
}

// the context type must be declared in `contextType` class variable
TodoList.contextType = ThemeContext;
```

TODO LIST REPOSITORY

- “In allegato” al webinar un repository che implementa una semplice todo list
- Il codice dell’app utilizza tutti i concetti visti nel webinar
- Può essere utile, in aggiunta alle slide, come reference per un codice (minimale) funzionante

Q&A

Alessandro Giannini

alessandro@develer.com

Vuoi rimanere aggiornato sugli eventi Develer?
Seguici nei nostri canali social:

develer

www.develer.com