EBook Gratuito

APPRENDIMENTO React

Free unaffiliated eBook created from
Stack Overflow contributors.

#reactjs

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: react

It is an unofficial and free React ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official React.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con React

Osservazioni

React è una libreria JavaScript dichiarativa, basata su componenti, utilizzata per creare interfacce utente.

Per ottenere il framework MVC come le funzionalità di React, gli sviluppatori lo usano in combinazione con un gusto di scelta di Flux , ad es. Redux .

Versioni

Versione	Data di rilascio
0.3.0	2013/05/29
0.4.0	2013/07/17
0.5.0	2013/10/16
0.8.0	2013/12/19
0.9.0	2014/02/20
0.10.0	2014/03/21
0.11.0	2014/07/17
0.12.0	2014/10/28
0.13.0	2015/03/10
0.14.0	2015/10/07
15.0.0	2016/04/07
15.1.0	2016/05/20
15.2.0	2016/07/01
15.2.1	2016/07/08
15.3.0	2016/07/29
15.3.1	2016/08/19
15.3.2	2016/09/19

Versione	Data di rilascio
15.4.0	2016/11/16
15.4.1	2016/11/23
15.4.2	2017/01/06
15.5.0	2017/04/07
15.6.0	2017/06/13

Examples

Installazione o configurazione

ReactJS è una libreria JavaScript contenuta in un singolo file `react-<version>.js` che può essere inclusa in qualsiasi pagina HTML. Inoltre, le persone installano comunemente la libreria React DOM `react-dom-<version>.js` insieme al file principale di React:

Inclusione di base

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script type="text/javascript">
        // Use react JavaScript code here or in a separate file
    </script>
    </body>
</html>
```

Per ottenere i file JavaScript, vai alla pagina di installazione della documentazione ufficiale di React.

React supporta anche la sintassi JSX . JSX è un'estensione creata da Facebook che aggiunge la sintassi XML a JavaScript. Per utilizzare JSX devi includere la libreria Babel e cambiare `<script type="text/javascript">` in `<script type="text/babel">` per tradurre JSX in codice Javascript.

```
<!DOCTYPE html>
<html>
    <head></head>
    <body>
    <script type="text/javascript" src="/path/to/react.js"></script>
    <script type="text/javascript" src="/path/to/react-dom.js"></script>
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>
    <script type="text/babel">
        // Use react JSX code here or in a separate file
    </script>
    </body>
```

```
</html>
```

Installazione via npm

Puoi anche installare React usando npm attenendo alla seguente procedura:

```
npm install --save react react-dom
```

Per utilizzare React nel tuo progetto JavaScript, puoi fare quanto segue:

```
var React = require('react');
var ReactDOM = require('react-dom');
ReactDOM.render(<App />, ...);
```

Installazione via filato

Facebook ha rilasciato il proprio gestore di pacchetti denominato Yarn , che può essere utilizzato anche per installare React. Dopo aver installato Yarn è sufficiente eseguire questo comando:

```
yarn add react react-dom
```

Puoi quindi utilizzare React nel tuo progetto esattamente nello stesso modo in cui avevi installato React via npm.

Ciao World Component

Un componente React può essere definito come una classe ES6 che estende la classe base `React.Component` . Nella sua forma minima, un componente *deve* definire un metodo `render` che specifica come il componente viene `render` al DOM. Il metodo di `render` restituisce i nodi React, che possono essere definiti usando la sintassi JSX come tag di tipo HTML. L'esempio seguente mostra come definire una componente minima:

```
import React from 'react'

class HelloWorld extends React.Component {
    render() {
        return <h1>Hello, World!</h1>
    }
}

export default HelloWorld
```

Un componente può anche ricevere `props` . Queste sono proprietà passate dal suo genitore al fine di specificare alcuni valori che il componente non può conoscere da solo; una proprietà può anche contenere una funzione che può essere chiamata dal componente dopo che si sono verificati determinati eventi - ad esempio, un pulsante potrebbe ricevere una funzione per la sua proprietà `onClick` e chiamarla ogni volta che viene fatto clic. Durante la scrittura di un componente, è possibile accedere ai suoi `props` attraverso l'oggetto `props` sulla Componente stessa:

```
import React from 'react'
```

```
class Hello extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}!</h1>
    }
}

export default Hello
```

L'esempio sopra mostra come il componente può eseguire il rendering di una stringa arbitraria passata nel `name` prop dal suo genitore. Si noti che un componente non può modificare i puntelli che riceve.

Un componente può essere reso all'interno di qualsiasi altro componente, o direttamente nel DOM se è il componente più in alto, utilizzando `ReactDOM.render` e fornendo sia il componente che il Nodo DOM in cui si desidera rendere l'albero di Reazione:

```
import React from 'react'
import ReactDOM from 'react-dom'
import Hello from './Hello'

ReactDOM.render(<Hello name="Billy James" />, document.getElementById('main'))
```

Ormai sai come creare un componente base e accettare `props` . Facciamo un ulteriore passo avanti e introduciamo lo `state` .

Per amor di demo, facciamo la nostra app Hello World, mostra solo il nome se viene dato un nome completo.

```
import React from 'react'

class Hello extends React.Component {

    constructor(props){

        //Since we are extending the default constructor,
        //handle default activities first.
        super(props);

        //Extract the first-name from the prop
        let firstName = this.props.name.split(" ")[0];

        //In the constructor, feel free to modify the
        //state property on the current context.
        this.state = {
            name: firstName
        }

    } //Look maa, no comma required in JSX based class defs!

    render() {
        return <h1>Hello, {this.state.name}!</h1>
    }
}
```

```
export default Hello
```

Nota: ogni componente può avere il proprio stato o accettare lo stato del genitore come supporto.

Codepen Link to Example.

Ciao mondo

Senza JSX

Ecco un esempio di base che utilizza l'API principale di React per creare un elemento React e l'API DOM React per eseguire il rendering dell'elemento React nel browser.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>

  </head>
  <body>
    <div id="example"></div>

    <script type="text/javascript">

      // create a React element rElement
      var rElement = React.createElement('h1', null, 'Hello, world!');

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Con JSX

Invece di creare un elemento React dalle stringhe, si può usare JSX (un'estensione Javascript creata da Facebook per aggiungere la sintassi XML a JavaScript), che consente di scrivere

```
var rElement = React.createElement('h1', null, 'Hello, world!');
```

come equivalente (e più facile da leggere per qualcuno che abbia familiarità con HTML)

```
var rElement = <h1>Hello, world!</h1>;
```

Il codice che contiene JSX deve essere racchiuso in un `<script type="text/babel">` . Tutto all'interno di questo tag verrà trasformato in semplice Javascript utilizzando la libreria Babel (che deve essere inclusa in aggiunta alle librerie React).

Quindi, finalmente l'esempio sopra diventa:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>

    <!-- Include the React and ReactDOM libraries -->
    <script src="https://fb.me/react-15.2.1.js"></script>
    <script src="https://fb.me/react-dom-15.2.1.js"></script>
    <!-- Include the Babel library -->
    <script src="https://npmcdn.com/babel-core@5.8.38/browser.min.js"></script>

  </head>
  <body>
   <div id="example"></div>

    <script type="text/babel">

      // create a React element rElement using JSX
      var rElement = <h1>Hello, world!</h1>;

      // dElement is a DOM container
      var dElement = document.getElementById('example');

      // render the React element in the DOM container
      ReactDOM.render(rElement, dElement);

    </script>

  </body>
</html>
```

Cos'è ReactJS?

ReactJS è una libreria front-end basata su componenti open source responsabile solo del **livello** di **vista** dell'applicazione. È gestito da Facebook.

ReactJS utilizza un meccanismo basato su DOM virtuale per riempire dati (viste) in DOM HTML. Il DOM virtuale funziona in modo rapido per il fatto che modifica solo singoli elementi DOM invece di ricaricare DOM completo ogni volta

> Un'applicazione React è composta da più **componenti** , ciascuno responsabile dell'output di un piccolo pezzo HTML riutilizzabile. I componenti possono essere annidati all'interno di altri componenti per consentire di creare applicazioni complesse con semplici componenti. Un componente può anche mantenere lo stato interno - ad esempio, un componente TabList può memorizzare una variabile corrispondente alla scheda attualmente aperta.

React ci consente di scrivere componenti utilizzando un linguaggio specifico del dominio chiamato

JSX. JSX ci consente di scrivere i nostri componenti usando l'HTML, mescolando gli eventi JavaScript. React convertirà internamente questo in un DOM virtuale e alla fine produrrà il nostro codice HTML per noi.

Reagire " *reagisce* " allo stato in modo rapido e automatico delle modifiche apportate ai componenti per eseguire il rerender dei componenti nel DOM HTML utilizzando il DOM virtuale. Il DOM virtuale è una rappresentazione in memoria di un DOM reale. Effettuando la maggior parte dell'elaborazione all'interno del DOM virtuale piuttosto che direttamente nel DOM del browser, React può agire rapidamente e solo aggiungere, aggiornare e rimuovere componenti che sono stati modificati dall'ultimo ciclo di rendering.

Ciao mondo con funzioni senza stato

I componenti stateless traggono la loro filosofia dalla programmazione funzionale. Il che implica che: Una funzione restituisce sempre la stessa cosa esattamente su ciò che gli viene dato.

Per esempio:

```
const statelessSum = (a, b) => a + b;

let a = 0;
const statefulSum = () => a++;
```

Come puoi vedere dall'esempio precedente, statelessSum restituirà sempre gli stessi valori di a e b. Tuttavia, la funzione statefulSum non restituirà gli stessi valori dati anche senza parametri. Questo tipo di comportamento della funzione viene anche definito come *effetto collaterale* . Dal momento che, il componente influisce su qualcosa di oltre.

Pertanto, si consiglia di utilizzare i componenti senza stato più spesso, poiché sono *privi di effetti collaterali* e creeranno sempre lo stesso comportamento. Questo è quello che vuoi essere nelle tue app perché lo stato fluttuante è lo scenario peggiore per un programma gestibile.

Il tipo più elementare di componente reattivo è uno senza stato. Reagire a componenti che sono pure funzioni dei loro oggetti di scena e non richiedono alcuna gestione dello stato interno possono essere scritti come semplici funzioni JavaScript. Si dice che siano `Stateless Functional Components` perché sono una funzione solo di `props` di `props` , senza avere nessuno `state` da tenere traccia di.

Ecco un semplice esempio per illustrare il concetto di un `Stateless Functional Component` :

```
// In HTML
<div id="element"></div>

// In React
const MyComponent = props => {
    return <h1>Hello, {props.name}!</h1>;
};

ReactDOM.render(<MyComponent name="Arun" />, element);
```

```
// Will render <h1>Hello, Arun!</h1>
```

Nota che tutto ciò che fa questo componente è rendere un elemento `h1` contenente il `name` prop. Questo componente non tiene traccia di nessuno stato. Ecco un esempio ES6:

```
import React from 'react'

const HelloWorld = props => (
    <h1>Hello, {props.name}!</h1>
)

HelloWorld.propTypes = {
    name: React.PropTypes.string.isRequired
}

export default HelloWorld
```

Poiché questi componenti non richiedono un'istanza di backup per gestire lo stato, React ha più spazio per le ottimizzazioni. L'implementazione è pulita, ma finora non sono state implementate tali ottimizzazioni per i componenti stateless .

Crea App di Reazione

create-react-app è un generatore di caldaie di app React creato da Facebook. Fornisce un ambiente di sviluppo configurato per la facilità d'uso con una configurazione minima, tra cui:

- Transpilation ES6 e JSX
- Dev server con ricaricamento modulo caldo
- Linting di codice
- Prefisso automatico CSS
- Crea script con JS, CSS e image bundling e sourcemaps
- Quadro di test di Jest

Installazione

Innanzitutto, installa l'app create-react globalmente con il gestore dei pacchetti del nodo (npm).

```
npm install -g create-react-app
```

Quindi avvia il generatore nella directory scelta.

```
create-react-app my-app
```

Passare alla directory appena creata ed eseguire lo script di avvio.

```
cd my-app/
npm start
```

Configurazione

create-react-app è intenzionalmente non configurabile per impostazione predefinita. Se è richiesto un utilizzo non predefinito, ad esempio, per utilizzare un linguaggio CSS compilato come Sass, è possibile utilizzare il comando di espulsione.

```
npm run eject
```

Ciò consente la modifica di tutti i file di configurazione. NB questo è un processo irreversibile.

alternative

Le piastre riscaldanti alternative React includono:

- enclave
- nwb
- movimento
- rackt-cli
- Budo
- RSF
- veloce
- Sagui
- roc

Costruisci App React

Per creare la tua app pronta per la produzione, esegui il comando seguente

```
npm run build
```

Nozioni di base assolute sulla creazione di componenti riutilizzabili

Componenti e puntelli

Poiché React si occupa solo della vista di un'applicazione, la maggior parte dello sviluppo in React sarà la creazione di componenti. Un componente rappresenta una parte della vista della tua applicazione. Gli " `<SomeComponent someProp="some prop's value" />` " sono semplicemente gli attributi usati su un nodo JSX (ad esempio `<SomeComponent someProp="some prop's value" />`), e sono il modo principale con cui la nostra applicazione interagisce con i nostri componenti. Nel frammento sopra, all'interno di SomeComponent, avremmo accesso a `this.props` , il cui valore sarebbe l'oggetto `{someProp: "some prop's value"}` .

Può essere utile pensare a componenti React come funzioni semplici - prendono input sotto forma di "oggetti di scena" e producono output come markup. Molti componenti semplici fanno un ulteriore passo in avanti, diventando "Pure Functions", nel senso che non emettono effetti

collaterali e sono idempotenti (dato un insieme di input, il componente produrrà sempre lo stesso output). Questo obiettivo può essere applicato formalmente creando effettivamente componenti come funzioni, piuttosto che "classi". Esistono tre modi per creare un componente React:

- **Componenti funzionali ("Stateless")**

```
const FirstComponent = props => (
    <div>{props.content}</div>
);
```

- **React.createClass ()**

```
const SecondComponent = React.createClass({
    render: function () {
        return (
            <div>{this.props.content}</div>
        );
    }
});
```

- **Classi ES2015**

```
class ThirdComponent extends React.Component {
    render() {
        return (
            <div>{this.props.content}</div>
        );
    }
}
```

Questi componenti sono utilizzati esattamente nello stesso modo:

```
const ParentComponent = function (props) {
    const someText = "FooBar";
    return (
        <FirstComponent content={someText} />
        <SecondComponent content={someText} />
        <ThirdComponent content={someText} />
    );
}
```

Gli esempi sopra mostreranno tutti markup identici.

I componenti funzionali non possono avere "stato" all'interno di essi. Quindi se il tuo componente ha bisogno di uno stato, allora vai per i componenti basati sulla classe. Consultare Creazione di componenti per ulteriori informazioni.

Come nota finale, i puntelli di reazione sono immutabili una volta che sono stati passati, il che significa che non possono essere modificati all'interno di un componente. Se il genitore di un componente cambia il valore di un oggetto di scena, React gestisce la sostituzione dei vecchi oggetti di scena con il nuovo, il componente si ripresenterà utilizzando i nuovi valori.

Vedi Thinking In React and Reusable Components per immersioni più profonde nel rapporto tra oggetti di scena e componenti.

Leggi Iniziare con React online: https://riptutorial.com/it/reactjs/topic/797/iniziare-con-react

Capitolo 2: Come configurare un webpack di base, reagire e fare i babel dell'ambiente

Osservazioni

Questa pipeline di build non è esattamente ciò che chiameresti "production ready" ma ti dà un solido inizio per aggiungere ad esso le cose di cui hai bisogno per ottenere l'esperienza di sviluppo che stai cercando. L'approccio che alcune persone prendono (incluso me stesso a volte) è di prendere una pipeline completamente costruita di Yeoman.io o da qualche altra parte e quindi rimuovere le cose che non vogliono finché non si adatta allo stile. Non c'è niente di sbagliato in questo, ma forse con l'esempio sopra puoi optare per l'approccio opposto e costruire dalle ossa nude.

Alcune cose che potresti aggiungere sono cose come un framework di test e statistiche di copertura come Karma con Mocha o Jasmine. Linting con ESLint. Sostituzione di moduli caldi in webpack-dev-server in modo da poter ottenere l'esperienza di sviluppo Ctrl + S, F5. Inoltre, la pipeline corrente viene creata solo in modalità dev, quindi un'attività di produzione dovrebbe essere buona.

Grattacapi!

Si noti nella proprietà context del `webpack.config.js` che abbiamo usato il modulo del percorso del nodo per definire il nostro percorso piuttosto che concatenare semplicemente `__dirname` alla stringa `/src` perché windows odia i tagli in avanti . Quindi, per rendere la soluzione più compatibile su più piattaforme, utilizzare il nodo leva per aiutarci.

Spiegazione delle proprietà webpack.config.js

contesto

Questo è il percorso file per il quale webpack verrà utilizzato come percorso root allo scopo di risolvere i percorsi dei file relativi. Quindi in index.jsx dove usiamo `require('./index.html')` quel punto si risolve effettivamente nella directory `src/` perché lo abbiamo definito come tale in questa proprietà.

iscrizione

Dove webpack sembra prima di iniziare a raggruppare la soluzione. Questo è il motivo per cui vedrai che nell'indice.jsx stiamo cucendo insieme la soluzione con richieste e importazioni.

produzione

Qui è dove definiamo il punto in cui webpack deve rilasciare i file di file che ha trovato in bundle. Abbiamo anche definito un nome per il file in cui verranno scartati i nostri javascript e gli stili in bundle.

devserver

Queste sono impostazioni specifiche per webpack-dev-server. Il `contentBase` definisce dove il server dovrebbe renderlo root, abbiamo definito la `dist/` folder come base qui. La `port` è la porta su cui verrà ospitato il server. `open` è ciò che viene utilizzato per istruire webpack-dev-server per aprire il browser predefinito per te una volta attivato il server.

modulo> caricatori

Questo definisce una mappatura per il webpack da usare in modo che sia sa cosa fare quando trova file diversi. La proprietà `test` fornisce un'espressione regolare per il webpack da utilizzare per determinare se deve applicare questo modulo, nella maggior parte dei casi abbiamo corrispondenze sulle estensioni di file. `loader` o `loaders` forniscono il nome del modulo del caricatore che vorremmo usare per caricare il file nel pacchetto web e lasciare che il caricatore si occupi del raggruppamento di quel tipo di file. C'è anche una proprietà di `query` sul javascript, questo fornisce solo una stringa di query al loader, quindi probabilmente avremmo probabilmente usato una proprietà di query sul loader html se lo avessimo voluto. È solo un modo diverso di fare le cose.

Examples

Come costruire una pipeline per un "Hello world" personalizzato con immagini.

Passaggio 1: Installa Node.js

La pipeline di generazione che si sta costruendo si basa su Node.js, pertanto è necessario accertarsi in prima istanza che sia installato. Per istruzioni su come installare Node.js è possibile eseguire il checkout dei documenti SO per questo qui

Passaggio 2: inizializzare il progetto come modulo nodo

Aprire la cartella del progetto sulla riga di comando e utilizzare il seguente comando:

```
npm init
```

Ai fini di questo esempio, puoi sentirti libero di prendere i valori predefiniti o se desideri maggiori informazioni su ciò che significa tutto questo, puoi controllare questo documento SO sull'impostazione della configurazione del pacchetto.

Passaggio 3: installare i pacchetti npm necessari

Eseguire il seguente comando sulla riga di comando per installare i pacchetti necessari per questo esempio:

```
npm install --save react react-dom
```

Quindi, per le dipendenze dello sviluppatore, esegui questo comando:

```
npm install --save-dev babel-core babel-preset-react babel-preset-es2015 webpack babel-loader
css-loader style-loader file-loader image-webpack-loader
```

Infine, webpack e webpack-dev-server sono cose che vale la pena installare a livello globale piuttosto che come dipendenza del tuo progetto, se preferisci aggiungerlo come dipendenza a cui funzionerà, io no. Ecco il comando da eseguire:

```
npm install --global webpack webpack-dev-server
```

Passaggio 3: aggiungere un file .babelrc alla radice del progetto

Questo imposterà babel per usare i preset che hai appena installato. Il tuo file .babelrc dovrebbe assomigliare a questo:

```
{
  "presets": ["react", "es2015"]
}
```

Passaggio 4: installazione della struttura della directory del progetto

Imposta te stesso una struttura di directory che assomiglia al seguente nella radice della tua directory:

```
|- node_modules
|- src/
  |- components/
  |- images/
  |- styles/
  |- index.html
  |- index.jsx
|- .babelrc
|- package.json
```

NOTA: `node_modules` , `.babelrc` e `package.json` dovrebbero essere già tutti presenti nei precedenti passaggi che ho appena incluso, in modo da poter vedere dove si adattano.

Passaggio 5: compilare il progetto con i file di progetto Hello World

Questo non è molto importante per il processo di costruzione di una pipeline, quindi ti fornirò il codice per questi ed è possibile copiarli incollandoli in:

src / componenti / HelloWorldComponent.jsx

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

class HelloWorldComponent extends Component {
  constructor(props) {
    super(props);
    this.state = {name: 'Student'};
    this.handleChange = this.handleChange.bind(this);
  }
```

```
  handleChange(e) {
    this.setState({name: e.target.value});
  }

  render() {
    return (
      <div>
        <div className="image-container">
          <img src="./images/myImage.gif" />
        </div>
        <div className="form">
          <input type="text" onChange={this.handleChange} />
          <div>
            My name is {this.state.name} and I'm a clever cloggs because I built a React build
pipeline
          </div>
        </div>
      </div>
    );
  }
}

export default HelloWorldComponent;
```

src / images / myImage.gif

Sentiti libero di sostituirlo con qualsiasi immagine che desideri sia semplicemente lì per dimostrare che possiamo anche riunire le immagini. Se fornisci la tua immagine e la dai un nome diverso, dovrai aggiornare `HelloWorldComponent.jsx` per riflettere le tue modifiche. Allo stesso modo, se si sceglie un'immagine con un'estensione di file diversa, è necessario modificare la proprietà di `test` del caricatore di immagini nel webpack.config.js con la regex appropriata in modo che corrisponda alla nuova estensione del file.

src / stili / styles.css

```
.form {
  margin: 25px;
  padding: 25px;
  border: 1px solid #ddd;
  background-color: #eaeaea;
  border-radius: 10px;
}

.form div {
  padding-top: 25px;
}

.image-container {
  display: flex;
  justify-content: center;
}
```

index.html

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
  <meta charset="UTF-8">
  <title>Learning to build a react pipeline</title>
</head>
<body>
  <div id="content"></div>
  <script src="app.js"></script>
</body>
</html>
```

index.jsx

```
import React from 'react';
import { render } from 'react-dom';
import HelloWorldComponent from './components/HelloWorldComponent.jsx';

require('./images/myImage.gif');
require('./styles/styles.css');
require('./index.html');

render(<HelloWorldComponent />, document.getElementById('content'));
```

Passaggio 6: creare la configurazione del Webpack

Crea un file chiamato webpack.config.js nella radice del tuo progetto e copia questo codice in esso:

webpack.config.js

```
var path = require('path');

var config = {
  context: path.resolve(__dirname + '/src'),
  entry: './index.jsx',
  output: {
    filename: 'app.js',
    path: path.resolve(__dirname + '/dist'),
  },
  devServer: {
    contentBase: path.join(__dirname + '/dist'),
    port: 3000,
    open: true,
  },
  module: {
    loaders: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        loader: 'babel-loader'
      },
      {
        test: /\.css$/,
        loader: "style!css"
      },
      {
        test: /\.gif$/,
        loaders: [
          'file?name=[path][name].[ext]',
```

```
          'image-webpack',
        ]
      },
      { test: /\.(html)$/,
        loader: "file?name=[path][name].[ext]"
      }
    ],
  },
};

module.exports = config;
```

Passaggio 7: creare attività npm per la pipeline

Per fare ciò è necessario aggiungere due proprietà alla chiave di script del JSON definita nel file package.json nella radice del progetto. Rendi la tua chiave di script simile a questa:

```
  "scripts": {
    "start": "webpack-dev-server",
    "build": "webpack",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
```

Lo script di `test` sarà già stato lì e puoi scegliere se tenerlo o no, non è importante per questo esempio.

Passaggio 8: utilizzare la pipeline

Dalla riga di comando, se ci si trova nella directory root del progetto, ora si dovrebbe essere in grado di eseguire il comando:

```
npm run build
```

Questo raggrupperà la piccola applicazione che hai creato e la inserirà nella directory `dist/` che creerà nella root della cartella del tuo progetto.

Se si esegue il comando:

```
npm start
```

Quindi l'applicazione che hai creato verrà pubblicata nel tuo browser Web predefinito all'interno di un'istanza del server di sviluppo webpack.

Leggi Come configurare un webpack di base, reagire e fare i babel dell'ambiente online: https://riptutorial.com/it/reactjs/topic/6294/come-configurare-un-webpack-di-base--reagire-e-fare-i-babel-dell-ambiente

Capitolo 3: Come e perché usare le chiavi in React

introduzione

Ogni volta che si esegue il rendering di un elenco di componenti React, ogni componente deve avere un attributo `key` . La chiave può essere qualsiasi valore, ma deve essere unica per quella lista.

Quando React deve eseguire il rendering delle modifiche in un elenco di elementi, React esegue semplicemente un'iterazione su entrambi gli elenchi di bambini contemporaneamente e genera una mutazione ogni volta che c'è una differenza. Se non ci sono tasti impostati per i bambini, React esegue la scansione di ogni bambino. Altrimenti, React confronta le chiavi per sapere quali sono stati aggiunti o rimossi dall'elenco

Osservazioni

Per maggiori informazioni, visita questo link per leggere come utilizzare le chiavi: https://facebook.github.io/react/docs/lists-and-keys.html

E visita questo link per sapere perché è consigliabile utilizzare le chiavi: https://facebook.github.io/react/docs/reconciliation.html#recursing-on-children

Examples

Esempio di base

Per un componente React senza classe:

```
function SomeComponent(props){

    const ITEMS = ['cat', 'dog', 'rat']
    function getItemsList(){
        return ITEMS.map(item => <li key={item}>{item}</i>);
    }

    return (
        <ul>
            {getItemsList()}
        </ul>
    );
}
```

Per questo esempio, il componente sopra risolve:

```
<ul>
```

```
    <li key='cat'>cat</li>
    <li key='dog'>dog</li>
    <li key='rat'>rat</li>
<ul>
```

Leggi Come e perché usare le chiavi in React online:
https://riptutorial.com/it/reactjs/topic/9665/come-e-perche-usare-le-chiavi-in---react

Capitolo 4: componenti

Osservazioni

`React.createClass` stato deprecato in v15.5 e dovrebbe essere rimosso in v16 . C'è un pacchetto di sostituzione drop-in per quelli che lo richiedono ancora. Gli esempi che lo utilizzano dovrebbero essere aggiornati.

Examples

Componente di base

Dato il seguente file HTML:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Puoi creare un componente base usando il seguente codice in un file separato:

scripts / example.js

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';

class FirstComponent extends Component {
  render() {
    return (
      <div className="firstComponent">
        Hello, world! I am a FirstComponent.
      </div>
    );
  }
}
ReactDOM.render(
  <FirstComponent />, // Note that this is the same as the variable you stored above
  document.getElementById('content')
);
```

Otterrai il seguente risultato (nota cosa c'è all'interno del `div#content`):

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content">
        <div className="firstComponent">
            Hello, world! I am a FirstComponent.
        </div>
    </div>
    <script type="text/babel" src="scripts/example.js"></script>
  </body>
</html>
```

Componenti di nidificazione

Gran parte della potenza di ReactJS è la sua capacità di consentire l'annidamento di componenti. Prendi i seguenti due componenti:

```
var React = require('react');
var createReactClass = require('create-react-class');

var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = reactCreateClass({
  render: function() {
    return (
      <div className="commentForm">
        Hello, world! I am a CommentForm.
      </div>
    );
  }
});
```

È possibile nidificare e fare riferimento a tali componenti nella definizione di un componente diverso:

```
var React = require('react');
var createReactClass = require('create-react-class');
```

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList /> // Which was defined above and can be reused
        <CommentForm /> // Same here
      </div>
    );
  }
});
```

L'ulteriore nidificazione può essere effettuata in tre modi, ognuno dei quali ha il proprio spazio da utilizzare.

1. Annidamento senza l'utilizzo di bambini

(continua da sopra)

```
var CommentList = reactCreateClass({
  render: function() {
    return (
      <div className="commentList">
        <ListTitle/>
        Hello, world! I am a CommentList.
      </div>
    );
  }
});
```

Questo è lo stile in cui A compone B e B compone C.

Professionisti

- Facile e veloce per separare gli elementi dell'interfaccia utente
- Facile da iniettare oggetti di scena fino ai bambini in base allo stato del componente genitore

Contro

- Meno visibilità nell'architettura della composizione
- Meno riusabilità

Buono se

- B e C sono solo componenti di presentazione
- B dovrebbe essere responsabile per il ciclo di vita di C

2. Annidamento utilizzando i bambini

(continua da sopra)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList>
            <ListTitle/> // child
        </CommentList>
        <CommentForm />
      </div>
    );
  }
});
```

Questo è lo stile in cui A compone B e A indica a B di comporre C. Più potere ai componenti principali.

Professionisti

- Migliore gestione del ciclo di vita dei componenti
- Migliore visibilità nell'architettura della composizione
- Migliore riusabilità

Contro

- Iniezione di oggetti di scena può diventare un po 'costoso
- Minore flessibilità e potenza nei componenti del bambino

Buono se

- B dovrebbe accettare di comporre qualcosa di diverso da C in futuro o da qualche altra parte
- A dovrebbe controllare il ciclo di vita di C

B renderebbe C usando `this.props.children` , e non c'è un modo strutturato per B di sapere a che cosa servono quei bambini. Quindi, B può arricchire i componenti del bambino offrendo ulteriori oggetti di scena, ma se B ha bisogno di sapere esattamente cosa sono, # 3 potrebbe essere un'opzione migliore.

---

3. Nesting usando oggetti di scena

(continua da sopra)

```
var CommentBox = reactCreateClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList title={ListTitle}/> //prop
        <CommentForm />
      </div>
    );
  }
});
```

Questo è lo stile in cui A compone B e B offre un'opzione per A per passare qualcosa da comporre per uno scopo specifico. Composizione più strutturata

Professionisti

- Composizione come caratteristica
- Facile convalida
- Migliore compositività

Contro

- Iniezione di oggetti di scena può diventare un po 'costoso
- Minore flessibilità e potenza nei componenti del bambino

Buono se

- B ha caratteristiche specifiche definite per comporre qualcosa
- B dovrebbe solo sapere come rendere non cosa renderizzare

3 è di solito un must per creare una libreria pubblica di componenti, ma anche una buona pratica in generale per creare componenti componibili e definire chiaramente le caratteristiche della composizione. # 1 è il modo più semplice e veloce per realizzare qualcosa che funzioni, ma i # 2 e # 3 dovrebbero fornire alcuni vantaggi in vari casi d'uso.

Creazione di componenti

Questa è un'estensione dell'esempio di base:

Struttura basilare

```
import React, { Component } from 'react';
import { render } from 'react-dom';

class FirstComponent extends Component {
    render() {
        return (
```

```
            <div>
                Hello, {this.props.name}! I am a FirstComponent.
            </div>
        );
    }
}

render(
    <FirstComponent name={ 'User' } />,
    document.getElementById('content')
);
```

L'esempio precedente è chiamato componente **stateless** in quanto non contiene lo stato (nel senso React della parola).

In tal caso, alcune persone ritengono preferibile utilizzare i componenti funzionali stateless, basati sulle funzioni di freccia ES6 .

Componenti funzionali stateless

In molte applicazioni ci sono componenti intelligenti che mantengono lo stato, ma rendono i componenti stupidi che ricevono semplicemente oggetti di scena e restituiscono HTML come JSX. I componenti funzionali stateless sono molto più riutilizzabili e hanno un impatto positivo sulle prestazioni della tua applicazione.

Hanno 2 caratteristiche principali:

1. Quando sono resi ricevono un oggetto con tutti gli oggetti di scena che sono stati tramandati
2. Devono restituire il JSX da rendere

```
// When using JSX inside a module you must import React
import React from 'react';
import PropTypes from 'prop-types';

const FirstComponent = props => (
    <div>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

//arrow components also may have props validation
FirstComponent.propTypes = {
    name: PropTypes.string.isRequired,
}

// To use FirstComponent in another file it must be exposed through an export call:
export default FirstComponent;
```

Componenti stateful

In contrasto con i componenti 'senza stato' mostrati sopra, i componenti 'stateful' hanno un

oggetto stato che può essere aggiornato con il metodo `setState` . Lo stato deve essere inizializzato nel `constructor` prima che possa essere impostato:

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

class SecondComponent extends Component {
    constructor(props) {
        super(props);

        this.state = {
            toggle: true
        };

        // This is to bind context when passing onClick as a callback
        this.onClick = this.onClick.bind(this);
    }

    onClick() {
        this.setState((prevState, props) => ({
            toggle: !prevState.toggle
        }));
    }

    render() {
        return (
            <div onClick={this.onClick}>
                Hello, {this.props.name}! I am a SecondComponent.
                <br />
                Toggle is: {this.state.toggle}
            </div>
        );
    }
}
```

L'estensione di un componente con `PureComponent` anziché `Component` implementa automaticamente il metodo del ciclo di vita `shouldComponentUpdate()` con confronto tra puntelli e stati poco profondi. Ciò mantiene l'applicazione più performante riducendo la quantità di rendering non necessari che si verificano. Ciò presuppone che i componenti siano "Puri" e restituiscano sempre lo stesso output con lo stesso input di stato e oggetti di scena.

Componenti dell'ordine superiore

I componenti di ordine superiore (HOC) consentono di condividere la funzionalità del componente.

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

const PrintHello = ComposedComponent => class extends Component {
    onClick() {
        console.log('hello');
    }

    /* The higher order component takes another component as a parameter
    and then renders it with additional props */
    render() {
        return <ComposedComponent {...this.props } onClick={this.onClick} />
```

```
    }
}

const FirstComponent = props => (
    <div onClick={ props.onClick }>
        Hello, {props.name}! I am a FirstComponent.
    </div>
);

const ExtendedComponent = PrintHello(FirstComponent);
```

I componenti di ordine superiore vengono utilizzati quando si desidera condividere la logica tra più componenti indipendentemente dal modo in cui vengono visualizzati.

setState insidie

È necessario prestare attenzione quando si utilizza `setState` in un contesto asincrono. Ad esempio, potresti provare a chiamare `setState` nel callback di una richiesta get:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {}
        };
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });
    }

    render() {
        return <h1>{this.state.user}</h1>;
    }
}
```

Questo potrebbe chiamare problemi - se il callback viene chiamato dopo che `Component` è stato disinstallato, `this.setState` non sarà una funzione. Ogni volta che questo è il caso, dovresti fare attenzione a garantire che il tuo utilizzo di `setState` sia cancellabile.

In questo esempio, potresti voler annullare la richiesta XHR quando il componente smonta:

```
class MyClass extends React.Component {
    constructor() {
        super();

        this.state = {
            user: {},
```

```
            xhr: null
        };
    }

    componentWillUnmount() {
        let xhr = this.state.xhr;

        // Cancel the xhr request, so the callback is never called
        if (xhr && xhr.readyState != 4) {
            xhr.abort();
        }
    }

    componentDidMount() {
        this.fetchUser();
    }

    fetchUser() {
        let xhr = $.get('/api/users/self')
            .then((user) => {
                this.setState({user: user});
            });

        this.setState({xhr: xhr});
    }
}
```

Il metodo asincrono viene salvato come stato. Nel `componentWillUnmount` si esegue tutta la pulizia, compresa l'annullamento della richiesta XHR.

Potresti anche fare qualcosa di più complesso. In questo esempio, sto creando una funzione 'stateSetter' che accetta questo oggetto come argomento e impedisce `this.setState` quando è stata chiamata la funzione `cancel` :

```
function stateSetter(context) {
    var cancelled = false;
    return {
        cancel: function () {
            cancelled = true;
        },
        setState(newState) {
            if (!cancelled) {
                context.setState(newState);
            }
        }
    }
}

class Component extends React.Component {
    constructor(props) {
        super(props);
        this.setter = stateSetter(this);
        this.state = {
            user: 'loading'
        };
    }
    componentWillUnmount() {
        this.setter.cancel();
    }
```

```
    componentDidMount() {
        this.fetchUser();
    }
    fetchUser() {
        $.get('/api/users/self')
            .then((user) => {
                this.setter.setState({user: user});
            });
    }
    render() {
        return <h1>{this.state.user}</h1>
    }
}
```

Funziona perché la variabile `cancelled` è visibile nella chiusura `setState` abbiamo creato.

puntelli

I puntelli sono un modo per passare le informazioni in un componente React, possono avere qualsiasi tipo incluse le funzioni, a volte indicate come callback.

In JSX gli oggetti di scena vengono passati con la sintassi degli attributi

```
<MyComponent userID={123} />
```

All'interno della definizione per MyComponent userID sarà ora accessibile dall'oggetto oggetti di scena

```
// The render function inside MyComponent
render() {
    return (
        <span>The user's ID is {this.props.userID}</span>
    )
}
```

È importante definire tutti gli `props` , i loro tipi e, ove applicabile, il loro valore predefinito:

```
// defined at the bottom of MyComponent
MyComponent.propTypes = {
    someObject: React.PropTypes.object,
    userID: React.PropTypes.number.isRequired,
    title: React.PropTypes.string
};

MyComponent.defaultProps = {
    someObject: {},
    title: 'My Default Title'
}
```

In questo esempio la proprietà `someObject` è facoltativa, ma è necessario l' `userID` prop. Se si riesce a fornire `userID` a `MyComponent` , in fase di esecuzione il motore di Reagire mostrerà una console che avvisa che l'elica non è stata fornita. Attenzione però, questo avviso viene mostrato solo nella versione di sviluppo della libreria React, la versione di produzione non registra eventuali avvisi.

L'utilizzo di `defaultProps` ti consente di semplificare

```
const { title = 'My Default Title' } = this.props;
console.log(title);
```

a

```
console.log(this.props.title);
```

È anche una salvaguardia per l'uso di `array` di `object` e `functions` . Se non fornisci un puntello predefinito per un oggetto, il seguente messaggio genera un errore se il puntello non viene passato:

```
if (this.props.someObject.someKey)
```

Nell'esempio sopra, `this.props.someObject undefined` è `undefined` e quindi il controllo di `someKey` genererà un errore e il codice si interromperà. Con l'uso di `defaultProps` è possibile utilizzare tranquillamente il controllo di cui sopra.

Stati componenti - Interfaccia utente dinamica

Supponiamo di voler avere il seguente comportamento: abbiamo un'intestazione (diciamo elemento h3) e, facendo clic su di essa, vogliamo che diventi una casella di input in modo che possiamo modificare il nome dell'intestazione. React rende questo molto semplice e intuitivo utilizzando gli stati dei componenti e le istruzioni if if. (Spiegazione del codice di seguito)

```
// I have used ReactBootstrap elements. But the code works with regular html elements also
var Button = ReactBootstrap.Button;
var Form = ReactBootstrap.Form;
var FormGroup = ReactBootstrap.FormGroup;
var FormControl = ReactBootstrap.FormControl;

var Comment = reactCreateClass({
  getInitialState: function() {
    return {show: false, newTitle: ''};
  },

  handleTitleSubmit: function() {
     //code to handle input box submit - for example, issue an ajax request to change name in
database
  },

  handleTitleChange: function(e) {
    //code to change the name in form input box. newTitle is initialized as empty string. We
need to update it with the string currently entered by user in the form
    this.setState({newTitle: e.target.value});
  },

  changeComponent: function() {
    // this toggles the show variable which is used  for dynamic UI
    this.setState({show: !this.state.show)};
  },
```

```
  render: function() {

    var clickableTitle;

    if(this.state.show) {
        clickableTitle = <Form inline onSubmit={this.handleTitleSubmit}>
                            <FormGroup controlId="formInlineTitle">
                                <FormControl type="text" onChange={this.handleTitleChange}>
                            </FormGroup>
                        </Form>;
    } else {
        clickabletitle = <div>
                           <Button bsStyle="link" onClick={this.changeComponent}>
                               <h3> Default Text </h3>
                           </Button>
                        </div>;
    }

    return (
        <div className="comment">
            {clickableTitle}
        </div>
    );
  }
});

ReactDOM.render(
    <Comment />, document.getElementById('content')
);
```

La parte principale del codice è la variabile **clickableTitle** . In base allo **show della** variabile di stato, può essere un elemento Form o un elemento Button. React consente di annidare i componenti.

Quindi possiamo aggiungere un elemento {clickableTitle} nella funzione render. Cerca la variabile clickableTitle. In base al valore 'this.state.show', visualizza l'elemento corrispondente.

Variazioni di componenti funzionali stateless

```
const languages = [
  'JavaScript',
  'Python',
  'Java',
  'Elm',
  'TypeScript',
  'C#',
  'F#'
]
```

```
// one liner
const Language = ({language}) => <li>{language}</li>

Language.propTypes = {
  message: React.PropTypes.string.isRequired
}
```

```
/**
* If there are more than one line.
* Please notice that round brackets are optional here,
* However it's better to use them for readability
*/
const LanguagesList = ({languages}) => {
  <ul>
    {languages.map(language => <Language language={language} />)}
  </ul>
}

LanguagesList.PropTypes = {
  languages: React.PropTypes.array.isRequired
}
```

```
/**
 * This syntax is used if there are more work beside just JSX presentation
 * For instance some data manipulations needs to be done.
 * Please notice that round brackets after return are required,
 * Otherwise return will return nothing (undefined)
 */
const LanguageSection = ({header, languages}) => {
  // do some work
  const formattedLanguages = languages.map(language => language.toUpperCase())
  return (
    <fieldset>
      <legend>{header}</legend>
      <LanguagesList languages={formattedLanguages} />
    </fieldset>
  )
}

LanguageSection.PropTypes = {
  header: React.PropTypes.string.isRequired,
  languages: React.PropTypes.array.isRequired
}
```

```
ReactDOM.render(
  <LanguageSection
    header="Languages"
    languages={languages} />,
  document.getElementById('app')
)
```

Qui puoi trovare un esempio funzionante.

Leggi componenti online: https://riptutorial.com/it/reactjs/topic/1185/componenti

Capitolo 5: Componenti dell'ordine superiore

introduzione

Componenti ordine superiore (in breve "HOC") è un modello di progettazione di un'applicazione reattiva che viene utilizzato per migliorare i componenti con codice riutilizzabile. Consentono di aggiungere funzionalità e comportamenti alle classi di componenti esistenti.

Un HOC è una pura funzione javascript che accetta un componente come argomento e restituisce un nuovo componente con la funzionalità estesa.

Osservazioni

Gli HOC sono spesso usati nelle librerie di terze parti. Come la funzione di connessione di Redux.

Examples

Semplice componente di ordine superiore

Diciamo che vogliamo console.log ogni volta che il componente monta:

hocLogger.js

```
export default function hocLogger(Component) {
  return class extends React.Component {
    componentDidMount() {
      console.log('Hey, we are mounted!');
    }
    render() {
      return <Component {...this.props} />;
    }
  }
}
```

Usa questo HOC nel tuo codice:

MyLoggedComponent.js

```
import React from "react";
import {hocLogger} from "./hocLogger";

export class MyLoggedComponent extends React.Component {
    render() {
        return (
            <div>
                This component get's logged to console on each mount.
            </div>
        );
    }
```

```
}

// Now wrap MyLoggedComponent with the hocLogger function
export default hocLogger(MyLoggedComponent);
```

Componente ordine superiore che verifica l'autenticazione

Diciamo che abbiamo un componente che dovrebbe essere visualizzato solo se l'utente è loggato.

Quindi creiamo un HOC che verifica l'autenticazione su ogni render ():

AuthenticatedComponent.js

```
import React from "react";

export function requireAuthentication(Component) {
    return class AuthenticatedComponent extends React.Component {

        /**
         * Check if the user is authenticated, this.props.isAuthenticated
         * has to be set from your application logic (or use react-redux to retrieve it from
global state).
         */
        isAuthenticated() {
            return this.props.isAuthenticated;
        }

        /**
         * Render
         */
        render() {
            const loginErrorMessage = (
                <div>
                    Please <a href="/login">login</a> in order to view this part of the
application.
                </div>
            );

            return (
                <div>
                    { this.isAuthenticated === true ? <Component {...this.props} /> :
loginErrorMessage }
                </div>
            );
        }
    };
}

export default requireAuthentication;
```

Quindi usiamo questo componente di ordine superiore nei nostri componenti che dovrebbe essere nascosto agli utenti anonimi:

MyPrivateComponent.js

```
import React from "react";
```

```
import {requireAuthentication} from "./AuthenticatedComponent";

export class MyPrivateComponent extends React.Component {
    /**
     * Render
     */
    render() {
        return (
            <div>
                My secret search, that is only viewable by authenticated users.
            </div>
        );
    }
}

// Now wrap MyPrivateComponent with the requireAuthentication function
export default requireAuthentication(MyPrivateComponent);
```

Questo esempio è descritto in maggior dettaglio qui .

Leggi Componenti dell'ordine superiore online:
https://riptutorial.com/it/reactjs/topic/9819/componenti-dell-ordine-superiore

Capitolo 6: Componenti funzionali stateless

Osservazioni

I componenti funzionali stateless in React sono pure funzioni dei `props` passati. Questi componenti non si basano sullo stato e scartano l'uso dei metodi del ciclo di vita dei componenti. Tuttavia, è possibile definire ancora `propTypes` e `defaultPropts` .

Vedi https://facebook.github.io/react/docs/reusable-components.html#stateless-functions per ulteriori informazioni sui componenti funzionali stateless.

Examples

Componente funzionale stateless

I componenti consentono di suddividere l'interfaccia utente in pezzi *indipendenti* e *riutilizzabili* . Questa è la bellezza di React; possiamo separare una pagina in molti piccoli **componenti** riutilizzabili.

Prima di React v14 è stato possibile creare un componente React stateful utilizzando `React.Component` (in ES6) o `React.createClass` (in ES5), indipendentemente dal fatto che sia necessario o meno uno stato gestire i dati.

React v14 ha introdotto un modo più semplice per definire i componenti, solitamente definiti **componenti funzionali stateless** . Questi componenti utilizzano semplici funzioni JavaScript.

Per esempio:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Questa funzione è un componente React valido perché accetta un argomento oggetto `props` singolo con dati e restituisce un elemento React. Chiamiamo tali componenti **funzionali** perché sono letteralmente *funzioni* JavaScript.

I componenti funzionali stateless si concentrano in genere sull'interfaccia utente; lo stato dovrebbe essere gestito da componenti "contenitore" di livello superiore, o tramite Flux / Redux ecc. I componenti funzionali stateless non supportano i metodi dello stato o del ciclo di vita.

Benefici:

1. Nessun overhead di classe
2. Non devi preoccuparti di `this` parola chiave
3. Facile da scrivere e facile da capire
4. Non devi preoccuparti di gestire i valori di stato

5. Miglioramento delle prestazioni

Riepilogo : se si sta scrivendo un componente React che non richiede lo stato e si desidera creare un'interfaccia utente riutilizzabile, anziché creare un componente di reazione standard, è possibile scriverlo come **componente funzionale stateless** .

Facciamo un semplice esempio:

Supponiamo di avere una pagina in grado di registrare un utente, cercare utenti registrati o visualizzare un elenco di tutti gli utenti registrati.

Questo è il punto di ingresso dell'applicazione, `index.js` :

```
import React from 'react';
import ReactDOM from 'react-dom';

import HomePage from './homepage'

ReactDOM.render(
    <HomePage/>,
    document.getElementById('app')
);
```

Il componente `HomePage` fornisce l'interfaccia utente per la registrazione e la ricerca di utenti. Si noti che si tratta di un tipico componente React che include stato, interfaccia utente e codice comportamentale. I dati per l'elenco degli utenti registrati sono memorizzati nella variabile di `state` , ma il nostro `List` riutilizzabile (mostrato sotto) incapsula il codice UI per l'elenco.

`homepage.js` :

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

import List from './list';

export default class Temp extends Component{

    constructor(props) {
        super();
        this.state={users:[], showSearchResult: false, searchResult: []};
    }

    registerClick(){
        let users = this.state.users.slice();
        if(users.indexOf(this.refs.mail_id.value) == -1){
            users.push(this.refs.mail_id.value);
            this.refs.mail_id.value = '';
            this.setState({users});
        }else{
            alert('user already registered');
        }
    }

    searchClick(){
        let users = this.state.users;
        let index = users.indexOf(this.refs.search.value);
```

```
        if(index >= 0){
            this.setState({searchResult: users[index], showSearchResult: true});
        }else{
            alert('no user found with this mail id');
        }
    }

    hideSearchResult(){
        this.setState({showSearchResult: false});
    }

    render() {
        return (
            <div>
                <input placeholder='email-id' ref='mail_id'/>
                <input type='submit' value='Click here to register'
onClick={this.registerClick.bind(this)}/>
                <input style={{marginLeft: '100px'}} placeholder='search' ref='search'/>
                <input type='submit' value='Click here to register'
onClick={this.searchClick.bind(this)}/>
                {this.state.showSearchResult ?
                    <div>
                        Search Result:
                        <List users={[this.state.searchResult]}/>
                        <p onClick={this.hideSearchResult.bind(this)}>Close this</p>
                    </div>
                    :
                    <div>
                        Registered users:
                        <br/>
                        {this.state.users.length ?
                            <List users={this.state.users}/>
                            :
                            "no user is registered"
                        }
                    </div>
                }
            </div>
        );
    }
}
```

Infine, la nostra `List` **componenti funzionali senza stato** , che viene utilizzata, mostra sia l'elenco degli utenti registrati *che* i risultati della ricerca, ma senza mantenere alcuno stato.

`list.js` :

```
import React from 'react';
var colors = ['#6A1B9A', '#76FF03', '#4527A0'];

var List = (props) => {
    return(
        <div>
            {
                props.users.map((user, i)=>{
                    return(
                        <div key={i} style={{color: colors[i%3]}}>
                            {user}
                        </div>
```

```
                    );
                })
            }
        </div>
    );
}

export default List;
```

Riferimento: https://facebook.github.io/react/docs/components-and-props.html

Leggi Componenti funzionali stateless online:
https://riptutorial.com/it/reactjs/topic/6588/componenti-funzionali-stateless

Capitolo 7: Comunicare tra i componenti

Examples

Comunicazione tra componenti funzionali stateless

In questo esempio utilizzeremo i moduli `Redux` e `React Redux` per gestire il nostro stato di applicazione e per il re-rendering automatico dei nostri componenti funzionali., E ofcourse `React` e `React Dom`

Puoi controllare la demo completata qui

Nell'esempio seguente abbiamo tre diversi componenti e un componente connesso

- **UserInputForm** : questo componente visualizza un campo di input E quando il valore del campo cambia, chiama il metodo `inputChange` su `props` (fornito dal componente padre) e se vengono forniti anche i dati, lo visualizza nel campo di input.
- **UserDashboard:** Questo componente visualizza un messaggio semplice e nidifica anche `UserInputForm` componente, passa anche `inputChange` metodo per `UserInputForm` componente, `UserInputForm` componente consente inturn uso di questo metodo per comunicare con il componente principale.
    - **UserDashboardConnected** : questo componente avvolge semplicemente il componente `UserDashboard` utilizzando il metodo di `ReactRedux connect` . Ciò semplifica la gestione dello stato del componente e l'aggiornamento del componente quando lo stato cambia.
- **App** : questo componente `UserDashboardConnected` solo il rendering del componente `UserDashboardConnected` .

```
const UserInputForm = (props) => {

  let handleSubmit = (e) => {
    e.preventDefault();
  }

  return(
    <form action="" onSubmit={handleSubmit}>
      <label htmlFor="name">Please enter your name</label>
      <br />
      <input type="text" id="name" defaultValue={props.data.name || ''} onChange={
props.inputChange } />
    </form>
  )

}


const UserDashboard = (props) => {
```

```
  let inputChangeHandler = (event) => {
    props.updateName(event.target.value);
  }

  return(
    <div>
      <h1>Hi { props.user.name || 'User' }</h1>
      <UserInputForm data={props.user} inputChange={inputChangeHandler} />
    </div>
  )

}

const mapStateToProps = (state) => {
  return {
    user: state
  };
}
const mapDispatchToProps = (dispatch) => {
  return {
    updateName: (data) => dispatch( Action.updateName(data) ),
  };
};

const { connect, Provider } = ReactRedux;
const UserDashboardConnected = connect(
  mapStateToProps,
  mapDispatchToProps
)(UserDashboard);



const App = (props) => {
  return(
    <div>
      <h1>Communication between Stateless Functional Components</h1>
      <UserDashboardConnected />
    </div>
  )
}



const user = (state={name: 'John'}, action) => {
  switch (action.type) {
    case 'UPDATE_NAME':
      return Object.assign( {}, state, {name: action.payload}  );

    default:
      return state;
  }
};

const { createStore } = Redux;
const store = createStore(user);
const Action = {
  updateName: (data) => {
    return { type : 'UPDATE_NAME', payload: data }
  },
}
```

```
ReactDOM.render(
  <Provider store={ store }>
    <App />
  </Provider>,
  document.getElementById('application')
);
```

URL del cestino JS

Leggi Comunicare tra i componenti online: https://riptutorial.com/it/reactjs/topic/6137/comunicare-tra-i-componenti

Capitolo 8: Comunicazione tra componenti

Osservazioni

Ci sono un totale di 3 casi di comunicazione tra componenti React:

- Caso 1: comunicazione da padre a figlio
- Caso 2: comunicazione da bambino a genitore
- Caso 3: comunicazione dei componenti non correlati (qualsiasi componente a qualsiasi componente)

Examples

Componenti padre-figlio

Che il caso più semplice in realtà, molto naturale nel mondo di React e le possibilità sono - lo stai già usando.

Puoi **passare gli oggetti di scena ai componenti del bambino** . In questo `message` esempio è il sostegno che passiamo al componente figlio, il messaggio del nome è scelto arbitrariamente, puoi chiamarlo come vuoi.

```
import React from 'react';

class Parent extends React.Component {
    render() {
        const variable = 5;
        return (
            <div>
                <Child message="message for child" />
                <Child message={variable} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return <h1>{this.props.message}</h1>
    }
}

export default Parent;
```

Qui, il componente `<Parent />` esegue il rendering di due componenti `<Child />` , passando il `message for child` all'interno del primo componente e `5` all'interno del secondo.

In sintesi, hai un componente (genitore) che ne restituisce un altro (figlio) e passa ad alcuni oggetti di scena.

Componenti figlio-figlio

Inviando i dati indietro al genitore, per fare ciò passiamo semplicemente **una funzione come puntello dal componente padre al componente figlio** , e **il componente figlio chiama tale funzione** .

In questo esempio, cambieremo lo stato Genitore passando una funzione al componente Bambino e invocando quella funzione all'interno del componente Bambino.

```
import React from 'react';

class Parent extends React.Component {
    constructor(props) {
        super(props);
        this.state = { count: 0 };

        this.outputEvent = this.outputEvent.bind(this);
    }
    outputEvent(event) {
        // the event context comes from the Child
        this.setState({ count: this.state.count++ });
    }

    render() {
        const variable = 5;
        return (
            <div>
                Count: { this.state.count }
                <Child clickHandler={this.outputEvent} />
            </div>
        );
    }
}

class Child extends React.Component {
    render() {
        return (
            <button onClick={this.props.clickHandler}>
                Add One More
            </button>
        );
    }
}

export default Parent;
```

Si noti che il metodo `outputEvent` del `outputEvent` (che modifica lo stato padre) viene richiamato dall'evento `onClick` del pulsante figlio.

Componenti non correlati

L'unico modo se i tuoi componenti non hanno una relazione genitore-figlio (o sono correlati ma troppo lontani come un grand-grand-son) deve avere qualche tipo di segnale a cui un componente si abbona e l'altro scrive dentro.

Queste sono le 2 operazioni di base di qualsiasi sistema di eventi: **iscriviti / ascolta** un evento da notificare e **invia / attiva / pubblica / invia** un evento per avvisare chi lo desidera.

Ci sono almeno 3 modelli per farlo. Puoi trovare un confronto qui .

Ecco un breve riassunto:

- Pattern 1: **Event Emitter / Target / Dispatcher** : gli ascoltatori devono fare riferimento alla fonte per iscriversi.
    - per iscriversi: `otherObject.addEventListener('click', () => { alert('click!'); });`
    - per la spedizione: `this.dispatchEvent('click');`
- Motivo 2: **pubblicazione / sottoscrizione** : non è necessario un riferimento specifico all'origine che attiva l'evento, esiste un oggetto globale accessibile ovunque che gestisce tutti gli eventi.
    - per iscriversi: `globalBroadcaster.subscribe('click', () => { alert('click!'); });`
    - per la spedizione: `globalBroadcaster.publish('click');`
- Modello 3: **Segnali** : simili a Emettitore di eventi / Target / Dispatcher ma non si usano stringhe casuali qui. Ogni oggetto che potrebbe emettere eventi deve avere una proprietà specifica con quel nome. In questo modo, sai esattamente quali eventi possono emettere un oggetto.
    - per iscriversi: `otherObject.clicked.add( () => { alert('click'); });`
    - spedire: `this.clicked.dispatch();`

Leggi Comunicazione tra componenti online:
https://riptutorial.com/it/reactjs/topic/6567/comunicazione-tra-componenti

Capitolo 9: Forme di reazione

Examples

Componenti controllati

Un componente controllato è associato a un valore e le sue modifiche vengono gestite nel codice utilizzando callback basati su eventi.

```
class CustomForm extends React.Component {
constructor() {
    super();
    this.state = {
        person: {
            firstName: '',
            lastName: ''
        }
    }
}

handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}

render() {
    return (
        <form>
            <input
            type="text"
            name="firstName"
            value={this.state.firstName}
            onChange={this.handleChange.bind(this)} />

            <input
            type="text"
            name="lastName"
            value={this.state.lastName}
            onChange={this.handleChange.bind(this)} />
        </form>
    )
}
```

}

In questo esempio inizializziamo lo stato con un oggetto persona vuota. Quindi leghiamo i valori dei 2 input alle singole chiavi dell'oggetto person. Quindi, mentre l'utente digita, `handleChange` ciascun valore nella funzione `handleChange` . Poiché i valori dei componenti sono vincolati allo stato, possiamo eseguire il rerender mentre l'utente digita chiamando `setState()` .

NOTA: se non si chiama `setState()` quando si ha a che fare con componenti controllati, l'utente deve digitare, ma non vedere l'input perché React esegue solo le modifiche quando gli viene

richiesto di farlo.

È anche importante notare che i nomi degli input sono gli stessi dei nomi delle chiavi nell'oggetto person. Questo ci permette di catturare il valore nel modulo del dizionario come visto qui.

```
handleChange(event) {
    let person = this.state.person;
    person[event.target.name] = event.target.value;
    this.setState({person});
}
```

`person[event.target.name]` è uguale a `person.firstName || person.lastName` . Ovviamente ciò dipenderà dal tipo di input attualmente digitato. Poiché non sappiamo dove l'utente digiterà, utilizzando un dizionario e facendo corrispondere i nomi di input ai nomi dei tasti, ci permetterà di catturare l'input dell'utente no importa da dove viene chiamato il onChange.

Leggi Forme di reazione online: https://riptutorial.com/it/reactjs/topic/8047/forme-di-reazione

Capitolo 10: Forms e User Input

Examples

Componenti controllati

I componenti del modulo controllato sono definiti con una proprietà `value` . Il valore degli ingressi controllati è gestito da React, gli input dell'utente non avranno alcuna influenza diretta sull'input reso. Invece, una modifica alla proprietà del `value` deve riflettere questo cambiamento.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: ''
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          value={this.state.name} />
      </div>
    )
  }
}
```

L'esempio sopra mostra come la proprietà `value` definisce il valore corrente dell'input e il gestore di eventi `onChange` aggiorna lo stato del componente con l'input dell'utente.

Gli input del modulo dovrebbero essere definiti come componenti controllati laddove possibile. Ciò garantisce che lo stato del componente e il valore di input siano sincronizzati in ogni momento, anche se il valore viene modificato da un trigger diverso da un input dell'utente.

Componenti non controllati

I componenti non controllati sono input che non hanno una proprietà `value` . Al contrario dei componenti controllati, è responsabilità dell'applicazione mantenere in sincrono lo stato del componente e il valore di input.

```
class Form extends React.Component {
  constructor(props) {
    super(props);

    this.onChange = this.onChange.bind(this);

    this.state = {
      name: 'John'
    };
  }

  onChange(e) {
    this.setState({
      name: e.target.value
    });
  }

  render() {
    return (
      <div>
        <label for='name-input'>Name: </label>
        <input
          id='name-input'
          onChange={this.onChange}
          defaultValue={this.state.name} />
      </div>
    )
  }
}
```

Qui, lo stato del componente viene aggiornato tramite il gestore di eventi `onChange` , proprio come per i componenti controllati. Tuttavia, anziché una proprietà `value` , viene fornita una proprietà `defaultValue` . Questo determina il valore iniziale dell'input durante il primo rendering. Qualsiasi modifica successiva allo stato del componente non viene automaticamente riflessa dal valore di input; Se è necessario, dovrebbe essere usato un componente controllato.

Leggi Forms e User Input online: https://riptutorial.com/it/reactjs/topic/2884/forms-e-user-input

Capitolo 11: Impostazione dell'ambiente reattivo

Examples

Semplice componente di reazione

Vogliamo essere in grado di compilare sotto il componente e renderlo nella nostra pagina web

Nome file : src / index.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';

class ToDo extends React.Component {
    render() {
        return (<div>I am working</div>);
    }
}

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

Installa tutte le dipendenze

```
# install react and react-dom
$ npm i react react-dom --save

# install webpack for bundling
$ npm i webpack -g

# install babel for module loading, bundling and transpiling
$ npm i babel-core babel-loader --save

# install babel presets for react and es6
$ npm i babel-preset-react babel-preset-es2015 --save
```

Configura il webpack

Crea un file `webpack.config.js` nella `webpack.config.js` principale della directory di lavoro

Nome file : webpack.config.js

```
module.exports = {
    entry: __dirname + "/src/index.jsx",
    devtool: "source-map",
    output: {
        path: __dirname + "/build",
        filename: "bundle.js"
    },
```

```
    module: {
        loaders: [
            {test: /\.jsx?$/, exclude: /node_modules/, loader: "babel-loader"}
        ]
    }
}
```

Configura babele

Crea un file `.babelrc` nella `.babelrc` principale della nostra directory di lavoro

Nome file : .babelrc

```
{
    "presets": ["es2015","react"]
}
```

Il file HTML da utilizzare reagisce al componente

Imposta un semplice file html nella directory principale della directory del progetto

Nome file : index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
  </head>
  <body>
    <div id="App"></div>
    <script src="build/bundle.js" charset="utf-8"></script>
  </body>
</html>
```

Trasponi e impacchetta il tuo componente

Usando il webpack, puoi raggruppare il tuo componente:

```
$ webpack
```

Questo creerà il nostro file di output nella directory di `build` .

Apri la pagina HTML in un browser per vedere il componente in azione

Leggi Impostazione dell'ambiente reattivo online:
https://riptutorial.com/it/reactjs/topic/7480/impostazione-dell-ambiente-reattivo

Capitolo 12: Installazione

Examples

Installazione semplice

Impostazione delle cartelle

Questo esempio presuppone che il codice sia in `src/` e l'output da inserire in `out/` . In quanto tale, la struttura della cartella dovrebbe essere simile

```
example/
|-- src/
|   |-- index.js
|   `-- ...
|-- out/
`-- package.json
```

Impostazione dei pacchetti

Supponendo un ambiente di setup npm, abbiamo prima bisogno di configurare babel per traspondere il codice React in codice es5 compatibile.

```
$npm install --save-dev babel-core babel-loader babel-preset-es2015 babel-preset-react
```

Il comando precedente istruirà npm per installare le librerie core babel e il modulo loader da utilizzare con webpack. Installiamo anche es6 e reagiamo i preset per babel per capire il codice del modulo JSX ed es6. (Ulteriori informazioni sui preset sono disponibili qui Preselezioni di Babel)

```
$npm i -D webpack
```

Questo comando installerà webpack come dipendenza di sviluppo. (**io** sono la scorciatoia per l'installazione e **-D** la stenografia per --save-dev)

Potresti anche voler installare pacchetti webpack aggiuntivi (come caricatori aggiuntivi o l'estensione webpack-dev-server)

Infine, avremo bisogno del codice di reazione effettivo

```
$npm i -D react react-dom
```

Configurare il webpack

Con l'installazione delle dipendenze avremo bisogno di un file webpack.config.js per dire al webpack cosa fare

semplice webpack.config.js:

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  }
};
```

Questo file dice al webpack di iniziare con il file index.js (presumibilmente in src /) e convertirlo in un singolo file bundle.js nella directory `out` .

Il blocco del `module` dice a webpack di testare tutti i file incontrati sull'espressione regolare e, se corrispondono, richiamerà il loader specificato. (`babel-loader` in questo caso) Inoltre, la regex `exclude` dice al webpack di ignorare questo caricatore speciale per tutti i moduli nella cartella `node_modules` , questo aiuta ad accelerare il processo di transpilation. Infine, l'opzione `query` dice a webpack quali parametri passare a babel e viene utilizzato per passare i preset che abbiamo installato in precedenza.

Test della configurazione

Tutto ciò che rimane ora è creare il file `src/index.js` e provare a riempire l'applicazione

src / index.js:

```
'use strict'

import React from 'react'
import { render } from 'react-dom'

const App = () => {
  return <h1>Hello world!</h1>
}

render(
```

```
  <App />,
  document. getElementById('app')
)
```

Questo file normalmente renderebbe un semplice `<h1>Hello world!</h1>` Header nel tag html con l'id 'app', ma per ora dovrebbe essere sufficiente per traspolare il codice una volta.

`$./node_modules/.bin/webpack` . Eseguirà la versione localmente installata del webpack (usa `$webpack` se hai installato il webpack a livello globale con -g)

Questo dovrebbe creare il file `out/bundle.js` con il codice transpiled all'interno e conclude l'esempio.

Utilizzo di webpack-dev-server

Impostare

Dopo aver impostato un semplice progetto per usare il webpack, babel e reagire con l'emissione di `$npm i -g webpack-dev-server` installerà il server di sviluppo http per uno sviluppo più rapido.

Modifica di webpack.config.js

```
var path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    path: path.resolve(__dirname, 'out'),
    publicPath: '/public/',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules)/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  },
  devServer: {
    contentBase: path.resolve(__dirname, 'public'),
    hot: true
  }
};
```

Le modifiche sono in

- `output.publicPath` che imposta un percorso per l'invio del pacchetto (vedere i file di configurazione di Webpack per maggiori informazioni)
- `devServer`
    - `contentBase` del percorso di base per servire file statici (ad esempio `index.html`)
    - `hot` imposta il webpack-dev-server su hot reload quando le modifiche vengono apportate ai file sul disco

E alla fine abbiamo solo bisogno di un semplice index.html per testare la nostra app in.

index.html:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>React Sandbox</title>
  </head>
  <body>

    <div id="app" />

    <script src="public/bundle.js"></script>
  </body>
</html>
```

Con questa configurazione in esecuzione `$webpack-dev-server` dovrebbe avviare un server http locale sulla porta 8080 e al momento della connessione dovrebbe eseguire il rendering di una pagina contenente un `<h1>Hello world!</h1>` .

Leggi Installazione online: https://riptutorial.com/it/reactjs/topic/6441/installazione

Capitolo 13: Installazione di React, Webpack e Typescript

Osservazioni

Per ottenere l'evidenziazione della sintassi nell'editor (ad es. VS Code) è necessario scaricare le informazioni di digitazione per i moduli che si utilizzano nel progetto.

Supponi ad esempio di utilizzare React e ReactDOM nel tuo progetto e desideri ottenere l'evidenziazione e Intellisense per loro. Dovrai aggiungere i tipi al tuo progetto usando questo comando:

```
npm install --save @types/react @types/react-dom
```

Ora il tuo editor dovrebbe rilevare automaticamente queste informazioni di digitazione e fornirti il completamento automatico e Intellisense per questi moduli.

Examples

webpack.config.js

```
module.exports = {
    entry: './src/index',
    output: {
        path: __dirname + '/build',
        filename: 'bundle.js'
    },
    module: {
        rules: [{
            test: /\.tsx?$/,
            loader: 'ts-loader',
            exclude: /node_modules/
        }]
    },
    resolve: {
        extensions: ['.ts', '.tsx']
    }
};
```

I componenti principali sono (oltre alla `entry` standard, `output` e altre proprietà del pacchetto Web):

Il caricatore

Per questo è necessario creare una regola per `.tsx` le estensioni di file `.ts` e `.tsx` , specificare `ts-loader` come loader.

Risolvi le estensioni TS

È inoltre necessario aggiungere le estensioni `.ts` e `.tsx` nell'array di `resolve` , altrimenti il webpack non le vedrà.

tsconfig.json

Questo è un tsconfig minimo per farti funzionare.

```
{
    "include": [
        "src/*"
    ],
    "compilerOptions": {
        "target": "es5",
        "jsx": "react",
        "allowSyntheticDefaultImports": true
    }
}
```

Passiamo attraverso le proprietà una ad una:

`include`

Questa è una matrice di codice sorgente. Qui abbiamo solo una voce, `src/*` , che specifica che tutto nella directory `src` deve essere incluso nella compilazione.

`compilerOptions.target`

Specifica che vogliamo compilare il target ES5

`compilerOptions.jsx`

Impostando su `true` , TypeScript compila automaticamente la sintassi tsx da `<div />` a `React.createElement("div")` .

`compilerOptions.allowSyntheticDefaultImports`

Proprietà maneggevole che ti consentirà di importare i moduli nodo come se fossero moduli ES6, quindi invece di farlo

```
import * as React from 'react'
const { Component } = React
```

puoi solo farlo

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

senza errori che ti dicono che React non ha un'esportazione predefinita.

Il mio primo componente

```
import React, { Component } from 'react';
import ReactDOM from 'react-dom';


interface AppProps {
    name: string;
}
interface AppState {
    words: string[];
}

class App extends Component<AppProps, AppState> {
    constructor() {
        super();
        this.state = {
            words: ['foo', 'bar']
        };
    }

    render() {
        const { name } = this.props;
        return (<h1>Hello {name}!</h1>);
    }
}

const root = document.getElementById('root');
ReactDOM.render(<App name="Foo Bar" />, root);
```

Quando si utilizza TypeScript con React, dopo aver scaricato le definizioni di tipo React DefinitelyTyped (`npm install --save @types/react`), ogni componente richiederà l'aggiunta di annotazioni di tipo.

Lo fai in questo modo:

```
class App extends Component<AppProps, AppState> { }
```

dove `AppProps` e `AppState` sono interfacce (o digitano alias) per i puntelli e lo stato dei componenti rispettivamente.

Leggi Installazione di React, Webpack e Typescript online:
https://riptutorial.com/it/reactjs/topic/9590/installazione-di-react--webpack-e-typescript

Capitolo 14: Introduzione al rendering lato server

Examples

Componenti di rendering

Esistono due opzioni per eseguire il rendering dei componenti sul server: `renderToString` e `renderToStaticMarkup` .

renderToString

Questo renderà i componenti React in HTML sul server. Questa funzione aggiungerà anche proprietà di `data-react-` ai `data-react-` agli elementi HTML, quindi React sul client non dovrà più eseguire il rendering degli elementi.

```
import { renderToString } from "react-dom/server";
renderToString(<App />);
```

renderToStaticMarkup

Ciò renderà i componenti di React in HTML, ma senza proprietà di `data-react-` , non è consigliabile utilizzare componenti che verranno sottoposti a rendering sul client, poiché i componenti verranno sottoposti a rerender.

```
import { renderToStaticMarkup } from "react-dom/server";
renderToStaticMarkup(<App />);
```

Leggi Introduzione al rendering lato server online:
https://riptutorial.com/it/reactjs/topic/7478/introduzione-al-rendering-lato-server

Capitolo 15: JSX

Osservazioni

JSX è un **passo del preprocessore** che aggiunge la sintassi XML a JavaScript. Puoi sicuramente usare React senza JSX ma JSX rende React molto più elegante.

Proprio come XML, i tag JSX hanno un nome di tag, attributi e figli. Se un valore di attributo è racchiuso tra virgolette, il valore è una stringa. Altrimenti, avvolgere il valore in parentesi graffe e il valore è l'espressione JavaScript inclusa.

Fondamentalmente, JSX fornisce solo zucchero sintattico per la funzione
`React.createElement(component, props, ...children)` .

Quindi, il seguente codice JSX:

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}

ReactDOM.render(<HelloMessage name="Kalo" />, mountNode);
```

Compila il seguente codice JavaScript:

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
}

ReactDOM.render(React.createElement(HelloMessage, { name: "Kalo" }), mountNode);
```

In conclusione, si noti che **la seguente riga in JSX non è né una stringa né HTML** :

```
const element = <h1>Hello, world!</h1>;
```

Si chiama JSX ed è **un'estensione di sintassi di JavaScript** . JSX potrebbe ricordarti un linguaggio modello, ma ha tutta la potenza di JavaScript.

Il team di React dice nei loro documenti che consiglia di usarlo per descrivere come dovrebbe essere l'interfaccia utente.

Examples

Puntelli in JSX

Esistono diversi modi per specificare oggetti di scena in JSX.

Espressioni JavaScript

Puoi passare **qualsiasi espressione JavaScript** come supporto, circondandola con `{}` . Ad esempio, in questo JSX:

```
<MyComponent count={1 + 2 + 3 + 4} />
```

All'interno di `MyComponent` , il valore di `props.count` sarà `10` , poiché viene valutata l'espressione `1 + 2 + 3 + 4` .

Se le istruzioni e i cicli non sono espressioni in JavaScript, non possono essere utilizzate direttamente in JSX.

String letterali

Naturalmente, puoi anche passare qualsiasi `string literal` come oggetto di `prop` . Queste due espressioni JSX sono equivalenti:

```
<MyComponent message="hello world" />

<MyComponent message={'hello world'} />
```

Quando si passa una stringa letterale, il suo valore è HTML-escape. Quindi queste due espressioni JSX sono equivalenti:

```
<MyComponent message="&lt;3" />

<MyComponent message={'<3'} />
```

Questo comportamento di solito non è rilevante. È solo menzionato qui per completezza.

Valore predefinito puntelli

Se non si passa alcun valore per un sostegno, il **valore predefinito è** `true` . Queste due espressioni JSX sono equivalenti:

```
<MyTextBox autocomplete />

<MyTextBox autocomplete={true} />
```

Tuttavia, il team di React dice che nei loro documenti l' **utilizzo di questo approccio non è raccomandato** , perché può essere confuso con l'oggetto ES6 shorthand `{foo}` che è l'abbreviazione di `{foo: foo}` piuttosto che di `{foo: true}` . Dicono che questo comportamento è solo lì in modo che corrisponda al comportamento di HTML.

Spread Attributes

Se hai già oggetti di scena come oggetto e vuoi passarlo in JSX, puoi usare `...` come operatore di spread per passare l'intero oggetto oggetti di scena. Questi due componenti sono equivalenti:

```
function Case1() {
  return <Greeting firstName="Kaloyab" lastName="Kosev" />;
}

function Case2() {
  const person = {firstName: 'Kaloyan', lastName: 'Kosev'};
  return <Greeting {...person} />;
}
```

Bambini in JSX

Nelle espressioni JSX che contengono sia un tag di apertura che un tag di chiusura, il contenuto tra questi tag viene passato come puntello speciale: `props.children` . Esistono diversi modi per passare i bambini:

String letterali

Puoi mettere una stringa tra i tag di apertura e di chiusura e `props.children` sarà solo quella stringa. Questo è utile per molti degli elementi HTML incorporati. Per esempio:

```
<MyComponent>
    <h1>Hello world!</h1>
</MyComponent>
```

Questo è JSX valido, e `props.children` in MyComponent sarà semplicemente `<h1>Hello world!</h1>` .

Nota che **l'HTML non ha caratteri maiuscoli** , quindi puoi generalmente scrivere JSX proprio come se scrivessi HTML.

Bare a mente, che in questo caso JSX:

- rimuove gli spazi bianchi all'inizio e alla fine di una riga;
- rimuove le righe vuote;
- le nuove righe adiacenti ai tag vengono rimosse;
- le nuove linee che si verificano nel mezzo di stringhe letterali sono condensate in un unico spazio.

JSX bambini

Puoi fornire più elementi JSX come i bambini. Questo è utile per visualizzare i componenti nidificati:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

È possibile **combinare diversi tipi di bambini, quindi è possibile utilizzare letterali stringa insieme ai bambini JSX** . Questo è un altro modo in cui JSX è come HTML, quindi questo è sia JSX valido che HTML valido:

```
<div>
  <h2>Here is a list</h2>
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

Si noti che un componente React **non può restituire più elementi React, ma una singola espressione JSX può avere più figli** . Quindi, se vuoi che un componente mostri più oggetti, puoi avvolgerli in un `div` come nell'esempio sopra.

Espressioni JavaScript

È possibile passare qualsiasi espressione JavaScript come child racchiudendola entro `{}` . Ad esempio, queste espressioni sono equivalenti:

```
<MyComponent>foo</MyComponent>

<MyComponent>{'foo'}</MyComponent>
```

Questo è spesso utile per il rendering di un elenco di espressioni JSX di lunghezza arbitraria. Ad esempio, questo rende un elenco HTML:

```
const Item = ({ message }) => (
```

```
  <li>{ message }</li>
);

const TodoList = () => {
  const todos = ['finish doc', 'submit review', 'wait stackoverflow review'];
  return (
    <ul>
      { todos.map(message => (<Item key={message} message={message} />)) }
    </ul>
  );
};
```

Nota che le espressioni JavaScript possono essere mescolate con altri tipi di bambini.

Funziona da bambino

Normalmente, le espressioni JavaScript inserite in JSX valuteranno una stringa, un elemento React o un elenco di tali elementi. Tuttavia, `props.children` funziona come qualsiasi altro oggetto di `props.children` in quanto può passare qualsiasi tipo di dati, non solo quelli che React sa come renderizzare. Ad esempio, se si dispone di un componente personalizzato, è possibile farlo prendere una richiamata come `props.children` :

```
const ListOfTenThings = () => (
  <Repeat numTimes={10}>
    {(index) => <div key={index}>This is item {index} in the list</div>}
  </Repeat>
);

// Calls the children callback numTimes to produce a repeated component
const Repeat = ({ numTimes, children }) => {
  let items = [];
  for (let i = 0; i < numTimes; i++) {
    items.push(children(i));
  }
  return <div>{items}</div>;
};
```

I bambini passati a un componente personalizzato possono essere qualsiasi cosa, purché quel componente li trasformi in qualcosa che React può capire prima del rendering. Questo utilizzo non è comune, ma funziona se si desidera ottimizzare ciò di cui JSX è capace.

Valori ignorati

Si noti che `false` , `null` , `undefined` e `true` sono figli validi. Ma semplicemente non rendono. Queste espressioni JSX renderanno tutte la stessa cosa:

```
<MyComponent />
```

```
<MyComponent></MyComponent>

<MyComponent>{false}</MyComponent>

<MyComponent>{null}</MyComponent>

<MyComponent>{true}</MyComponent>
```

Questo è estremamente utile per rendere condizionalmente gli elementi di React. Questo JSX esegue il rendering solo se `showHeader` è true:

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

Un avvertimento importante è che alcuni valori "falsi", come il numero `0` , sono ancora resi da React. Ad esempio, questo codice non si comporta come ci si potrebbe aspettare perché `0` verrà stampato quando `props.messages` è un array vuoto:

```
<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

Un approccio per risolvere questo problema è assicurarsi che l'espressione prima di `&&` sia sempre boolean:

```
<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>
```

Infine, tieni a mente che se vuoi che un valore come `false` , `true` , `null` o `undefined` appaia nell'output, devi prima convertirlo in una stringa:

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

Leggi JSX online: https://riptutorial.com/it/reactjs/topic/8027/jsx

Capitolo 16: Le chiavi reagiscono

introduzione

Le chiavi in risposta vengono utilizzate per identificare internamente un elenco di elementi DOM della stessa gerarchia.

Quindi, se stai iterando su un array per mostrare un elenco di elementi li, ognuno degli elementi li ha bisogno di un identificativo univoco specificato dalla proprietà chiave. Questo di solito può essere l'id del tuo oggetto del database o l'indice dell'array.

Osservazioni

L'utilizzo dell'indice dell'array come chiave non è generalmente raccomandato quando l'array cambierà nel tempo. Da React Docs:

> Come ultima risorsa, puoi passare l'indice dell'oggetto nell'array come chiave. Questo può funzionare bene se gli articoli non vengono mai riordinati, ma i riordini saranno lenti.

Un buon esempio a riguardo: https://medium.com/@robinpokorny/index-as-a-key-is-an-anti-pattern-e0349aece318

Examples

Utilizzando l'id di un elemento

Qui stiamo visualizzando un elenco di elementi che vengono passati agli oggetti di scena del nostro componente.

Ogni articolo di TOD ha una proprietà text e id. Immagina che la proprietà id provenga da un datastore back-end ed è un valore numerico univoco:

```
todos = [
  {
    id: 1,
    text: 'value 1'
  },
  {
    id: 2,
    text: 'value 2'
  },
  {
    id: 3,
    text: 'value 3'
  },
  {
    id: 4,
```

```
    text: 'value 4'
  },
];
```

`todo-${todo.id}` l'attributo chiave di ogni elemento dell'elenco iterato su `todo-${todo.id}` modo che la reazione possa identificarlo internamente:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo) =>
        <li key={ `todo-${todo.id}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Utilizzando l'indice dell'array

Se non disponi di ID di database univoci, puoi anche utilizzare l'indice numerico del tuo array in questo modo:

```
render() {
  const { todos } = this.props;
  return (
    <ul>
      { todos.map((todo, index) =>
        <li key={ `todo-${index}` }>
          { todo.text }
        </li>
      }
    </ul>
  );
}
```

Leggi Le chiavi reagiscono online: https://riptutorial.com/it/reactjs/topic/9805/le-chiavi-reagiscono

Capitolo 17: Prestazione

Examples

Nozioni di base - HTML DOM vs Virtual DOM

HTML DOM è costoso

Ogni pagina Web è rappresentata internamente come un albero di oggetti. Questa rappresentazione è chiamata *Modello oggetto documento* . Inoltre, è un'interfaccia linguistica che consente ai linguaggi di programmazione (come JavaScript) di accedere agli elementi HTML.

In altre parole

> HTML DOM è uno standard per come ottenere, modificare, aggiungere o eliminare elementi HTML.

Tuttavia, queste **operazioni DOM** sono estremamente **costose** .

DOM virtuale è una soluzione

Quindi il team di React ha avuto l'idea di astrarre il *DOM HTML* e creare il proprio *Virtual DOM* per calcolare il numero minimo di operazioni che è necessario applicare sul *DOM HTML* per replicare lo stato attuale della nostra applicazione.

Il Virtual DOM risparmia tempo da modifiche DOM non necessarie.

Come esattamente?

In ogni momento, React ha lo stato dell'applicazione rappresentato come un `Virtual DOM` . Ogni volta che lo stato dell'applicazione cambia, questi sono i passaggi che React esegue per ottimizzare le prestazioni

1. Genera un nuovo *DOM virtuale* che rappresenta il nuovo stato della nostra applicazione
2. Confronta il vecchio Virtual DOM (che rappresenta il DOM HTML attuale) rispetto al nuovo Virtual DOM
3. Basato su 2. trova il numero minimo di operazioni per trasformare il vecchio Virtual DOM (che rappresenta il DOM HTML attuale) nel nuovo Virtual DOM

- per saperne di più su questo - leggi React's Diff Algorithm

4. Dopo che tali operazioni sono state trovate, vengono mappate nelle loro equivalenti operazioni *DOM HTML*

- Ricorda, il *Virtual DOM* è solo un'astrazione del *DOM HTML* e c'è una relazione isomorfa tra di loro

5. Ora il numero minimo di operazioni che sono state trovate e trasferite alle loro equivalenti operazioni *DOM HTML* sono ora applicate direttamente sul *DOM HTML* dell'applicazione, che consente di risparmiare tempo dalla modifica del *DOM HTML* inutilmente.

Nota: le operazioni applicate sul DOM virtuale sono economiche, poiché il DOM virtuale è un oggetto JavaScript.

React's diff algoritm

La generazione del numero minimo di operazioni per trasformare un albero in un altro ha una complessità nell'ordine di O (n ^ 3) dove n è il numero di nodi nell'albero. React si basa su due presupposti per risolvere questo problema in un tempo lineare - O (n)

1. Due componenti della stessa classe genereranno alberi simili e due componenti di classi diverse genereranno alberi diversi.

2. È possibile fornire una chiave univoca per elementi che è stabile tra diversi rendering.

Per decidere se due nodi sono diversi, React differenzia 3 casi

1. Due nodi sono diversi, se hanno tipi diversi.

- ad esempio, `<div>...</div>` è diverso da `<span>...</span>`

2. Ogni volta che due nodi hanno chiavi diverse

- ad esempio, `<div key="1">...</div>` è diverso da `<div key="2">...</div>`

Inoltre, **ciò che segue è fondamentale ed estremamente importante per capire** se si desidera ottimizzare le prestazioni

> Se [i due nodi] non sono dello stesso tipo, React non tenterà nemmeno di far corrispondere ciò che rendono. Rimuove solo il primo dal DOM e inserisce il secondo.

Ecco perché

> È molto improbabile che un elemento generi un DOM che assomiglierà a ciò che genererebbe. Invece di perdere tempo a cercare di abbinare queste due strutture, React ricostruisce l'albero da zero.

Consigli e trucchi

Quando due nodi non sono dello stesso tipo, React non prova ad abbinarli: rimuove solo il primo nodo dal DOM e inserisce il secondo. Ecco perché dice il primo suggerimento

1. Se ti vedi alternando tra due classi di componenti con un output molto simile, potresti voler

fare la stessa classe.

---

2. Utilizzare shouldComponentUpdate per impedire al componente di eseguire il rerender, se si sa che non cambierà, ad esempio

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

Misura delle prestazioni con ReactJS

Non puoi migliorare qualcosa che non puoi misurare . Per migliorare le prestazioni dei componenti React, dovresti essere in grado di misurarlo. ReactJS fornisce strumenti *addon* per misurare le prestazioni. Importa il modulo `react-addons-perf` per misurare le prestazioni

```
import Perf from 'react-addons-perf' // ES6
var Perf = require('react-addons-perf') // ES5 with npm
var Perf = React.addons.Perf; // ES5 with react-with-addons.js
```

Puoi utilizzare i seguenti metodi dal modulo `Perf` importato:

- Perf.printInclusive ()
- Perf.printExclusive ()
- Perf.printWasted ()
- Perf.printOperations ()
- Perf.printDOM ()

Il più importante di cui avrai bisogno la maggior parte del tempo è `Perf.printWasted()` che ti offre la rappresentazione tabellare del tempo sprecato del tuo singolo componente

(index)	Owner > component	Waste
0	"Todos > TodoItem"	102.7

Total time: 132.71 ms

Puoi notare la colonna del **tempo sprecato** nella tabella e migliorare le prestazioni del componente utilizzando la sezione **Suggerimenti e trucchi** sopra

Fai riferimento alla Guida ufficiale di React e all'eccellente articolo di Benchling Engg. su React Performance

Leggi Prestazione online: https://riptutorial.com/it/reactjs/topic/6875/prestazione

Capitolo 18: Puntelli in React

Osservazioni

NOTA: A partire da React 15.5 e il componente PropTypes vive nel proprio pacchetto npm, ovvero 'prop-types' e ha bisogno di una propria istruzione di importazione quando si usano PropTypes. Vedi la documentazione ufficiale di reazione per il cambiamento di rottura: https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html

Examples

introduzione

`props` sono usati per passare dati e metodi da un componente genitore a un componente figlio.

Cose interessanti su `props` di `props`

1. Sono immutabili.
2. Ci permettono di creare componenti riutilizzabili.

Esempio di base

```
class Parent extends React.Component{
  doSomething(){
     console.log("Parent component");
  }
  render() {
    return <div>
         <Child
           text="This is the child number 1"
           title="Title 1"
           onClick={this.doSomething} />
         <Child
           text="This is the child number 2"
           title="Title 2"
           onClick={this.doSomething} />
      </div>
  }
}

class Child extends React.Component{
  render() {
    return <div>
       <h1>{this.props.title}</h1>
       <h2>{this.props.text}</h2>
      </div>
  }
}
```

Come puoi vedere nell'esempio, grazie agli `props` di `props` possiamo creare componenti riutilizzabili.

Oggetti di scena di default

`defaultProps` consente di impostare i valori di default, o fallback, per i propri `props` . `defaultProps` sono utili quando si chiamano componenti da viste diverse con oggetti fissi, ma in alcune viste è necessario passare un valore diverso.

Sintassi

ES5

```
var MyClass = React.createClass({
  getDefaultProps: function() {
    return {
      randomObject: {},
      ...
    };
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.defaultProps = {
    randomObject: {},
    ...
}
```

ES7

```
class MyClass extends React.Component {
    static defaultProps = {
        randomObject: {},
        ...
    };
}
```

Il risultato di `getDefaultProps()` o `defaultProps` verrà memorizzato nella cache e utilizzato per garantire che `this.props.randomObject` abbia un valore se non è stato specificato dal componente principale.

PropTypes

`propTypes` consente di specificare che cosa `props` i vostri bisogni dei componenti e il tipo dovrebbero essere. Il tuo componente funzionerà senza impostare `propTypes` , ma è buona norma definirlo in quanto renderà il tuo componente più leggibile, fungerà da documentazione per gli altri

sviluppatori che stanno leggendo il tuo componente, e durante lo sviluppo, React ti avviserà se provi a imposta un oggetto di scena diverso dalla definizione che hai impostato per esso.

Alcune primitive `propTypes` e comunemente utilizzabili `propTypes` sono -

```
  optionalArray: React.PropTypes.array,
  optionalBool: React.PropTypes.bool,
  optionalFunc: React.PropTypes.func,
  optionalNumber: React.PropTypes.number,
  optionalObject: React.PropTypes.object,
  optionalString: React.PropTypes.string,
  optionalSymbol: React.PropTypes.symbol
```

Se si allega `isRequired` a qualsiasi `propType` allora quel puntello deve essere fornito durante la creazione dell'istanza di quel componente. Se non si forniscono i `propTypes` **richiesti** , l'istanza del componente non può essere creata.

Sintassi

ES5

```
var MyClass = React.createClass({
  propTypes: {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
  }
}
```

ES6

```
class MyClass extends React.Component {...}

MyClass.propTypes = {
    randomObject: React.PropTypes.object,
    callback: React.PropTypes.func.isRequired,
    ...
};
```

ES7

```
class MyClass extends React.Component {
     static propTypes = {
        randomObject: React.PropTypes.object,
        callback: React.PropTypes.func.isRequired,
        ...
    };
}
```

Convalida dei puntelli più complessa

Allo stesso modo, `PropTypes` ti consente di specificare una convalida più complessa

Convalidare un oggetto

```
...
    randomObject: React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    }).isRequired,
...
```

Convalida sulla matrice di oggetti

```
...
    arrayOfObjects: React.PropTypes.arrayOf(React.PropTypes.shape({
        id: React.PropTypes.number.isRequired,
        text: React.PropTypes.string,
    })).isRequired,
...
```

Passando giù puntelli usando l'operatore di spread

Invece di

```
var component = <Component foo={this.props.x} bar={this.props.y} />;
```

Dove ogni proprietà deve essere passata come un singolo valore prop è possibile utilizzare l'operatore di diffusione `...` supportato per gli array in ES6 per trasmettere tutti i valori. Il componente sarà ora simile a questo.

```
var component = <Component {...props} />;
```

Ricorda che le proprietà dell'oggetto che si passano vengono copiate sui puntelli del componente.

L'ordine è importante Gli attributi successivi sostituiscono quelli precedenti.

```
var props = { foo: 'default' };
var component = <Component {...props} foo={'override'} />;
console.log(component.props.foo); // 'override'
```

Un altro caso è che puoi anche usare l'operatore di spread per passare solo parti di oggetti di scena ai componenti figli, quindi puoi usare di nuovo la sintassi destrutturante dai puntelli.

È molto utile quando i componenti dei bambini hanno bisogno di molti oggetti di scena ma non vogliono passarli uno per uno.

```
const { foo, bar, other } = this.props // { foo: 'foo', bar: 'bar', other: 'other' };
var component = <Component {...{foo, bar}} />;
const { foo, bar } = component.props
console.log(foo, bar); // 'foo bar'
```

Oggetti di scena e composizione dei componenti

I componenti "figlio" di un componente sono disponibili su un puntello speciale, `props.children` . Questo puntello è molto utile per i componenti "Compositing" e può rendere il markup JSX più intuitivo o riflettente della struttura finale del DOM:

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {this.props.children}
            </div>
        </article>
    );
}
```

Che ci consente di includere un numero arbitrario di sottoelementi quando utilizzi il componente in un secondo momento:

```
var ParentComponent = function () {
    return (
        <SomeComponent heading="Amazing Article Box" >
            <p className="first"> Lots of content </p>
            <p> Or not </p>
        </SomeComponent>
    );
}
```

Props.children può anche essere manipolato dal componente. Poiché props.children può o non può essere un array , React fornisce funzioni di utilità come React.Children . Considera nell'esempio precedente se avessimo voluto racchiudere ogni paragrafo nel proprio elemento `<section>` :

```
var SomeComponent = function () {
    return (
        <article className="textBox">
            <header>{this.props.heading}</header>
            <div className="paragraphs">
                {React.Children.map(this.props.children, function (child) {
                    return (
                        <section className={child.props.className}>
                            React.cloneElement(child)
                        </section>
                    );
                })}
            </div>
        </article>
    );
}
```

Nota l'uso di React.cloneElement per rimuovere gli oggetti di scena dal tag `<p>` secondario - perché gli oggetti di scena sono immutabili, questi valori non possono essere modificati direttamente. Invece, un clone senza questi oggetti di scena deve essere usato.

Inoltre, quando si aggiungono elementi nei loop, tenere presente che React riconcilia i bambini durante un rerender e considerare seriamente l' inclusione di un puntello `key` univoco a livello globale sugli elementi figlio aggiunti in un ciclo.

Rilevazione del tipo di componenti per bambini

A volte è davvero utile conoscere il tipo di componente figlio durante l'iterazione attraverso di essi. Per iterare attraverso i componenti figli è possibile utilizzare la funzione React `Children.map` util:

```
React.Children.map(this.props.children, (child) => {
  if (child.type === MyComponentType) {
    ...
  }
});
```

L'oggetto figlio espone la proprietà `type` che è possibile confrontare con un componente specifico.

Leggi Puntelli in React online: https://riptutorial.com/it/reactjs/topic/2749/puntelli-in-react

Capitolo 19: React Boilerplate [React + Babel + Webpack]

Examples

Impostazione del progetto

È necessario Node Package Manager per installare le dipendenze del progetto. Scarica il nodo per il tuo sistema operativo da Nodejs.org . Node Package Manager viene fornito con nodo.

È inoltre possibile utilizzare Node Version Manager per gestire meglio le versioni di nodo e npm. È ottimo per testare il tuo progetto su diverse versioni di nodi. Tuttavia, non è raccomandato per l'ambiente di produzione.

Una volta installato il nodo sul tuo sistema, vai avanti e installa alcuni pacchetti essenziali per far esplodere il tuo primo progetto React usando Babel e Webpack.

Prima di iniziare effettivamente a premere i comandi nel terminale. Dai un'occhiata a cosa sono usati Babel e Webpack .

Puoi avviare il tuo progetto eseguendo `npm init` nel tuo terminale. Segui la configurazione iniziale. Successivamente, esegui i seguenti comandi nel tuo terminale-

dipendenze:

```
npm install react react-dom --save
```

Dev Dependecies:

```
npm install babel-core babel-loader babel-preset-es2015 babel-preset-react babel-preset-stage-0
webpack webpack-dev-server react-hot-loader --save-dev
```

Dipendenze Dev opzionali:

```
npm install eslint eslint-plugin-react babel-eslint --save-dev
```

Si può fare riferimento a questo esempio di pacchetto.json

Crea `.babelrc` nella root del tuo progetto con i seguenti contenuti:

```
{
  "presets": ["es2015", "stage-0", "react"]
}
```

Se lo `.eslintrc` crea `.eslintrc` nella root del tuo progetto con i seguenti contenuti:

```
{
  "ecmaFeatures": {
    "jsx": true,
```

```
    "modules": true
  },
  "env": {
    "browser": true,
    "node": true
  },
  "parser": "babel-eslint",
  "rules": {
    "quotes": [2, "single"],
    "strict": [2, "never"],
  },
  "plugins": [
    "react"
  ]
}
```

Crea un file `.gitignore` per impedire il caricamento dei file generati sul tuo repository git.

```
node_modules
npm-debug.log
.DS_Store
dist
```

Creare il file `webpack.config.js` con i seguenti contenuti minimi.

```
var path = require('path');
var webpack = require('webpack');

module.exports = {
  devtool: 'eval',
  entry: [
    'webpack-dev-server/client?http://localhost:3000',
    'webpack/hot/only-dev-server',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.HotModuleReplacementPlugin()
  ],
  module: {
    loaders: [{
      test: /\.js$/,
      loaders: ['react-hot', 'babel'],
      include: path.join(__dirname, 'src')
    }]
  }
};
```

E infine, creare un file `sever.js` per poter avviare `npm start` , con i seguenti contenuti:

```
var webpack = require('webpack');
var WebpackDevServer = require('webpack-dev-server');
var config = require('./webpack.config');
```

```
new WebpackDevServer(webpack(config), {
  publicPath: config.output.publicPath,
  hot: true,
  historyApiFallback: true
}).listen(3000, 'localhost', function (err, result) {
  if (err) {
    return console.log(err);
  }

  console.log('Serving your awesome project at http://localhost:3000/');
});
```

Crea il file `src/app.js` per vedere come fa il tuo progetto React.

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

export default class App extends Component {
  render() {
    return (
      <h1>Hello, world.</h1>
    );
  }
}
```

Esegui il `node server.js` o `npm start` dal terminale, se hai definito il significato di `start` nel tuo `package.json`

progetto di avviamento reattivo

Informazioni su questo progetto

Questo è un semplice progetto di caldaia. Questo post ti guiderà a configurare l'ambiente per ReactJs + Webpack + Bable.

Iniziamo

avremo bisogno del gestore dei pacchetti del nodo per attivare il server Express e gestire le dipendenze in tutto il progetto. se sei nuovo nel gestore dei pacchetti del nodo, puoi controllare qui . Nota: l'installazione del gestore pacchetti nodo è richiesta qui.

Crea una cartella con un nome adatto e naviga dal terminale o dalla GUI. Quindi vai al terminale e digita `npm init` questo creerà un file package.json, Niente di spaventoso, ti farà poche domande come nome del tuo progetto, versione, descrizione, punto di ingresso, repository git, autore, licenza ecc. Qui il punto di ingresso è importante perché inizialmente il nodo lo cercherà quando si esegue il progetto. Alla fine ti chiederà di verificare le informazioni che fornisci. Puoi digitare *sì* o modificarlo. Bene, il nostro file *package.json* è pronto.

Configurazione del server Express eseguire *npm install express @ 4 --save* . Queste sono tutte le dipendenze necessarie per questo progetto. Il flag di salvataggio è importante, senza che il file *package.js* non venga aggiornato. L'attività principale di *package.json* è di memorizzare l'elenco delle dipendenze. Aggiungerà Express versione 4. Il tuo *pacchetto.json* avrà l'aspetto di `"dependencies": { "express": "^4.13.4", ..... },`

Dopo il download completo puoi vedere che c'è la cartella *node_modules* e la sottocartella delle nostre dipendenze. Ora nella radice del progetto crea un nuovo file file *server.js* . Ora stiamo impostando il server express. Ho intenzione di superare tutto il codice e spiegarlo in seguito.

```
var express = require('express');
// Create our app
var app = express();

app.use(express.static('public'));

app.listen(3000, function () {
  console.log('Express server is using port:3000');
});
```

var express = require ('express'); questo ti darà l'accesso di tutta la Express API.

var app = express (); chiamerà la libreria espressa come funzione. *app.use ();* lascia che aggiunga la funzionalità alla tua applicazione Express. *app.use (express.static ('pubblico'));* specificherà il nome della cartella che verrà esposta nel nostro server web. *app.listen (port, function () {})* qui la nostra porta sarà *3000* e la funzione che stiamo chiamando verificherà che il web server stia funzionando correttamente. Ecco che è configurato il server express.

Ora vai al nostro progetto e crea una nuova cartella pubblica e crea il file *index.html* . *index.html* è il file predefinito per l'applicazione e il server Express cercherà questo file. *Index.html* è un semplice file html che sembra

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
</head>

<body>
  <h1>hello World</h1>
</body>

</html>
```

Andare al percorso del progetto attraverso il terminale e digitare *node server.js* . Quindi vedrai * console.log ('Il server Express usa la porta: 3000'); *.

Vai al browser e digita *http: // localhost: 3000* nella barra di navigazione vedrai *ciao Mondo* .

Ora vai nella cartella pubblica e crea un nuovo file *app.jsx* . JSX è un passo del preprocessore che aggiunge la sintassi XML al tuo JavaScript. Puoi sicuramente usare React senza JSX, ma JSX rende React molto più elegante. Ecco il codice di esempio per *app.jsx*

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

Ora vai su *index.html* e modifica il codice, dovrebbe assomigliare a questo

```
<!DOCTYPE html>
<html>

<head>
  <meta charset="UTF-8"/>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23
/browser.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react.js">
</script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react-dom.js">    </script>
</head>

<body>
  <div id="app"></div>

  <script type="text/babel" src="app.jsx"></script>
</body>

</html>
```

Con questo in atto hai finito, spero che tu lo trovi semplice.

Leggi React Boilerplate [React + Babel + Webpack] online:
https://riptutorial.com/it/reactjs/topic/5969/react-boilerplate--react-plus-babel-plus-webpack-

Capitolo 20: React Component Lifecycle

introduzione

I metodi del ciclo di vita devono essere utilizzati per eseguire codice e interagire con il componente in diversi punti della vita dei componenti. Questi metodi sono basati su un componente Montaggio, Aggiornamento e Smontaggio.

Examples

Creazione di componenti

Quando viene creato un componente React, vengono chiamate un numero di funzioni:

- Se si utilizza `React.createClass` (ES5), vengono chiamate 5 funzioni definite dall'utente
- Se stai usando la `class Component extends React.Component` (ES6), vengono chiamate 3 funzioni definite dall'utente

`getDefaultProps()` (solo ES5)

Questo è il **primo** metodo chiamato.

I valori delle proporzioni restituiti da questa funzione verranno utilizzati come valori predefiniti se non vengono definiti quando il componente viene istanziato.

Nell'esempio seguente, `this.props.name` verrà `this.props.name` come `this.props.name` su `Bob` se non diversamente specificato:

```
getDefaultProps() {
  return {
    initialCount: 0,
    name: 'Bob'
  };
}
```

`getInitialState()` (solo ES5)

Questo è il **secondo** metodo chiamato.

Il valore restituito da `getInitialState()` definisce lo stato iniziale del componente React. Il framework React chiamerà questa funzione e assegnerà il valore restituito a `this.state` .

Nell'esempio seguente, `this.state.count` verrà inizializzato con il valore di `this.props.initialCount` :

```
getInitialState() {
  return {
    count : this.props.initialCount
  };
}
```

`componentWillMount()` (ES5 e ES6)

Questo è il **terzo** metodo chiamato.

Questa funzione può essere utilizzata per apportare modifiche finali al componente prima che venga aggiunto al DOM.

```
componentWillMount() {
  ...
}
```

`render()` (ES5 e ES6)

Questo è il **quarto** metodo chiamato.

La funzione `render()` dovrebbe essere una pura funzione dello stato e degli oggetti del componente. Restituisce un singolo elemento che rappresenta il componente durante il processo di rendering e dovrebbe essere una rappresentazione di un componente DOM nativo (ad esempio `<p />`) o un componente composito. Se nulla deve essere reso, può restituire `null` o `undefined` .

Questa funzione verrà richiamata dopo qualsiasi modifica ai puntelli o allo stato del componente.

```
render() {
  return (
    <div>
      Hello, {this.props.name}!
    </div>
  );
}
```

`componentDidMount()` (ES5 e ES6)

Questo è il **quinto** metodo chiamato.

Il componente è stato montato e ora puoi accedere ai nodi DOM del componente, ad esempio tramite i `refs` .

Questo metodo dovrebbe essere usato per:

- Preparazione dei timer
- Recuperando i dati

- Aggiunta di listener di eventi
- Manipolazione di elementi DOM

```
componentDidMount() {
  ...
}
```

Sintassi ES6

Se il componente viene definito utilizzando la sintassi della classe ES6, non è possibile utilizzare le funzioni `getDefaultProps()` e `getInitialState()` .

Invece, dichiariamo il nostro `defaultProps` come una proprietà statica sulla classe e dichiariamo la forma dello stato e lo stato iniziale nel costruttore della nostra classe. Questi sono entrambi impostati sull'istanza della classe in fase di costruzione, prima che venga richiamata qualsiasi altra funzione del ciclo di vita di React.

Il seguente esempio dimostra questo approccio alternativo:

```
class MyReactClass extends React.Component {
  constructor(props){
    super(props);

    this.state = {
      count: this.props.initialCount
    };
  }

  upCount() {
    this.setState((prevState) => ({
      count: prevState.count + 1
    }));
  }

  render() {
    return (
      <div>
        Hello, {this.props.name}!<br />
        You clicked the button {this.state.count} times.<br />
        <button onClick={this.upCount}>Click here!</button>
      </div>
    );
  }
}

MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Sostituire `getDefaultProps()`

I valori predefiniti per i puntelli dei componenti vengono specificati impostando la proprietà

`defaultProps` della classe:

```
MyReactClass.defaultProps = {
  name: 'Bob',
  initialCount: 0
};
```

Sostituire `getInitialState()`

Il modo idiomatico per impostare lo stato iniziale del componente è impostare `this.state` nel costruttore:

```
constructor(props){
  super(props);

  this.state = {
    count: this.props.initialCount
  };
}
```

Aggiornamento del componente

`componentWillReceiveProps(nextProps)`

Questa è la **prima funzione chiamata sulle modifiche delle proprietà** .

Quando **le proprietà del componente cambiano** , React chiamerà questa funzione con le **nuove proprietà** . Puoi accedere ai vecchi oggetti di scena con *this.props* e ai nuovi oggetti di scena con *nextProps* .

Con queste variabili, puoi eseguire alcune operazioni di confronto tra vecchi e nuovi oggetti di scena, o chiamare la funzione perché una modifica di proprietà, ecc.

```
componentWillReceiveProps(nextProps){
  if (nextProps.initialCount && nextProps.initialCount > this.state.count){
    this.setState({
      count : nextProps.initialCount
    });
  }
}
```

`shouldComponentUpdate(nextProps, nextState)`

Questa è la **seconda funzione chiamata sulle modifiche delle proprietà e la prima sulle modifiche di stato** .

Per impostazione predefinita, se un altro componente / componente modifica una proprietà / uno stato del componente, **React** eseguirà il rendering di una nuova versione del componente. In questo caso, questa funzione restituisce sempre true.

È possibile ignorare questa funzione e **scegliere con maggiore precisione se il componente deve essere aggiornato o meno** .

Questa funzione è principalmente utilizzata per l' **ottimizzazione** .

Nel caso in cui la funzione restituisca **false** , la **pipeline di aggiornamento si interrompe immediatamente** .

```
componentShouldUpdate(nextProps, nextState){
  return this.props.name !== nextProps.name ||
    this.state.count !== nextState.count;
}
```

componentWillUpdate(nextProps, nextState)

Questa funzione funziona come `componentWillMount()` . **Le modifiche non sono in DOM** , quindi puoi apportare alcune modifiche appena prima dell'aggiornamento.

/! \: non puoi usare **this.setState ()** .

```
componentWillUpdate(nextProps, nextState){}
```

render()

Sono presenti alcune modifiche, quindi eseguire nuovamente il rendering del componente.

componentDidUpdate(prevProps, prevState)

Stesse cose di `componentDidMount()` : **DOM viene aggiornato** , quindi puoi fare un po 'di lavoro sul DOM qui.

```
componentDidUpdate(prevProps, prevState){}
```

Rimozione dei componenti

componentWillUnmount()

Questo metodo viene chiamato **prima che** un componente sia smontato dal DOM.

È un buon posto per eseguire operazioni di pulizia come:

- Rimozione dei listener di eventi.
- Timer di compensazione.
- Fermare le prese.
- Pulizia degli stati redux.

```
componentWillUnmount(){
  ...
```

```
}
```

Un esempio di rimozione del listener di eventi associato in `componentWillUnMount`

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

export default class SideMenu extends Component {

  constructor(props) {
    super(props);
    this.state = {
        ...
      };
    this.openMenu = this.openMenu.bind(this);
    this.closeMenu = this.closeMenu.bind(this);
  }

  componentDidMount() {
    document.addEventListener("click", this.closeMenu);
  }

  componentWillUnmount() {
    document.removeEventListener("click", this.closeMenu);
  }

  openMenu() {
    ...
  }

  closeMenu() {
    ...
  }

  render() {
    return (
      <div>
        <a
          href      = "javascript:void(0)"
          className = "closebtn"
          onClick   = {this.closeMenu}
        >
          x
        </a>
        <div>
          Some other structure
        </div>
      </div>
    );
  }
}
```

Reagire contenitore componente

Quando si crea un'applicazione React, è spesso consigliabile suddividere i componenti in base alla loro responsabilità principale, in componenti Presentational e Container.
I componenti della presentazione riguardano solo la visualizzazione dei dati, possono essere considerati e spesso implementati come funzioni che convertono un modello in una vista. In

genere non mantengono nessuno stato interno. I componenti del contenitore riguardano la gestione dei dati. Questo può essere fatto internamente attraverso il proprio stato, o agendo da intermediari con una libreria di gestione dello stato come Redux. Il componente contenitore non visualizzerà direttamente i dati, ma passerà i dati a un componente di presentazione.

```
// Container component
import React, { Component } from 'react';
import Api from 'path/to/api';

class CommentsListContainer extends Component {
    constructor() {
        super();
        // Set initial state
        this.state = { comments: [] }
    }

    componentDidMount() {
        // Make API call and update state with returned comments
        Api.getComments().then(comments => this.setState({ comments }));
    }

    render() {
        // Pass our state comments to the presentational component
        return (
            <CommentsList comments={this.state.comments} />;
        );
    }
}

// Presentational Component
const CommentsList = ({ comments }) => (
    <div>
        {comments.map(comment => (
            <div>{comment}</div>
        )}
    </div>
);

CommentsList.propTypes = {
    comments: React.PropTypes.arrayOf(React.PropTypes.string)
}
```

Chiamata al metodo del ciclo di vita in diversi stati

Questo esempio serve come complemento ad altri esempi che parlano di come utilizzare i metodi del ciclo di vita e quando verrà chiamato il metodo.

Questo esempio riepiloga quali metodi (componentWillMount, componentWillReceiveProps, ecc) verranno chiamati e in quale sequenza sarà diversa per un componente **in stati diversi** :

Quando un componente è inizializzato:

1. getDefaultProps
2. getInitialState
3. componentWillMount
4. rendere

5. componentDidMount

Quando un componente è stato modificato:

1. shouldComponentUpdate
2. componentWillUpdate
3. rendere
4. componentDidUpdate

Quando un componente ha oggetti di scena modificati:

1. componentWillReceiveProps
2. shouldComponentUpdate
3. componentWillUpdate
4. rendere
5. componentDidUpdate

Quando un componente è smontato:

1. componentWillUnmount

Leggi React Component Lifecycle online: https://riptutorial.com/it/reactjs/topic/2750/react-component-lifecycle

Capitolo 21: React Routing

Examples

Esempio di file Routes.js, seguito dall'uso del collegamento router nel componente

Inserisci un file come il seguente nella tua directory di primo livello. Definisce quali componenti renderizzare per quali percorsi

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import New from './containers/new-post';
import Show from './containers/show';

import Index from './containers/home';
import App from './components/app';

export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="posts/new" component={New} />
    <Route path="posts/:id" component={Show} />

  </Route>
);
```

Ora nel tuo index.js di livello superiore che è il tuo punto di accesso all'app, devi solo renderizzare questo componente del router in questo modo:

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, browserHistory } from 'react-router';
// import the routes component we created in routes.js
import routes from './routes';


// entry point
ReactDOM.render(
    <Router history={browserHistory} routes={routes} />
  , document.getElementById('main'));
```

Ora si tratta semplicemente di utilizzare `Link` posto di tag `<a>` nell'intera applicazione. Usando Link comunicheremo con React Router per cambiare il percorso di React Router al link specificato, che a sua volta renderà il componente corretto come definito in routes.js

```
import React from 'react';
import { Link } from 'react-router';

export default function PostButton(props) {
  return (
```

```
    <Link to={`posts/${props.postId}`}>
      <div className="post-button" >
        {props.title}
        <span>{props.tags}</span>
      </div>
    </Link>
  );
}
```

React Routing Async

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';

import Index from './containers/home';
import App from './components/app';

//for single Component lazy load use this
const ContactComponent = () => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          callback(null, require('./components/Contact')["default"]);
        }, 'Contact');
      }
    }
};

//for multiple componnets
 const groupedComponents = (pageName) => {
  return {
      getComponent: (location, callback)=> {
        require.ensure([], require => {
          switch(pageName){
            case 'about' :
                callback(null, require( "./components/about" )["default"]);
                    break ;
            case 'tos' :
                callback(null, require( "./components/tos" )["default"]);
                    break ;
          }
        }, "groupedComponents");
      }
    }
};
export default(
  <Route path="/" component={App}>
    <IndexRoute component={Index} />
    <Route path="/contact" {...ContactComponent()} />
    <Route path="/about" {...groupedComponents('about')} />
    <Route path="/tos" {...groupedComponents('tos')} />
  </Route>
);
```

Leggi React Routing online: https://riptutorial.com/it/reactjs/topic/6096/react-routing

Capitolo 22: React.createClass vs estende React.Component

Sintassi

- Caso 1: React.createClass ({})
- Caso 2: classe MyComponent estende React.Component {}

Osservazioni

`React.createClass` stato deprecato in v15.5 e dovrebbe essere rimosso in v16 . C'è un pacchetto di sostituzione drop-in per quelli che lo richiedono ancora. Gli esempi che lo utilizzano dovrebbero essere aggiornati.

Examples

Crea componente di reazione

Esploriamo le differenze di sintassi confrontando due esempi di codice.

React.createClass (deprecato)

Qui abbiamo un `const` con una classe React assegnata, con la funzione `render` successiva per completare una tipica definizione di componente base.

```
import React from 'react';

const MyComponent = React.createClass({
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Prendiamo la precedente definizione React.createClass e convertiamola per utilizzare una classe ES6.

```
import React from 'react';

class MyComponent extends React.Component {
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

In questo esempio stiamo ora utilizzando le classi ES6. Per le modifiche di React, ora creiamo una classe chiamata **MyComponent** ed estendiamo da React.Component anziché accedere direttamente a React.createClass. In questo modo, utilizziamo meno il boiler React e più JavaScript.

PS: Tipicamente questo sarebbe usato con qualcosa come Babel per compilare ES6 in ES5 per funzionare in altri browser.

Dichiarare puntelli predefiniti e PropTypes

Ci sono cambiamenti importanti nel modo in cui usiamo e dichiariamo oggetti di scena di default e i loro tipi.

React.createClass

In questa versione, la proprietà `propTypes` è un oggetto in cui è possibile dichiarare il tipo per ciascun oggetto. La proprietà `getDefaultProps` è una funzione che restituisce un oggetto per creare gli oggetti di scena iniziali.

```
import React from 'react';

const MyComponent = React.createClass({
  propTypes: {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  },
  getDefaultProps() {
    return {
      name: 'Home',
      position: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

Questa versione utilizza `propTypes` come proprietà sull'effettiva classe **MyComponent** anziché su una proprietà come parte `createClass` definizione `createClass` .

Il `getDefaultProps` ora è stato modificato in una proprietà Object nella classe denominata defaultProps, in quanto non è più una funzione "get", è solo un oggetto. Evita più React boilerplate, questo è semplicemente JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
MyComponent.propTypes = {
  name: React.PropTypes.string,
  position: React.PropTypes.number
};
MyComponent.defaultProps = {
  name: 'Home',
  position: 1
};

export default MyComponent;
```

Inoltre, esiste un'altra sintassi per `propTypes` e `defaultProps` . Questa è una scorciatoia se la tua build ha attivato gli inizializzatori delle proprietà di ES7:

```
import React from 'react';

class MyComponent extends React.Component {
  static propTypes = {
    name: React.PropTypes.string,
    position: React.PropTypes.number
  };
  static defaultProps = {
    name: 'Home',
    position: 1
  };
  constructor(props) {
    super(props);
  }
  render() {
    return (
      <div></div>
    );
  }
}
```

```
export default MyComponent;
```

Imposta stato iniziale

Ci sono dei cambiamenti nel modo in cui impostiamo gli stati iniziali.

React.createClass

Abbiamo una funzione `getInitialState` , che restituisce semplicemente un oggetto di stati iniziali.

```
import React from 'react';

const MyComponent = React.createClass({
  getInitialState () {
    return {
      activePage: 1
    };
  },
  render() {
    return (
      <div></div>
    );
  }
});

export default MyComponent;
```

React.Component

In questa versione dichiariamo tutto lo stato come una semplice **proprietà di inizializzazione nel costruttore** , invece di usare la funzione `getInitialState` . Sembra meno "React API" guidato poiché questo è semplicemente JavaScript.

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      activePage: 1
    };
  }
  render() {
    return (
      <div></div>
    );
  }
}

export default MyComponent;
```

mixins

Possiamo usare `mixins` solo con il modo in cui React.createClass.

React.createClass

In questa versione possiamo aggiungere `mixins` ai componenti usando la proprietà mixins che prende una matrice di mixin disponibili. Questi estendono quindi la classe del componente.

```
import React from 'react';

var MyMixin = {
  doSomething() {

  }
};
const MyComponent = React.createClass({
  mixins: [MyMixin],
  handleClick() {
    this.doSomething(); // invoke mixin's method
  },
  render() {
    return (
      <button onClick={this.handleClick}>Do Something</button>
    );
  }
});

export default MyComponent;
```

React.Component

I mixaggi di reazione non sono supportati quando si usano componenti React scritti in ES6. Inoltre, non avranno supporto per le classi ES6 in React. Il motivo è che sono considerati dannosi .

"questo" Contesto

L'utilizzo di React.createClass associa automaticamente `this` contesto (valori) correttamente, ma non è il caso quando si utilizzano le classi ES6.

React.createClass

Nota la dichiarazione `onClick` con il metodo `this.handleClick` associato. Quando viene chiamato questo metodo, React applicherà il contesto di esecuzione corretto a `handleClick` .

```
import React from 'react';
```

```
const MyComponent = React.createClass({
  handleClick() {
    console.log(this); // the React Component instance
  },
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
});

export default MyComponent;
```

React.Component

Con le classi ES6 `this` è `null` per impostazione predefinita, le proprietà della classe non si associano automaticamente all'istanza della classe React (componente).

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // null
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Ci sono alcuni modi in cui possiamo legare il diritto a `this` contesto.

Caso 1: legatura in linea:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick.bind(this)}></div>
```

```
    );
  }
}

export default MyComponent;
```

Caso 2: rilegatura nel costruttore della classe

Un altro approccio sta cambiando il contesto di `this.handleClick` all'interno del `constructor` . In questo modo evitiamo la ripetizione in linea. Considerato da molti come un approccio migliore che evita di toccare JSX affatto:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

Caso 3: utilizzare la funzione anonima ES6

Puoi anche utilizzare la funzione anonima ES6 senza doverli associare esplicitamente:

```
import React from 'react';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
  }
  handleClick = () => {
    console.log(this); // the React Component instance
  }
  render() {
    return (
      <div onClick={this.handleClick}></div>
    );
  }
}

export default MyComponent;
```

ES6 / Reagire con "questa" parola chiave con ajax per ottenere dati dal server

```
import React from 'react';

class SearchEs6 extends React.Component{
    constructor(props) {
        super(props);
        this.state = {
            searchResults: []
        };
    }

    showResults(response){
        this.setState({
            searchResults: response.results
        })
    }

    search(url){
        $.ajax({
            type: "GET",
            dataType: 'jsonp',
            url: url,
            success: (data) => {
                this.showResults(data);
            },
            error: (xhr, status, err) => {
                console.error(url, status, err.toString());
            }
        });
    }

    render() {
        return (
            <div>
                <SearchBox search={this.search.bind(this)} />
                <Results searchResults={this.state.searchResults} />
            </div>
        );
    }
}
```

Leggi React.createClass vs estende React.Component online:
https://riptutorial.com/it/reactjs/topic/6371/react-createclass-vs-estende-react-component

Capitolo 23: Reagire alla chiamata AJAX

Examples

Richiesta GET HTTP

A volte un componente deve eseguire il rendering di alcuni dati da un endpoint remoto (ad esempio un'API REST). Una pratica standard è quella di effettuare tali chiamate nel metodo `componentDidMount` .

Ecco un esempio, usando superagent come helper AJAX:

```
import React from 'react'
import request from 'superagent'

class App extends React.Component {
  constructor () {
    super()
    this.state = {}
  }
  componentDidMount () {
    request
      .get('/search')
      .query({ query: 'Manny' })
      .query({ range: '1..5' })
      .query({ order: 'desc' })
      .set('API-Key', 'foobar')
      .set('Accept', 'application/json')
      .end((err, resp) => {
        if (!err) {
          this.setState({someData: resp.text})
        }
      })
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

Una richiesta può essere avviata invocando il metodo appropriato sull'oggetto `request` , quindi chiamando `.end()` per inviare la richiesta. L'impostazione dei campi di intestazione è semplice, invoca `.set()` con un nome e un valore di campo.

Il metodo `.query()` accetta oggetti, che quando vengono utilizzati con il metodo GET formeranno una stringa di query. Quanto segue produrrà il percorso `/search?query=Manny&range=1..5&order=desc` .

Richieste POST

```
request.post('/user')
  .set('Content-Type', 'application/json')
  .send('{"name":"tj","pet":"tobi"}')
  .end(callback)
```

Vedi i documenti Superagenti per maggiori dettagli.

Ajax in React senza una libreria di terze parti, ovvero con VanillaJS

Quanto segue funzionerebbe in IE9 +

```
import React from 'react'

class App extends React.Component {
  constructor () {
    super()
    this.state = {someData: null}
  }
  componentDidMount () {
    var request = new XMLHttpRequest();
    request.open('GET', '/my/url', true);

    request.onload = () => {
      if (request.status >= 200 && request.status < 400) {
        // Success!
        this.setState({someData: request.responseText})
      } else {
        // We reached our target server, but it returned an error
        // Possibly handle the error by changing your state.
      }
    };

    request.onerror = () => {
      // There was a connection error of some sort.
      // Possibly handle the error by changing your state.
    };

    request.send();
  },
  render() {
    return (
      <div>{this.state.someData || 'waiting for response...'}</div>
    )
  }
}

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

Richiesta HTTP GET e looping dei dati

L'esempio seguente mostra come un set di dati ottenuti da un'origine remota può essere reso in un componente.

Facciamo una richiesta AJAX usando `fetch` , che è compilato nella maggior parte dei browser. Utilizzare un polyfill di `fetch` in produzione per supportare i browser più vecchi. È inoltre possibile utilizzare qualsiasi altra libreria per effettuare richieste (ad esempio assios , SuperAgent o anche

solo Javascript).

Impostiamo i dati che riceviamo come stato del componente, in modo che possiamo accedervi all'interno del metodo di rendering. Lì, passiamo in rassegna i dati usando la `map` . Non dimenticare di aggiungere sempre un attributo `key` (o prop) univoco all'elemento loop, che è importante per le prestazioni di rendering di React.

```
import React from 'react';

class Users extends React.Component {
  constructor() {
    super();
    this.state = { users: [] };
  }

  componentDidMount() {
    fetch('/api/users')
      .then(response => response.json())
      .then(json => this.setState({ users: json.data }));
  }

  render() {
    return (
      <div>
        <h1>Users</h1>
        {
          this.state.users.length == 0
            ? 'Loading users...'
            : this.state.users.map(user => (
              <figure key={user.id}>
                <img src={user.avatar} />
                <figcaption>
                  {user.name}
                </figcaption>
              </figure>
            ))
        }
      </div>
    );
  }
}

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

Esempio di lavoro su JSBin .

Leggi Reagire alla chiamata AJAX online: https://riptutorial.com/it/reactjs/topic/6432/reagire-alla-chiamata-ajax

Capitolo 24: Reagire con Redux

introduzione

Redux è diventato lo status quo per la gestione dello stato a livello di applicazione sul front-end in questi giorni, e coloro che lavorano su "applicazioni su larga scala" spesso lo giurano. Questo argomento spiega perché e come utilizzare la libreria di gestione dello stato, Redux, nelle applicazioni React.

Osservazioni

Sebbene l'architettura basata su componenti di React sia fantastica per scomporre l'applicazione in piccoli pezzi modulari e incapsulati, introduce alcune sfide per la gestione dello stato dell'applicazione nel suo complesso. Il tempo di usare Redux è quando hai bisogno di visualizzare gli stessi dati su più di un componente o pagina (pseudonimo). A quel punto non è più possibile memorizzare i dati nelle variabili locali di un componente o dell'altro e l'invio di messaggi tra i componenti diventa rapidamente un disastro. Con Redux i componenti sono tutti abbonati agli stessi dati condivisi nello store e quindi lo stato può essere facilmente riflesso in modo coerente nell'intera applicazione.

Examples

Utilizzando Connect

Creare un negozio Redux con *createStore* .

```
import { createStore } from 'redux'
import todoApp from './reducers'
let store = createStore(todoApp, { inistialStateVariable: "derp"})
```

Usa *connetti* per connettere il componente all'archivio Redux e tirare i puntelli da negozio a componente.

```
import { connect } from 'react-redux'

const VisibleTodoList = connect(
  mapStateToProps,
  mapDispatchToProps
)(TodoList)

export default VisibleTodoList
```

Definire le azioni che consentono ai componenti di inviare messaggi all'archivio di Redux.

```
/*
 * action types
```

```
 */

export const ADD_TODO = 'ADD_TODO'

export function addTodo(text) {
  return { type: ADD_TODO, text }
}
```

Gestire questi messaggi e creare un nuovo stato per l'archivio nelle funzioni di riduzione.

```
function todoApp(state = initialState, action) {
  switch (action.type) {
    case SET_VISIBILITY_FILTER:
      return Object.assign({}, state, {
        visibilityFilter: action.filter
      })
    default:
      return state
  }
}
```

Leggi Reagire con Redux online: https://riptutorial.com/it/reactjs/topic/10856/reagire-con-redux

Capitolo 25: Soluzioni di interfaccia utente

introduzione

Diciamo che ci ispiriamo ad alcune idee delle moderne interfacce utente usate nei programmi e le convertiamo in componenti React. Questo è l'argomento di " **Soluzioni di interfaccia utente** ". L'attribuzione è appretiata.

Examples

Riquadro di base

```
import React from 'react';

class Pane extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'section', this.props
        );
    }
}
```

Pannello

```
import React from 'react';

class Panel extends React.Component {
    constructor(props) {
        super(props);
    }

    render(...elements) {
        var props = Object.assign({
            className: this.props.active ? 'active' : '',
            tabIndex: -1
        }, this.props);

        var css = this.css();
        if (css != '') {
            elements.unshift(React.createElement(
                'style', null,
                css
            ));
        }

        return React.createElement(
            'div', props,
            ...elements
```

```
        );
    }

    static title() {
        return '';
    }
    static css() {
        return '';
    }
}
```

Le principali differenze dal pannello semplice sono:

- il pannello è focalizzato, ad esempio, quando viene richiamato dallo script o cliccato con il mouse;
- il pannello ha il `title` metodo statico per componente, quindi può essere esteso da un altro componente del pannello con `title` sottoposto a override (la ragione è che la funzione può essere richiamata di nuovo sul rendering per scopi di localizzazione, ma nei limiti di questo esempio il `title` non ha senso) ;
- può contenere singoli fogli di stile dichiarati nel metodo statico `css` (puoi precaricare il contenuto del file da `PANEL.css`).

linguetta

```
import React from 'react';

class Tab extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        var props = Object.assign({
            className: this.props.active ? 'active' : ''
        }, this.props);
        return React.createElement(
            'li', props,
            React.createElement(
                'span', props,
                props.panelClass.title()
            )
        );
    }
}
```

`panelClass` proprietà `panelClass` dell'istanza `Tab` deve contenere la classe del *pannello* utilizzata per la descrizione.

PanelGroup

```
import React from 'react';
import Tab from './Tab.js';

class PanelGroup extends React.Component {
```

```
    constructor(props) {
        super(props);
        this.setState({
            panels: props.panels
        });
    }

    render() {
        this.tabSet = [];
        this.panelSet = [];
        for (let panelData of this.state.panels) {
            var tabIsActive = this.state.activeTab == panelData.name;
            this.tabSet.push(React.createElement(
                Tab, {
                    name: panelData.name,
                    active: tabIsActive,
                    panelClass: panelData.class,
                    onMouseDown: () => this.openTab(panelData.name)
                }
            ));
            this.panelSet.push(React.createElement(
                panelData.class, {
                    id: panelData.name,
                    active: tabIsActive,
                    ref: tabIsActive ? 'activePanel' : null
                }
            ));
        }
        return React.createElement(
            'div', { className: 'PanelGroup' },
            React.createElement(
                'nav', null,
                React.createElement(
                    'ul', null,
                    ...this.tabSet
                )
            ),
            ...this.panelSet
        );
    }

    openTab(name) {
        this.setState({ activeTab: name });
        this.findDOMNode(this.refs.activePanel).focus();
    }
}
```

`panels` proprietà `PanelGroup` dell'istanza `PanelGroup` deve contenere una matrice con oggetti. Ogni oggetto lì dichiara importanti dati sui pannelli:

- **`name`** - identificatore del pannello usato dallo script del controllore;
- **`class`** - **`class`** pannello.

Non dimenticare di impostare proprietà `activeTab` sul nome della scheda necessaria.

Una precisazione

Quando il tab è disattivato, il pannello necessario sta ottenendo il nome della classe `active`

sull'elemento DOM (significa che sarà visibile) ed è ora focalizzato.

Esempio di vista con `PanelGroup`s

```
import React from 'react';
import Pane from './components/Pane.js';
import Panel from './components/Panel.js';
import PanelGroup from './components/PanelGroup.js';

class MainView extends React.Component {
    constructor(props) {
        super(props);
    }

    render() {
        return React.createElement(
            'main', null,
            React.createElement(
                Pane, { id: 'common' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'console',
                                panelClass: ConsolePanel
                            },
                            {
                                name: 'figures',
                                panelClass: FiguresPanel
                            }
                        ],
                        activeTab: 'console'
                    }
                )
            ),
            React.createElement(
                Pane, { id: 'side' },
                React.createElement(
                    PanelGroup, {
                        panels: [
                            {
                                name: 'properties',
                                panelClass: PropertiesPanel
                            }
                        ],
                        activeTab: 'properties'
                    }
                )
            )
        );
    }
}

class ConsolePanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Console';
```

```
    }
}

class FiguresPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Figures';
    }
}

class PropertiesPanel extends Panel {
    constructor(props) {
        super(props);
    }

    static title() {
        return 'Properties';
    }
}
```

Leggi Soluzioni di interfaccia utente online: https://riptutorial.com/it/reactjs/topic/8112/soluzioni-di-interfaccia-utente

Capitolo 26: Stato in React

Examples

Stato di base

Lo stato nei componenti React è essenziale per gestire e comunicare i dati nella tua applicazione. È rappresentato come oggetto JavaScript e ha un ambito a *livello di componente* , può essere considerato come i dati privati del componente.

Nell'esempio seguente stiamo definendo alcuni stati iniziali nella funzione di `constructor` del nostro componente e ne facciamo uso nella funzione di `render` .

```
class ExampleComponent extends React.Component {
  constructor(props){
    super(props);

    // Set-up our initial state
    this.state = {
      greeting: 'Hiya Buddy!'
    };
  }

  render() {
    // We can access the greeting property through this.state
    return(
      <div>{this.state.greeting}</div>
    );
  }
}
```

setState ()

Il modo principale con cui si effettuano gli aggiornamenti dell'interfaccia utente alle applicazioni React è tramite una chiamata alla funzione `setState()` . Questa funzione eseguirà *un'unione superficiale* tra il nuovo stato che fornisci e lo stato precedente e attiverà un re-rendering del tuo componente e di tutti i decedenti.

parametri

1. `updater` : può essere un oggetto con un numero di coppie chiave-valore che devono essere unite nello stato o una funzione che restituisce tale oggetto.
2. `callback (optional)` : una funzione che verrà eseguita dopo che `setState()` è stato eseguito con successo. A causa del fatto che le chiamate a `setState()` non sono garantite da React per essere atomiche, a volte può essere utile se si desidera eseguire qualche azione dopo che si è `setState()` che `setState()` è stato eseguito correttamente.

Uso:

Il metodo `setState` accetta un argomento `updater` che può essere sia un oggetto con un numero di coppie chiave-valore che devono essere unite nello stato, sia una funzione che restituisce tale oggetto calcolato da `prevState` e `props` .

Usando `setState()` con un oggetto come programma di `updater`

```
//
// An example ES6 style component, updating the state on a simple button click.
// Also demonstrates where the state can be set directly and where setState should be used.
//
class Greeting extends React.Component {
    constructor(props) {
        super(props);
        this.click = this.click.bind(this);
        // Set initial state (ONLY ALLOWED IN CONSTRUCTOR)
        this.state = {
            greeting: 'Hello!'
        };
    }
    click(e) {
        this.setState({
              greeting: 'Hello World!'
        });
    }
    render() {
        return(
            <div>
                <p>{this.state.greeting}</p>
                <button onClick={this.click}>Click me</button>
            </div>
        );
    }

}
```

Utilizzo di `setState()` con una funzione come `updater`

```
//
// This is most often used when you want to check or make use
// of previous state before updating any values.
//

this.setState(function(previousState, currentProps) {
  return {
    counter: previousState.counter + 1
  };
});
```

Questo può essere più sicuro dell'uso di un argomento oggetto in cui vengono utilizzate più chiamate a `setState()` , poiché più chiamate possono essere raggruppate insieme da React e

eseguite contemporaneamente, ed è l'approccio preferito quando si usano gli oggetti di scena attuali per impostare lo stato.

```
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
this.setState({ counter: this.state.counter + 1 });
```

Queste chiamate possono essere raggruppate insieme da React usando `Object.assign()` , facendo in modo che il contatore venga incrementato di 1 anziché di 3.

L'approccio funzionale può anche essere utilizzato per spostare la logica di impostazione dello stato al di fuori dei componenti. Ciò consente l'isolamento e il riutilizzo della logica di stato.

```
// Outside of component class, potentially in another file/module

function incrementCounter(previousState, currentProps) {
    return {
        counter: previousState.counter + 1
    };
}

// Within component

this.setState(incrementCounter);
```

Chiamando `setState()` con un oggetto e una funzione di callback

```
//
// 'Hi There' will be logged to the console after setState completes
//

this.setState({ name: 'John Doe' }, console.log('Hi there'));
```

Antipattern comune

Non dovresti salvare `props` nello `state` . È considerato un anti-modello . Per esempio:

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            url: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
```

```
            url: this.props.url + '/days=?' + e.target.value
        });
    }

    componentWillMount() {
        this.setState({url: this.props.url});
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.state.url}
            </div>
        )
    }
}
```

L' `url` prop viene salvato su `state` e quindi modificato. Invece, scegliere di salvare le modifiche in uno stato, quindi creare il percorso completo utilizzando sia lo `state` che gli `props` :

```
export default class MyComponent extends React.Component {
    constructor() {
        super();

        this.state = {
            days: ''
        }

        this.onChange = this.onChange.bind(this);
    }

    onChange(e) {
        this.setState({
            days: e.target.value
        });
    }

    render() {
        return (
            <div>
                <input defaultValue={2} onChange={this.onChange} />

                URL: {this.props.url + '/days?=' + this.state.days}
            </div>
        )
    }
}
```

Questo perché in un'applicazione React vogliamo avere un'unica fonte di verità, ovvero tutti i dati sono responsabilità di un singolo componente e di un solo componente. È responsabilità di questo componente archiviare i dati nel suo stato e distribuire i dati ad altri componenti tramite oggetti di scena.

Nel primo esempio, sia la classe MyComponent che la sua parent mantengono "url" all'interno del loro stato. Se aggiorniamo lo stato.url in MyComponent, queste modifiche non si riflettono nel

genitore. Abbiamo perso la nostra unica fonte di verità, e diventa sempre più difficile monitorare il flusso di dati attraverso la nostra applicazione. Confrontalo con il secondo esempio: l'url viene mantenuto solo nello stato del componente principale e utilizzato come oggetto di supporto in MyComponent, pertanto manteniamo un'unica fonte di verità.

Stato, eventi e controlli gestiti

Ecco un esempio di un componente React con un campo di input "gestito". Ogni volta che cambia il valore del campo di input, viene chiamato un gestore di eventi che aggiorna lo stato del componente con il nuovo valore del campo di input. La chiamata a `setState` nel gestore eventi attiverà una chiamata per eseguire il `render` dell'aggiornamento del componente nella dom.

```
import React from 'react';
import {render} from 'react-dom';


class ManagedControlDemo extends React.Component {

  constructor(props){
    super(props);
    this.state = {message: ""};
  }

  handleChange(e){
    this.setState({message: e.target.value});
  }

  render() {
    return (
      <div>
        <legend>Type something here</legend>
          <input
            onChange={this.handleChange.bind(this)}
            value={this.state.message}
            autoFocus />
        <h1>{this.state.message}</h1>
      </div>
   );
  }
}


render(<ManagedControlDemo/>, document.querySelector('#app'));
```

È molto importante notare il comportamento di runtime. Ogni volta che un utente modifica il valore nel campo di input

- `handleChange` sarà chiamato e così
- `setState` sarà chiamato e così
- `render` sarà chiamato

Quiz pop, dopo aver digitato un carattere nel campo di input, quali elementi DOM cambiano

1. tutti questi - il livello più alto div, legend, input, h1
2. solo l'input e h1

3. Niente
4. che cos'è un DOM?

Puoi sperimentare di più qui per trovare la risposta

Leggi Stato in React online: https://riptutorial.com/it/reactjs/topic/1816/stato-in-react

Capitolo 27: Strumenti di reazione

Examples

link

Luoghi da trovare Componenti e librerie di React;

- Catalogo dei componenti di reazione
- JS.coach

Leggi Strumenti di reazione online: https://riptutorial.com/it/reactjs/topic/6595/strumenti-di-reazione

Capitolo 28: Usando ReactJS in modo Flux

introduzione

È molto utile utilizzare l'approccio Flux, quando è prevista la crescita dell'applicazione con ReactJS sul frontend, a causa delle strutture limitate e di un po 'di nuovo codice per rendere più facili le modifiche dello stato in fase di esecuzione.

Osservazioni

Flux è l'architettura dell'applicazione che Facebook utilizza per creare applicazioni web lato client. Completa i componenti di visualizzazione composable di React utilizzando un flusso di dati unidirezionale. È più un pattern piuttosto che un framework formale, e puoi iniziare a usare Flux immediatamente senza un sacco di nuovo codice.

Le applicazioni di flusso hanno tre parti principali: *il dispatcher* , *i negozi* e *le viste* (componenti React). Questi non dovrebbero essere confusi con Model-View-Controller. I controller esistono in un'applicazione Flux, ma sono viste del controllore - viste spesso trovate nella parte superiore della gerarchia che recuperano i dati dai negozi e li trasmettono ai loro figli. Inoltre, i creatori di azioni - metodi helper del dispatcher - vengono utilizzati per supportare un'API semantica che descrive tutte le modifiche possibili nell'applicazione. Può essere utile considerarli come una quarta parte del ciclo di aggiornamento di Flux.

Flux evita MVC in favore di un flusso di dati unidirezionale. Quando un utente interagisce con una visualizzazione React, la visualizzazione propaga un'azione attraverso un dispatcher centrale, ai vari negozi che contengono i dati dell'applicazione e la logica aziendale, che aggiorna tutte le viste interessate. Ciò funziona particolarmente bene con lo stile di programmazione dichiarativa di React, che consente allo store di inviare aggiornamenti senza specificare come passare le viste tra stati.

Examples

Flusso di dati

> Questo è il profilo di una panoramica completa.

Il modello di flusso presuppone l'uso del flusso di dati unidirezionale.

1. **Azione** : oggetto semplice che descrive il `type` azione e altri dati di input.
2. **Dispatcher** - ricevitore a singola azione e controller di callback. Immagina che sia l'hub centrale della tua applicazione.
3. **Store** : contiene lo stato e la logica dell'applicazione. Registra la richiamata nel dispatcher ed emette un evento da visualizzare quando si è verificata la modifica al livello dati.

4. **Visualizza** - Componente React che riceve eventi di modifica e dati dall'archivio. Provoca il re-rendering quando qualcosa è cambiato.

   A partire dal flusso di dati Flux, le viste possono anche **creare azioni** e passarle al dispatcher per le interazioni dell'utente.

ripristinata

Per renderlo più chiaro, possiamo iniziare dalla fine.

- I componenti React diversi (*viste*) ottengono i dati da diversi negozi sulle modifiche apportate.

  Poche componenti possono essere chiamate **controller-views** , perché forniscono il codice della colla per ottenere i dati dai negozi e per passare i dati lungo la catena dei loro discendenti. Le visualizzazioni del controller rappresentano qualsiasi sezione significativa della pagina.

- *Gli store* possono essere commentati come callback che mettono a confronto il tipo di azione e altri dati di input per la logica aziendale dell'applicazione.

- *Dispatcher* è un contenitore di azioni e ricevitore di azioni comuni.

- *Le azioni* non sono altro che semplici oggetti con la proprietà di `type` richiesta.

  Precedentemente, ti consigliamo di utilizzare le costanti per i tipi di azione e i metodi di supporto (chiamati **creatori di azioni**).

Leggi Usando ReactJS in modo Flux online: https://riptutorial.com/it/reactjs/topic/8158/usando-reactjs-in-modo-flux

Capitolo 29: Utilizzo di React with Flow

introduzione

Come utilizzare il controllo del tipo di flusso per controllare i tipi nei componenti React.

Osservazioni

Flusso | Reagire

Examples

Utilizzo di Flow per controllare i tipi di prop di componenti funzionali stateless

```
type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

const AppContainer =
  ({ posts, dispatch, children }: Props) => (
    <div className="main-app">
      <Header {...{ posts, dispatch }} />
      {children}
    </div>
  )
```

Utilizzo di Flow per controllare i tipi di puntello

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

type Props = {
  posts: Array<Article>,
  dispatch: Function,
  children: ReactElement
}

class Posts extends Component {
  props: Props;

  render () {
    // rest of the code goes here
  }
}
```

Leggi Utilizzo di React with Flow online: https://riptutorial.com/it/reactjs/topic/7918/utilizzo-di-react-with-flow

Capitolo 30: Utilizzo di ReactJS con jQuery

Examples

ReactJS con jQuery

Innanzitutto, devi importare la libreria jquery. Dobbiamo anche importare findDOmNode mentre manipoleremo dom. E ovviamente stiamo importando anche React.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';
```

Stiamo impostando una funzione di freccia "handleToggle" che verrà attivata quando verrà cliccata un'icona. Stiamo solo mostrando e nascondendo un div con un nome di riferimento 'toggle' onClick su un'icona.

```
handleToggle = () => {
    const el = findDOMNode(this.refs.toggle);
    $(el).slideToggle();
};
```

Ora impostiamo la denominazione di riferimento 'toggle'

```
<ul className="profile-info additional-profile-info-list" ref="toggle">
  <li>
    <span className="info-email">Office Email</span>   me@shuvohabib.com
  </li>
</ul>
```

L'elemento div in cui verrà generato il comando "handleToggle" su onClick.

```
 <div className="ellipsis-click" onClick={this.handleToggle}>
   <i className="fa-ellipsis-h"/>
 </div>
```

Lascia che riveda il codice completo qui sotto, come sembra.

```
import React from 'react';
import { findDOMNode } from 'react-dom';
import $ from 'jquery';

export default class FullDesc extends React.Component {
    constructor() {
        super();
    }

    handleToggle = () => {
        const el = findDOMNode(this.refs.toggle);
        $(el).slideToggle();
```

```
    };

    render() {
        return (
            <div className="long-desc">
                <ul className="profile-info">
                    <li>
                        <span className="info-title">User Name : </span> Shuvo Habib
                    </li>
                </ul>

                <ul className="profile-info additional-profile-info-list" ref="toggle">
                    <li>
                        <span className="info-email">Office Email</span> me@shuvohabib.com
                    </li>
                </ul>

                <div className="ellipsis-click" onClick={this.handleToggle}>
                    <i className="fa-ellipsis-h"/>
                </div>
            </div>
        );
    }
}
```

Abbiamo chiuso! Questo è il modo, come possiamo usare **jQuery nel** componente **React** .

Leggi Utilizzo di ReactJS con jQuery online: https://riptutorial.com/it/reactjs/topic/6009/utilizzo-di-reactjs-con-jquery

Capitolo 31: Utilizzo di ReactJS con Typescript

Examples

Componente ReactJS scritto in Typescript

In realtà puoi usare i componenti di ReactJS in Typescript come nell'esempio di facebook. Basta sostituire l'estensione del file 'jsx' a 'tsx':

```
//helloMessage.tsx:
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

Ma per sfruttare appieno la funzione principale di Typescript (controllo del tipo statico), è necessario fare alcune cose:

1) converti l'esempio React.createClass in classe ES6:

```
//helloMessage.tsx:
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

2) aggiungere i puntelli e le interfacce di stato:

```
interface IHelloMessageProps {
    name:string;
}

interface IHelloMessageState {
  //empty in our case
}

class HelloMessage extends React.Component<IHelloMessageProps, IHelloMessageState> {
  constructor(){
    super();
  }
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

Ora Typescript mostrerà un errore se il programmatore dimenticherà di passare oggetti di scena. O se hanno aggiunto oggetti di scena che non sono definiti nell'interfaccia.

Stateless React Components in Typescript

Reagire a componenti che sono pure funzioni dei loro oggetti di scena e non richiedono alcuno stato interno può essere scritto come funzione JavaScript invece di usare la sintassi della classe standard, come:

```
import React from 'react'

const HelloWorld = (props) => (
    <h1>Hello, {props.name}!</h1>
);
```

Lo stesso può essere ottenuto in Typescript usando la classe `React.SFC` :

```
import * as React from 'react';

class GreeterProps {
   name: string
}

const Greeter : React.SFC<GreeterProps> = props =>
    <h1>Hello, {props.name}!</h1>;
```

Si noti che, il nome `React.SFC` è un alias per `React.StatelessComponent` Quindi, può essere utilizzato.

Installazione e configurazione

Per usare dattiloscritto con reagire in un progetto di nodo, è necessario prima avere una directory di progetto inizializzata con npm. Per inizializzare la directory con `npm init`

Installazione via npm o filato

È possibile installare React utilizzando npm attenendosi alla seguente procedura:

```
npm install --save react react-dom
```

Facebook ha rilasciato il proprio gestore di pacchetti denominato Yarn , che può essere utilizzato anche per installare React. Dopo aver installato Yarn è sufficiente eseguire questo comando:

```
yarn add react react-dom
```

Puoi quindi utilizzare React nel tuo progetto esattamente nello stesso modo in cui avevi installato React via npm.

Installare le definizioni del tipo di reazione in Typescript 2.0+

Per compilare il codice usando dattiloscritto, aggiungi / installa i file di definizione del tipo usando

npm o filato.

```
npm install --save-dev @types/react @types/react-dom
```

oppure, usando il filo

```
yarn add --dev @types/react @types/react-dom
```

Installare le definizioni del tipo di reazione nelle versioni precedenti di Typescript

Devi usare un pacchetto separato chiamato tsd

```
tsd install react react-dom --save
```

Aggiunta o modifica della configurazione di Typescript

Per usare JSX , una lingua che mescola javascript con html / xml, devi cambiare la configurazione del compilatore dattiloscritto. Nel file di configurazione del dattiloscritto del progetto (solitamente chiamato `tsconfig.json`), sarà necessario aggiungere l'opzione JSX come:

```
"compilerOptions": {
    "jsx": "react"
},
```

Quella opzione del compilatore dice fondamentalmente al compilatore dattiloscritto di tradurre i tag JSX nel codice in chiamate di funzione javascript.

Per evitare che il compilatore di dattiloscritto converta JSX in semplici chiamate di funzione javascript, utilizzare

```
"compilerOptions": {
    "jsx": "preserve"
},
```

Componenti senza stato e senza proprietà

Il componente di reazione più semplice senza stato e senza proprietà può essere scritto come:

```
import * as React from 'react';

const Greeter = () => <span>Hello, World!</span>
```

Quel componente, tuttavia, non può accedere a `this.props` poiché il typescript non può stabilire se si tratta di un componente react. Per accedere ai suoi oggetti di scena, utilizzare:

```
import * as React from 'react';

const Greeter: React.SFC<{}> = props => () => <span>Hello, World!</span>
```

Anche se il componente non ha proprietà definite in modo esplicito, ora può accedere a **`props.children`** poiché tutti i componenti hanno intrinsecamente figli.

Un altro uso analogo di componenti senza stato e senza proprietà si trova nel semplice modello di pagina. Di seguito è riportato un componente di `Page` semplice, ad esempio, presupponendo che ci siano componenti ipotetici `Container` , `NavTop` e `NavBottom` già presenti nel progetto:

```
import * as React from 'react';

const Page: React.SFC<{}> = props => () =>
    <Container>
        <NavTop />
        {props.children}
        <NavBottom />
    </Container>

const LoginPage: React.SFC<{}> = props => () =>
    <Page>
        Login Pass: <input type="password" />
    </Page>
```

In questo esempio, il componente `Page` può essere successivamente utilizzato da qualsiasi altra pagina effettiva come modello di base.

Leggi Utilizzo di ReactJS con Typescript online: https://riptutorial.com/it/reactjs/topic/1419/utilizzo-di-reactjs-con-typescript

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con React	Adam, Adrián Daraš, Alex, Alex Young, Anuj, Bart Riordan, Cassidy, Community, Daksh Gupta, Dave Kaye, diabolicfreak, DMan, Donald, Everettss, Gianluca Esposito, himanshuIIITian, hyde, Ilya Lyamkin, Inanc Gumus, ivarni, jengeb, jolyonruss, Jon Chan, JordanHendrix, juandemarco, Kaloyan Kosev, Konstantin Grushetsky, Maksim, Marty, MaxPRafferty, Md. Nahiduzzaman Rose, Md.Sifatul Islam, Ming Soon, MMachinegun, Nick Bartlett, orvi, paqash, Prakash, rossipedia, Shabin Hashim, Simplans, Sunny R Gupta, TheShadowbyte, Timo, Tushar Khanna, user2314737
2	Come configurare un webpack di base, reagire e fare i babel dell'ambiente	Bart Riordan, Tien Do, Zac Braddy
3	Come e perché usare le chiavi in React	Sammy I.
4	componenti	akashrajkn, Anuj, Bart Riordan, Bond, Brandon Roberts, Denis Ivanov, Diego V, DMan, Evan Hammer, Everettss, goldbullet, GordyD, hmnzr, Ilya Lyamkin, ivarni, Jagadish Upadhyay, jbmartinez, John Ruddell, jolyonruss, Jon Chan, jonathangoodman, JordanHendrix, justabuzz, k170, Kousha, Kyle Richardson, m_callens, Maayan Glikser, Michael Peyper, Paul Graffam, philpee2, QoP, Radu Brehar, Sai Vikas, sjmarshy , Timo, Vlad Bezden, WooCaSh, Zakaria Ridouh, zurfyx
5	Componenti dell'ordine superiore	Dennis Stücken
6	Componenti funzionali stateless	Adam, Mark Lapierre, Mayank Shukla, Valter Júnior
7	Comunicare tra i componenti	Random User
8	Comunicazione tra componenti	David, Kaloyan Kosev
9	Forme di reazione	promisified

10	Forms e User Input	Everettss, Henrik Karlsson, ivarni, Timo
11	Impostazione dell'ambiente reattivo	ghostffcode, Tien Do
12	Installazione	Rene R, Ruairi O'Brien
13	Installazione di React, Webpack e Typescript	Aron
14	Introduzione al rendering lato server	Adrián Daraš, MauroPorrasP
15	JSX	Kaloyan Kosev, Ming Soon
16	Le chiavi reagiscono	Dennis Stücken, thibmaek
17	Prestazione	Aditya Singh, lustoykov, thibmaek
18	Puntelli in React	Ahmad, Anuj, Danillo Corvalan, Everettss, Faktor 10, Fellow Stranger, hansn, Ilya Lyamkin, Jack7, Jagadish Upadhyay, JimmyLv, MaxPRafferty, QoP, Sergii Bishyr, vintproykt, WitVault , zbynour
19	React Boilerplate [React + Babel + Webpack]	Mihir, parlad neupane, Tien Do
20	React Component Lifecycle	Alex Young, Alexg2195, Anuj, Ashari, Everettss, F. Kauder, irrigator, John Ruddell, QoP, Salman Saleem, Saravana, Siddharth, skav, Timo, ultrasamad, Vivian, WitVault
21	React Routing	abhirathore2006, Robeen
22	React.createClass vs estende React.Component	Kaloyan Kosev, leonardoborges, Michael Peyper, pwolaq, Qianyue, sqzaman
23	Reagire alla chiamata AJAX	adamboro, Fabian Schultz, Jason Bourne, lifeiscontent, McGrady, Sunny R Gupta
24	Reagire con Redux	Jim
25	Soluzioni di interfaccia utente	vintproykt
26	Stato in React	Alex Young, Alexander, Brad Colthurst, Everettss, Kousha, Kyle Richardson, QoP, skav, Timo

27	Strumenti di reazione	brillout
28	Usando ReactJS in modo Flux	vintproykt
29	Utilizzo di React with Flow	JimmyLv, lifeiscontent, Rifat, Rory O'Kane
30	Utilizzo di ReactJS con jQuery	Kousha, Shuvo Habib
31	Utilizzo di ReactJS con Typescript	Everettss, John Ruddell, kevgathuku, Leone, Rajab Shakirov