Giorno 1

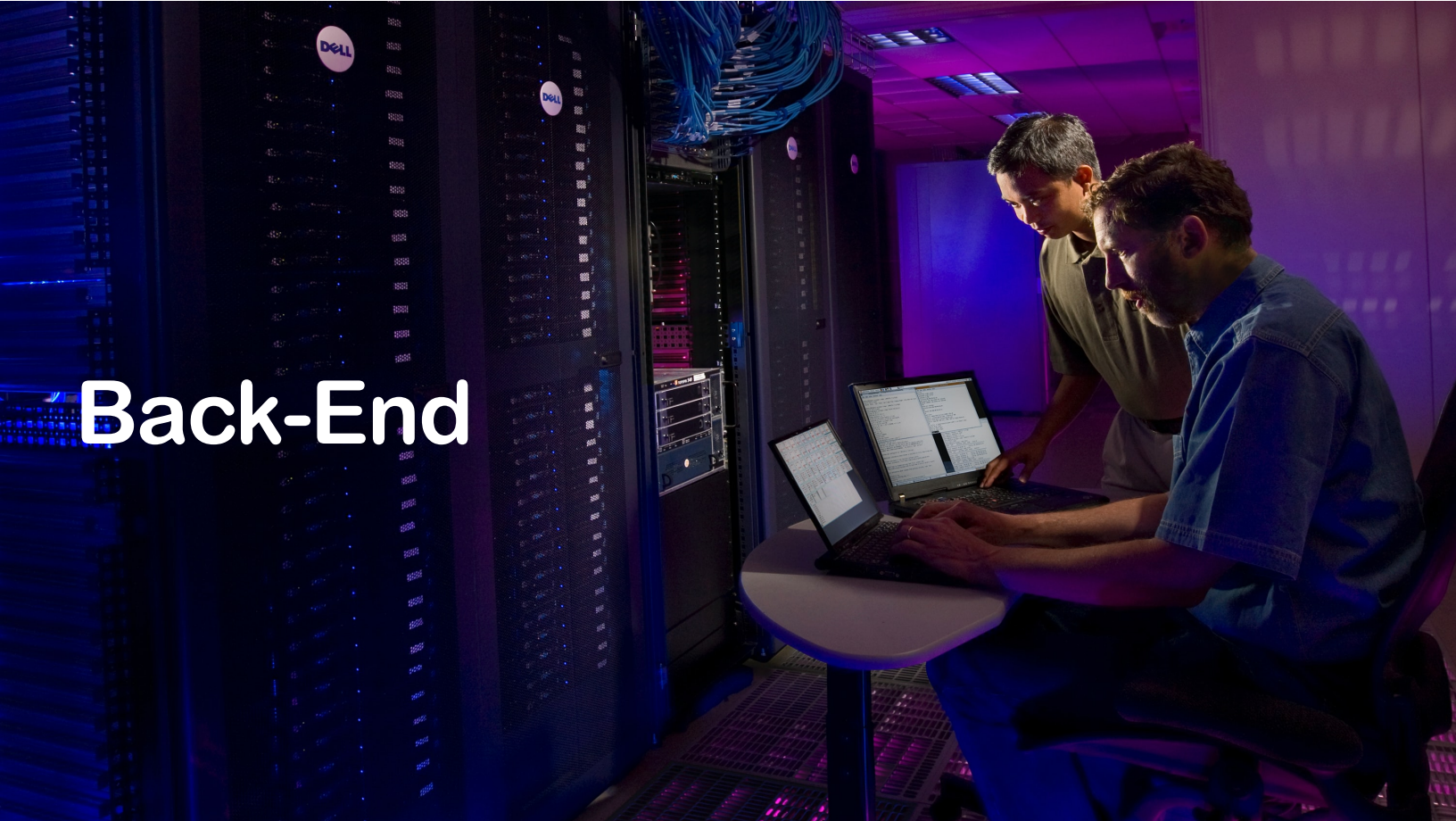
React

Marius Minia

Ciao 👋,
sono Marius

Qual è il vostro background?

Front-End


Back-End

node
JS

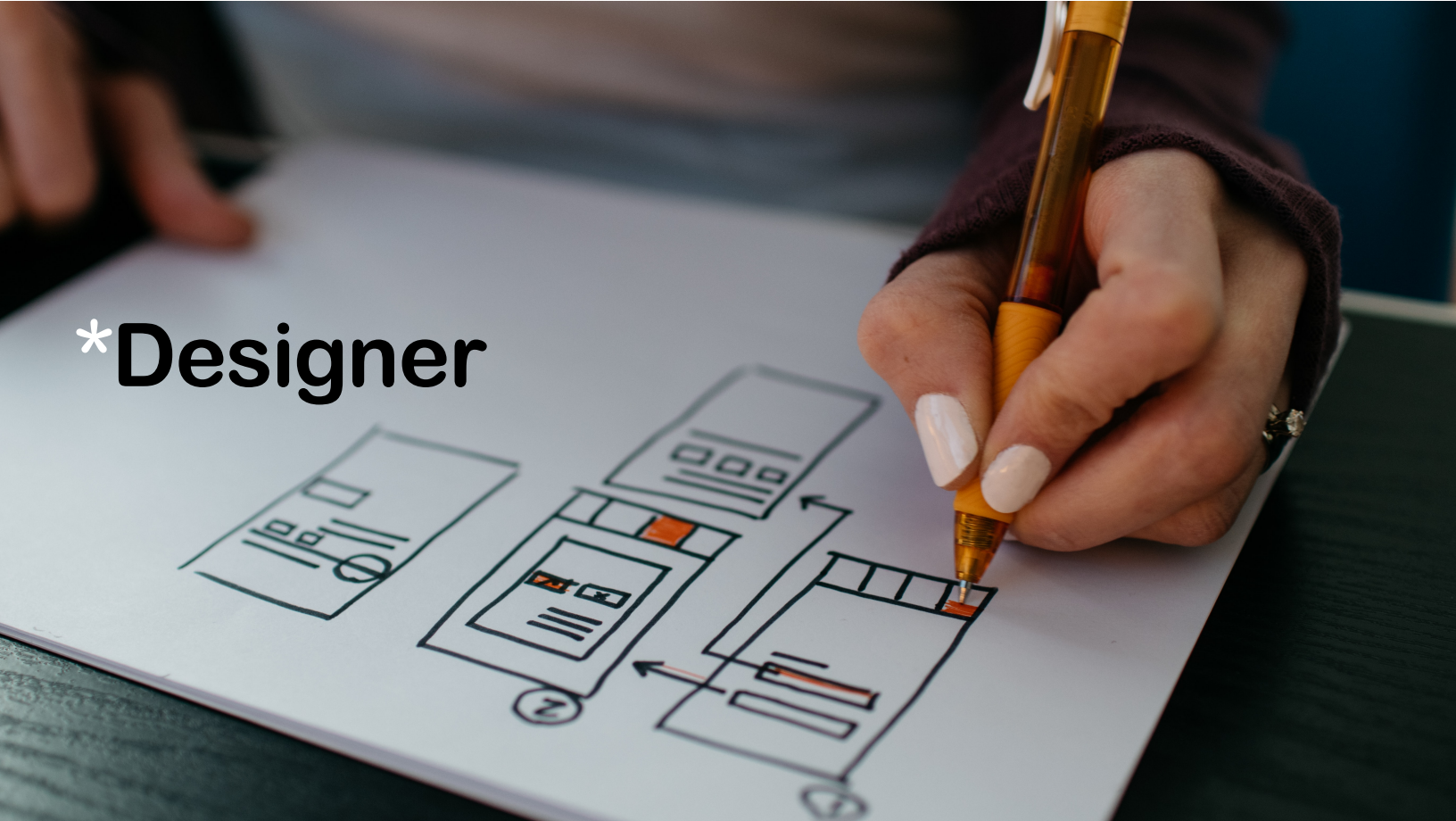
Flask
web development,
one drop at a time

spring

django

RAILS

Microsoft
.NET
Framework



*Designer

Full Stack Developer

LAS CIENCIAS
EL SIGLO XX
CATALONIA
COEL
ECCI
ONS

Bootstrap · The most popular

getbootstrap.com

Home Docs Examples Icons Themes Blog Download

Build fast, responsive sites with Bootstrap

Quickly design and customize responsive mobile-first sites with Bootstrap, the world's most popular front-end open source toolkit, featuring Sass variables and mixins, responsive grid system, extensive prebuilt components, and powerful JavaScript plugins.

Get started Download

Currently **v5.1.3** · v4.6.x docs · All releases

Installation

Install Bootstrap's source Sass and JavaScript files via npm, Composer, or Meteor.

Package managed installs don't include documentation or our full build scripts. You can also use our npm template repo to quickly generate a Bootstrap project via npm.

Read installation docs

```
$ npm install bootstrap
```

Copy

```
$ gem install bootstrap -v 5.1.3
```

Copy

jQuery

jquery.com

Plugins | Contribute | Events | Support | JS Foundation

jQuery

write less, do more.

Your donations help fund the continued development and growth of **jQuery**.

SUPPORT THE PROJECT

Download | API Documentation | Blog | Plugins | Browser Support

Search

Lightweight Footprint

Only 30kB minified and gzipped. Can also be included as an AMD module

CSS3 Compliant

Supports CSS3 selectors to find elements as well as in style property manipulation

Cross-Browser

Chrome, Edge, Firefox, IE, Safari, Android, iOS, and more

Download jQuery

v3.6.0

The 1.x and 2.x branches no longer receive patches.

View Source on GitHub →

How jQuery Works →

What is jQuery?

jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript.

Resources

- jQuery Core API Documentation
- jQuery Learning Center
- jQuery Blog
- Contribute to jQuery
- About the jQuery Foundation
- Browse or Submit jQuery Bugs

Other Related Projects

jQuery user interface | jQuery mobile | Sizzle CSS selector engine

A Brief Look

DOM Traversal and Manipulation

Get the `<button>` element with the class 'continue' and change its HTML to 'Next Step...'

```
$( "button.continue" ).html( "Next Step..." )
```

Event Handling

Show the `#banner-message` element that is hidden with `display:none` in its CSS when any button in `#button-container` is clicked.

```
var hiddenBox = $( "#banner-message" );
$( "#button-container button" ).on( "click", function( event ) {
  hiddenBox.show();
});
```

Ajax

```
$(document).ready(function(){
  $("button").click(function(){
    $("p").hide();
  });
});
```

React

React

Una libreria JavaScript per creare interfacce utente

Dichiarativo

React rende la creazione di UI interattive facile e indolore. Progetta interfacce per ogni stato della tua applicazione. Ad ogni cambio di stato React aggiornerà efficientemente solamente le parti della UI che dipendono da tali dati.

La natura dichiarativa dell'UI rende il tuo codice più prevedibile e facile da debuggare.

PROG. FUNZIONALE

Componenti

Crea componenti in isolamento e componili per creare UI complesse.

Dato che interazioni e logica per i componenti sono implementate in JavaScript puoi facilmente passare ed accedere strutture dati complesse in vari punti della tua applicazione senza dover salvare informazioni sul DOM.

Imparalo una volta, usalo ovunque

Non facciamo supposizioni sulle tecnologie che utilizzi correntemente. In questo modo puoi sviluppare nuove funzionalità in React senza riscrivere codice esistente.

React può inoltre effettuare rendering lato server con Node e in applicazioni mobile grazie a React Native.

HTML < HEAD
BODY — H1
DIV < H2
P
P

React

Un Componente Semplice

I componenti React implementano un metodo `render()` che riceve dati in input e ritorna cosa deve visualizzare. Questo esempio usa una sintassi simile ad XML chiamata JSX. I dati passati in input al componente possono essere acceduti da `render()` via `this.props`.

JSX è opzionale e non richiesto per utilizzare React. Prova il tool Babel REPL per vedere il codice JavaScript grezzo generato nel processo di compilazione JSX.

LIVE JSX EDITOR ☑ JSX?

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

root.render(<HelloMessage name="Claudia" />);
```

HTML JS

RESULT

Ciao Claudia

chrome

chrome web store

marius93rm@gmail.com

Home page > Estensioni > React Developer Tools

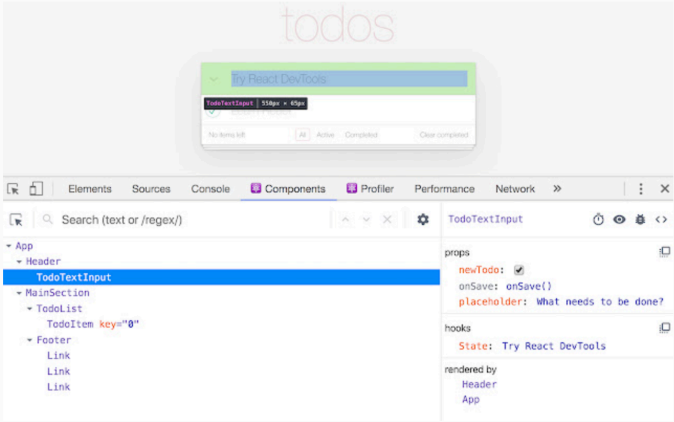
React Developer Tools

Rimuovi da Google Chrome

In primo piano

1.395 | Strumenti per sviluppatori | 3.000.000+ utenti

Panoramica | Norme di tutela della privacy | Recensioni | Assistenza | Correlati



Visual Studio Code

node
JS

JS

{_}

ES6

Giorno 2

React

Marius Minia

React

Glossario

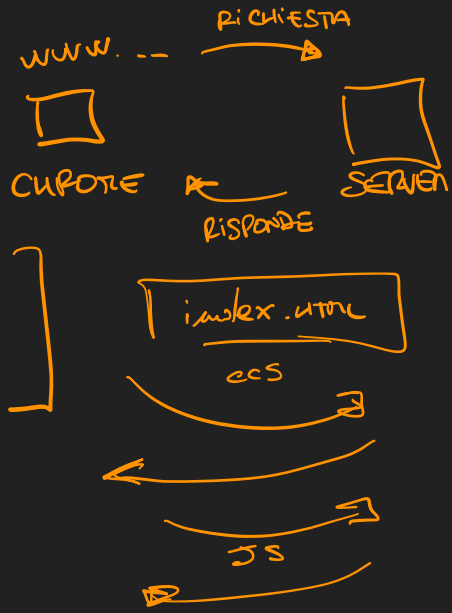
Applicazione Single-page

Un'applicazione single-page è un'applicazione che carica una singola pagina HTML e tutte le risorse necessarie (quali Javascript e CSS) per consentirne l'esecuzione. Qualunque interazione con la pagina o con le pagine successive non richiede un'interrogazione al server il che significa che la pagina non viene ricaricata.

Sebbene tu possa creare applicazioni single-page in React, questo non rappresenta un requisito. React può anche essere usato per migliorare piccole parti di un sito già esistente aggiungendo ulteriore interattività. Il codice scritto in React può coesistere tranquillamente con un markup renderizzato dal server tramite qualcosa di simile a PHP, o con altre librerie lato client. Infatti, questo è esattamente il modo in cui React viene utilizzato in Facebook.

COMPONENTE →

BOILERPLATE
(TEMPLATE)



Glossario

Compilatori

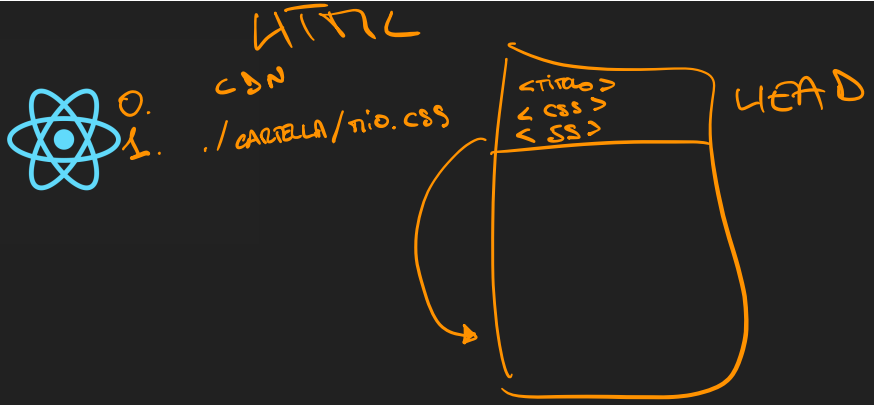
Un compilatore JavaScript prende il codice JavaScript, lo trasforma e ritorna codice JavaScript in un formato differente. Il caso d'uso più comune è quello di prendere un codice con sintassi ES6 e trasformarlo in uno con sintassi che può essere interpretata da browser più vecchi. Babel è il compilatore più comunemente usato con React.

Glossario

Gestori dei pacchetti

I gestori dei pacchetti sono strumenti che ti consentono di gestire le dipendenze nel tuo progetto. npm e Yarn sono i due gestori dei pacchetti solitamente usati nelle applicazioni React. Entrambi sono client verso lo stesso registro dei pacchetti di npm.

Glossario



CDN

CDN sta per Content Delivery Network. Le CDN consegnano contenuto statico e in cache da una rete di server sparsi nel globo.

Hello world

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<h1>Hello, world!</h1>);
```

Introduzione a JSX

Considera questa dichiarazione di variabile:

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

ELEMENTO

Questa strana sintassi con i tag non è né una stringa né HTML.

È chiamata JSX, ed è un'estensione della sintassi JavaScript. Ti raccomandiamo di utilizzarla con React per descrivere l'aspetto che dovrebbe avere la UI (*User Interface*, o interfaccia utente). JSX ti potrebbe ricordare un linguaggio di template, ma usufruisce di tutta la potenza del JavaScript.

JSX produce "elementi React". Studieremo il modo in cui gli elementi vengono renderizzati nel DOM nella prossima sezione. Qui sotto troverai le nozioni fondamentali di JSX, sufficienti per iniziare ad utilizzarlo.

Introduzione a JSX

REACT SI INTEGRA MOLTO BENE CON ALTRE TECNOLOGIE *

React non obbliga ad utilizzare JSX, ma la maggior parte delle persone lo trovano utile come aiuto visuale quando lavorano con la UI all'interno del codice JavaScript. Inoltre, JSX consente a React di mostrare messaggi di errore e di avvertimento più efficaci.

* CRA --TEMPLATE TYPESCRIPT

Introduzione a JSX

Incorporare espressioni in JSX

Nell'esempio in basso, dichiariamo una variabile chiamata `name` e poi la utilizziamo all'interno di JSX racchiudendola in parentesi graffe:

```
const name = 'Giuseppe Verdi';
const element = <h1>Hello, {name}</h1>;
```

Puoi inserire qualsiasi espressione JavaScript all'interno delle parentesi graffe in JSX. Ad esempio, `2 + 2`, `user.firstName` o `formatName(user)` sono tutte espressioni JavaScript valide.

FUNCTION

Introduzione a JSX

Nell'esempio in basso, includiamo il risultato della chiamata ad una funzione JavaScript, `formatName(user)`, in un elemento `<h1>`.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Giuseppe',
  lastName: 'Verdi'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);
```

Introduzione a JSX

JSX è un'Espressione

Dopo la compilazione, le espressioni JSX diventano normali chiamate a funzioni JavaScript che producono oggetti JavaScript.

Questo significa che puoi utilizzare JSX all'interno di istruzioni `if` e cicli `for`, assegnarlo a variabili, utilizzarlo come argomento di una funzione e restituirlo come risultato di una funzione:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

Introduzione a JSX

Specificare gli Attributi con JSX

Puoi utilizzare le virgolette per valorizzare gli attributi con una stringa:

```
const element = <a href="https://www.reactjs.org"> link </a>;
```

Puoi anche utilizzare le parentesi graffe per includere un'espressione JavaScript in un attributo:

```
const element = <img src={user.avatarUrl}></img>;
```

Non aggiungere le virgolette attorno alle parentesi graffe quando includi un'espressione JavaScript in un attributo. Dovresti utilizzare o le virgolette (per le stringhe) o le parentesi graffe (per le espressioni), ma mai entrambe nello stesso attributo.

Introduzione a JSX

Attenzione:
Dal momento che JSX è più vicino al JavaScript che all'HTML, React DOM utilizza la convenzione `camelCase` nell'assegnare il nome agli attributi, invece che quella utilizzata normalmente nell'HTML, e modifica il nome di alcuni attributi.

Ad esempio, `class` diventa `className` in JSX e `tabindex` diventa `tabIndex`.

Element.className

The **`className`** property of the `Element` interface gets and sets the value of the `class` attribute of the specified element.

Value

A string variable representing the class or space-separated classes of the current element.

Examples

```
const el = document.getElementById('item');
el.className = el.className === 'active' ? 'inactive' :
'active';
```

Notes

The name `className` is used for this property instead of `class` because of conflicts with the "class" keyword in many languages which are used to manipulate the DOM.

Introduzione a JSX

Specificare Figli in JSX

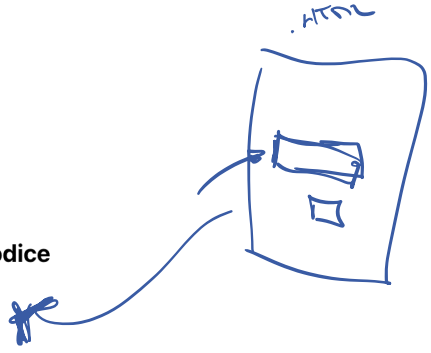
Se un tag è vuoto, puoi chiuderlo immediatamente con `/>`, come in XML:

```
const element = <img src={user.avatarUrl} />;
```

I tag JSX possono contenere figli:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

Introduzione a JSX



JSX Previene gli Attacchi di Iniezione del Codice

Utilizzare l'input degli utenti in JSX è sicuro:

```
const title = response.contenutoPotenzialmentePericoloso;
// Questo è sicuro:
const element = <h1>{title}</h1>;
```

React DOM effettua automaticamente l'escape di qualsiasi valore inserito in JSX prima di renderizzarlo. In questo modo, garantisce che non sia possibile iniettare nulla che non sia esplicitamente scritto nella tua applicazione. Ogni cosa è convertita in stringa prima di essere renderizzata. Questo aiuta a prevenire gli attacchi XSS (cross-site-scripting).

Introduzione a JSX

JSX Rappresenta Oggetti

Babel compila JSX in chiamate a `React.createElement()`.

Questi due esempi sono identici:

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

TAG
PROPRIETÀ
CONTENUTO

Introduzione a JSX

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

`React.createElement()` effettua alcuni controlli per aiutarti a scrivere codice senza bug, ma fondamentalmente crea un oggetto come questo:

input

```
// Nota: questa struttura è semplificata
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

Renderizzare Elementi

Gli elementi sono i più piccoli tra i vari mattoni costituenti apps scritte in React.

Un elemento descrive cosa vuoi vedere sullo schermo:

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

Contrariamente agli elementi DOM del browser, gli elementi React sono oggetti semplici e per questo più veloci da creare. Il DOM di React tiene cura di aggiornare il DOM del browser per essere consistente con gli elementi React.

Renderizzare Elementi

Renderizzare un Elemento nel DOM

Supponiamo di avere un `<div>` da qualche parte nel tuo file HTML:

```
<div id="root"></div>
```

Lo chiameremo nodo DOM “radice” (o root) in quanto ogni cosa al suo interno verrà gestita dal DOM di React.

Applicazioni costruite solo con React di solito hanno un solo nodo DOM radice. Se stai integrando React all’interno di apps esistenti, potresti avere più elementi DOM radice isolati, dipende dai casi.

Renderizzare Elementi

Per renderizzare un elemento React, passa l'elemento DOM a `ReactDOM.createRoot()`, successivamente passa l'elemento React a `root.render()`:

```
const root = ReactDOM.createRoot(
  document.getElementById('root')
);
const element = <h1>Hello, world</h1>;
root.render(element);
```

Renderizzare Elementi

Aggiornare un Elemento Renderizzato

Gli elementi React sono immutabili. Una volta creato un elemento, non puoi cambiare i suoi figli o attributi. Un elemento è come un singolo fotogramma di un film: rappresenta la UI (interfaccia utente) ad un certo punto nel tempo.

Con la conoscenza che abbiamo fino a questo punto, l'unico modo per aggiornare l'UI è quello di creare un nuovo elemento e di passarlo a `root.render()`.

Renderizzare Elementi

Prendiamo in considerazione il prossimo esempio, nel quale abbiamo un orologio:

```
const root = ReactDOM.createRoot(
  document.getElementById('root')
);

function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

Renderizzare Elementi

Chiama `root.render()` ogni secondo da una callback `setInterval()`.

> **Nota bene:**
> In pratica, la maggioranza delle applicazioni React chiamano `root.render()` solo una volta. Nelle sezioni successive impareremo che questo codice viene incapsulato in componenti stateful.

PROPS STATO

INPUT

IL COMPONENTE LA

Renderizzare Elementi

React Aggiorna Solo Quanto Necessario

Il DOM di React confronta l'elemento ed i suoi figli con il precedente, applicando solo gli aggiornamenti al DOM del browser necessari a renderlo consistente con lo stato desiderato.

Puoi verificare questo fatto ispezionando l'ultimo esempio usando i developer tools:

Hello, world!

It is 12:26:46 PM.

Console Sources Network Timeline

```
▼<div id="root">
  ▼<div data-reactroot>
     <h1>Hello, world!</h1>
   ▼<h2>
      <!-- react-text: 4 -->
      "It is "
      <!-- /react-text -->
      <!-- react-text: 5 -->
      "12:26:46 PM"
      <!-- /react-text -->
      <!-- react-text: 6 -->
      "."
      <!-- /react-text -->
    </h2>
  </div>
</div>
```

Anche se abbiamo creato un elemento che descrive l'intero albero della UI ad ogni tick (ogni qual volta la callback viene richiamata, nell'esempio, ogni secondo), solo il nodo testo il quale contenuto è stato modificato viene aggiornato dal DOM di React.

Nella nostra esperienza, pensare a come la UI deve essere rappresentata in ogni momento piuttosto che pensare a come alterarla nel tempo, elimina una intera classe di bugs.

Componenti e Props

I Componenti ti permettono di suddividere la UI (*User Interface*, o interfaccia utente) in parti indipendenti, riutilizzabili e di pensare ad ognuna di esse in modo isolato. Questo capitolo offre una introduzione al concetto dei componenti. Puoi trovare invece informazioni dettagliate nella API di riferimento dei componenti.

Concettualmente, i componenti sono come funzioni JavaScript: accettano in input dati arbitrari (sotto il nome di "props") e ritornano elementi React che descrivono cosa dovrebbe apparire sullo schermo.

Componenti e Props

Funzioni e Classi Componente

Il modo più semplice di definire un componente è quello di scrivere una funzione JavaScript:

```
function Ciao(props) {
  return <h1>Ciao, {props.nome}</h1>;
}
```

JSX

Questa funzione è un componente React valido in quanto accetta un oggetto parametro contenente dati sotto forma di una singola “props” (che prende il nome da “properties” in inglese, ossia “proprietà”) che è un oggetto parametro avente dati al suo interno e ritorna un elemento React. Chiameremo questo tipo di componenti “componenti funzione” perché sono letteralmente funzioni JavaScript.

Componenti e Props

Puoi anche usare una classe ES6 per definire un componente:

```
class Ciao extends React.Component {
  render() {
    return <h1>Ciao, {this.props.nome}</h1>;
  }
}
```

I due componenti appena visti sono equivalenti dal punto di vista di React.

Le Classi e i Componenti Funzione hanno funzionalità aggiuntive che verranno discusse in dettaglio nelle prossime sezioni.

Componenti e Props

Renderizzare un Componente

In precedenza, abbiamo incontrato elementi React che rappresentano tags DOM:

```
const elemento = <div />;
```

Comunque, gli elementi possono rappresentare anche componenti definiti dall'utente:

PROPS

```
const elemento = <Ciao nome="Sara" />;
```

Quando React incontra un elemento che rappresenta un componente definito dall'utente, passa gli attributi JSX ed i figli a questo componente come un singolo oggetto. Tale oggetto prende il nome di "props".

Ad esempio, il codice seguente renderizza il messaggio "Ciao, Sara" nella pagina:

```
function Ciao(props) {
  return <h1>Ciao, {props.nome}</h1>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
const elemento = <Ciao nome="Sara" />;
root.render(element);
```

COMPONENTE = ' '

PROPS

Componenti e Props

```
function Ciao(props) {
  return <h1>Ciao, {props.nome}</h1>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
const elemento = <Ciao nome="Sara" />;
root.render(element);
```

Ricapitoliamo cosa succede nell'esempio:

Props

1. Richiamiamo `root.render()` con l'elemento `<Ciao nome="Sara" />`.
2. React chiama a sua volta il componente ~~`Welcome`~~ Ciao con `{nome: 'Sara'}` passato in input come props.
3. Il nostro componente `Ciao` ritorna un elemento `<h1>Ciao, Sara</h1>` come risultato.
4. React DOM aggiorna efficientemente il DOM per far sì che contenga `<h1>Ciao, Sara</h1>`.

> **Nota Bene: Ricordati di chiamare i tuoi componenti con la prima lettera in maiuscolo.**
> React tratta i componenti che iniziano con una lettera minuscola come normali tags DOM. per esempio, `<div />` rappresenta un tag HTML div, `<Ciao />` rappresenta invece un componente e richiede `Ciao` all'interno dello scope.

Componenti e Props

Comporre Componenti

I componenti possono far riferimento ad altri componenti nel loro output. Ciò permette di utilizzare la stessa astrazione ad ogni livello di dettaglio. Un bottone, un form, una finestra di dialogo, una schermata: nelle applicazioni React, tutte queste cose di solito sono espresse come componenti.

Per esempio, possiamo creare un componente `App` che renderizza `Ciao` tante volte:

```
function Ciao(props) {
  return <h1>Ciao, {props.nome}</h1>;
}

function App() {
  return (
    <div>
      <Ciao nome="Sara" />
      <Ciao nome="Cahal" />
      <Ciao nome="Edite" />
    </div>
  );
}
```

Componenti e Props

Normalmente, le nuove applicazioni React hanno un singolo componente chiamato `App` al livello più alto che racchiude tutti gli altri componenti. Ad ogni modo, quando si va ad integrare React in una applicazione già esistente, è bene partire dal livello più basso e da piccoli componenti come ad esempio `Bottone` procedendo da lì fino alla cima della gerarchia della vista.

Componenti e Props

OGG. COMMENTO

Ad esempio, considera questo componente `Commento`:

FUNZIONE

```
function Commento(props) {
  return (
    <div className="Commento">
      <div className="InfoUtente">
        <img className="Avatar"
          src={props.autore.avatarUrl}
          alt={props.autore.nome}
        />
        <div className="InfoUtente-nome">
          {props.autore.nome}
        </div>
      </div>
      <div className="Commento-testo">
        {props.testo}
      </div>
      <div className="Commento-data">
        {formatDate(props.data)}
      </div>
    </div>
  );
}
```

Esso accetta come props: `autore` (un oggetto), `testo` (una stringa) e `data` (sotto forma di oggetto Date) al fine di renderizzare un commento in un sito di social media, come Facebook.

Un componente scritto in quel modo, con codice molto annidato, è difficile da modificare. Per lo stesso motivo, non si possono riutilizzare con facilità parti dello stesso. Procediamo quindi ad estrarre qualche componente.

Componenti e Props

Per cominciare, estraiamo `Avatar`:

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.utente.avatarUrl}
      alt={props.utente.nome}
    />
  );
}
```

`Avatar` non ha bisogno di sapere che viene renderizzato all'interno di un `Commento`. Ecco perché abbiamo dato alla sua prop un nome più generico: `utente` al posto di `autore`.

Consigliamo di dare il nome alle props dal punto di vista del componente piuttosto che dal contesto in cui viene usato.

Componenti e Props

Adesso possiamo semplificare un po' il componente `Commento`:

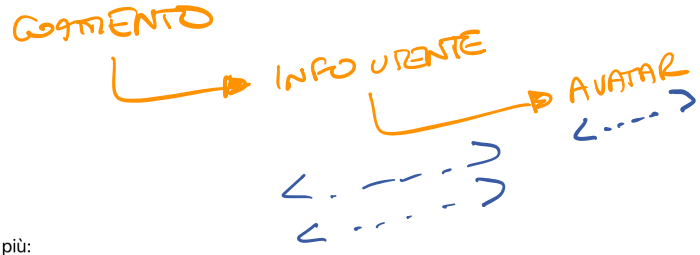
```
function Commento(props) {
  return (
    <div className="Commento">
      <div className="InfoUtente">
        <Avatar utente={props.autore} />
        <div className="InfoUtente-nome">
          {props.autore.nome}
        </div>
      </div>
      <div className="Commento-testo">
        {props.testo}
      </div>
      <div className="Commento-data">
        {formatDate(props.data)}
      </div>
    </div>
  );
}
```

Componenti e Props

Andiamo ora ad estrarre il componente `InfoUtente` che renderizza un `Avatar` vicino al nome dell'utente:

```
function InfoUtente(props) {
  return (
    <div className="InfoUtente">
      <Avatar utente={props.utente} />
      <div className="InfoUtente-nome">
        {props.utente.nome}
      </div>
    </div>
  );
}
```

Componenti e Props



Ciò ci permette di semplificare `Commento` ancora di più:

```
function Commento(props) {
  return (
    <div className="Commento">
      <InfoUtente utente={props.autore} />
      <div className="Commento-testo">
        {props.testo}
      </div>
      <div className="Commento-data">
        {formatDate(props.data)}
      </div>
    </div>
  );
}
```

Componenti e Props

Estrarre componenti può sembrare un'attività pesante ma avere una tavolozza di componenti riutilizzabili ripaga molto bene nelle applicazioni più complesse. Una buona regola da tenere a mente è che se una parte della tua UI viene usata diverse volte (`Bottone`, `Pannello`, `Avatar`) o se è abbastanza complessa di per sé (`App`, `StoriaFeed`, `Commento`), allora questi componenti sono buoni candidati per essere estratti come componenti separati.

Componenti e Props

Le Props Sono in Sola Lettura

Ogni volta che dichiari un componente come funzione o classe, non deve mai modificare le proprie props. Considera la funzione `somma`:

```
function somma(a, b) {
  return a + b;
}
```

Funzioni di questo tipo vengono chiamate "pure" perché non provano a cambiare i propri dati in input, ritornano sempre lo stesso risultato a partire dagli stessi dati in ingresso.

Al contrario, la funzione seguente è impura in quanto altera gli input:

```
function preleva(conto, ammontare) {
  conto.totale -= ammontare;
}
```

Componenti e Props

React è abbastanza flessibile ma ha una sola regola molto importante:

Tutti i componenti React devono comportarsi come funzioni pure rispetto alle proprie props.

Ovviamente, le UI delle applicazioni sono dinamiche e cambiano nel tempo. Nella prossima sezione, introdurremo il nuovo concetto di "stato". Lo stato permette ai componenti React di modificare il loro output nel tempo in seguito ad azioni dell'utente, risposte dalla rete (API) e qualsiasi altra cosa possa far renderizzare un output diverso di volta in volta, ciò avviene senza violare questa regola molto importante.

State e Lifecycle

Considera l'esempio dell'orologio di una delle sezioni precedenti. In Renderizzare Elementi, abbiamo appreso solamente un modo per aggiornare la UI. Chiamiamo `root.render()` per cambiare l'output renderizzato:

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function tick() {
  const element = (
    <div>
      <h1>Ciao, mondo!</h1>
      <h2>Sono le {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

State e Lifecycle

In questa sezione, apprenderemo come rendere il componente `Clock` davvero riutilizzabile ed incapsulato. Esso si occuperà di impostare il proprio timer e di aggiornarsi ogni secondo.

Possiamo iniziare incapsulando l'aspetto dell'orologio:

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function Clock(props) {
  return (
    <div>
      <h1>Ciao, mondo!</h1>
      <h2>Sono le {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  root.render(<Clock date={new Date()} />);
}

setInterval(tick, 1000);
```

State e Lifecycle

Tuttavia, manca un requisito fondamentale: il fatto che `Clock` imposti un timer ed aggiorni la propria UI ogni secondo dovrebbe essere un dettaglio implementativo di `Clock`.

Idealmente, vorremmo scrivere il seguente codice una volta sola, ed ottenere che `Clock` si aggiorni da solo:

```
root.render(<Clock />);
```

Per implementare ciò, abbiamo bisogno di aggiungere uno “stato” al componente `Clock`.

Lo state (o stato) è simile alle props, ma è privato e completamente controllato dal componente.

State e Lifecycle

Convertire una Funzione in una Classe

Puoi convertire un componente funzione come `Clock` in una classe in cinque passaggi:

1. Crea una classe ES6, con lo stesso nome, che estende `React.Component`.
2. Aggiungi un singolo metodo vuoto chiamato `render()`.
3. Sposta il corpo della funzione all'interno del metodo `render()`.
4. Sostituisci `props` con `this.props` nel corpo del metodo `render()`.
5. Rimuovi la dichiarazione della funzione rimasta vuota.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Ciao, mondo!</h1>
        <h2>Sono le {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

State e Lifecycle

`Clock` è ora definito da una classe, invece che da una funzione.

Il metodo `render` viene invocato ogni volta che si verifica un aggiornamento, ma finché renderizziamo `<Clock />` nello stesso nodo del DOM, verrà utilizzata un'unica istanza della classe `Clock`. Questo ci consente di utilizzare funzionalità aggiuntive come il local state e i metodi del lifecycle del componente.

State e Lifecycle

Aggiungere il Local State ad una Classe

Sposteremo `date` dalle props allo state in tre passaggi:

1. Sostituisci `this.props.date` con `this.state.date` nel metodo `render()`:

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Ciao, mondo!</h1>
        <h2>Sono le {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

State e Lifecycle

2. Aggiungi un costruttore di classe che assegna il valore iniziale di `this.state`:

FIGLIA PADRE

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Ciao, mondo!</h1>
        <h2>Sono le {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

State e Lifecycle

Nota come passiamo `props` al costruttore di base:

```
constructor(props) {
  super(props);
  this.state = {date: new Date()};
}
```

I componenti classe dovrebbero sempre chiamare il costruttore base passando `props` come argomento.

State e Lifecycle

3. Rimuovi la prop `date` dall'elemento `<Clock />`:

```
root.render(<Clock />);
```

In seguito ci occuperemo di aggiungere la parte di codice relativa al timer all'interno del componente stesso.

State e Lifecycle

Il risultato dovrebbe avere questo aspetto:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Ciao, mondo!</h1>
        <h2>Sono le {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

State e Lifecycle

Aggiungere Metodi di Lifecycle ad una Classe

Nelle applicazioni con molti componenti, è molto importante rilasciare le risorse occupate dai componenti quando questi vengono distrutti.

Nel nostro caso, vogliamo impostare un timer ogni volta che `Clock` è renderizzato nel DOM per la prima volta. Questo è definito “mounting” (“montaggio”) in React.

Vogliamo anche cancellare il timer ogni volta che il DOM prodotto da `Clock` viene rimosso. Questo è definito “unmounting” (“smontaggio”) in React.

State e Lifecycle

Possiamo dichiarare alcuni metodi speciali nel componente classe per eseguire del codice quando il componente viene montato e smontato:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
  }

  componentWillUnmount() {
  }

  render() {
    return (
      <div>
        <h1>Ciao, mondo!</h1>
        <h2>Sono le {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Questi metodi sono chiamati “metodi del lifecycle” (metodi del ciclo di vita).

State e Lifecycle

Il metodo `componentDidMount()` viene eseguito dopo che l'output del componente è stato renderizzato nel DOM. È un buon punto in cui impostare un timer:

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```

Nota come salviamo l'ID del timer direttamente in `this` (`this.timerID`).

State e Lifecycle

Ci occuperemo di cancellare il timer nel metodo del lifecycle `componentWillUnmount()`:

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

Infine, implementeremo un metodo chiamato `tick()` che verrà invocato dal componente `Clock` ogni secondo.

Il nuovo metodo utilizzerà `this.setState()` per pianificare gli aggiornamenti al local state del componente:

```
tick() {
  this.setState({
    date: new Date()
  });
}
```

State e Lifecycle

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Ciao, mondo!</h1>
        <h2>Sono le {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

Ricapitoliamo velocemente quello che sta succedendo e l'ordine con cui i metodi sono invocati:

1. Quando `<Clock />` viene passato a `root.render()`, React invoca il costruttore del componente `Clock`. Dal momento che `Clock` ha bisogno di mostrare l'ora corrente, inizializza `this.state` con un oggetto che include l'ora corrente. In seguito, aggiorneremo questo state.
2. In seguito, React invoca il metodo `render()` del componente `Clock`. Questo è il modo in cui React apprende cosa dovrebbe essere visualizzato sullo schermo. React si occupa di aggiornare il DOM in modo da farlo corrispondere all'output della renderizzazione di `Clock`.
3. Quando l'output della renderizzazione di `Clock` viene inserito nel DOM, React invoca il metodo del lifecycle `componentDidMount()`. Al suo interno, il componente `Clock` chiede al browser di impostare un timer con cui invocare il metodo `tick()` del componente una volta al secondo.
4. Ogni secondo, il browser invoca il metodo `tick()`. Al suo interno, il componente `Clock` pianifica un aggiornamento della UI invocando `setState()` con un oggetto che contiene la nuova ora corrente. Grazie alla chiamata a `setState()`, React viene informato del fatto che lo state è cambiato e invoca di nuovo il metodo `render()` per sapere che cosa deve essere mostrato sullo schermo. Questa volta, `this.state.date` nel metodo `render()` avrà un valore differente, di conseguenza l'output della renderizzazione includerà l'orario aggiornato. React aggiorna il DOM di conseguenza.
5. Se il componente `Clock` dovesse mai essere rimosso dal DOM, React invocherebbe il metodo del lifecycle `componentWillUnmount()` ed il timer verrebbe cancellato.

State e Lifecycle

Utilizzare Correttamente lo Stato

Ci sono tre cose che devi sapere a proposito di `setState()`.

Non Modificare lo Stato Direttamente

Per esempio, questo codice non farebbe ri-renderizzare un componente:

```
// Sbagliato
this.state.comment = 'Hello';
```

Devi invece utilizzare `setState()`:

```
// Giusto
this.setState({comment: 'Hello'});
```

L'unico punto in cui puoi assegnare direttamente un valore a `this.state` è nel costruttore.

State e Lifecycle

Gli Aggiornamenti di Stato Potrebbero Essere Asincroni

React potrebbe accorpare più chiamate a `setState()` in un unico aggiornamento per migliorare la performance.

Poiché `this.props` e `this.state` potrebbero essere aggiornate in modo asincrono, non dovresti basarti sul loro valore per calcolare lo stato successivo.

State e Lifecycle

Ad esempio, questo codice potrebbe non riuscire ad aggiornare correttamente il contatore:

```
// Sbagliato
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

Per effettuare correttamente questa operazione, bisogna utilizzare una seconda forma di `setState()` che accetta in input una funzione invece che un oggetto. Quella funzione riceverà come primo argomento lo stato precedente e come secondo argomento le proprietà, aggiornate al momento in cui l'aggiornamento di stato è applicato:

CALLBACK

```
// Giusto
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

State e Lifecycle

Gli Aggiornamenti di Stato Vengono Applicati Tramite Merge

Quando chiami `setState()`, React effettua il merge dell'oggetto che fornisci nello state corrente.

Ad esempio, il tuo state potrebbe contenere molte variabili indipendenti:

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

State e Lifecycle

A questo punto puoi aggiornarle indipendentemente con invocazioni separate del metodo `setState()`:

STATE

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

{ POSTS: " "

{ COMMENTS: " "

Quello che viene effettuato è uno "shallow merge", quindi `this.setState({comments})` lascia intatto `this.state.posts`, ma sostituisce completamente `this.state.comments`.

State e Lifecycle

Questo è spesso definito flusso di dati "top-down" (dall'alto verso il basso) o "unidirezionale". In questo paradigma, lo stato è sempre posseduto da uno specifico componente, e tutti i dati o la UI derivati da quello stato possono influenzare solamente i componenti "più in basso" nell'albero.

Se immagini un albero di componenti come una cascata di props, puoi pensare allo stato di ciascun componente come a una sorgente d'acqua aggiuntiva che si unisce alla cascata in un punto qualsiasi e flusice verso il basso insieme al resto dell'acqua.

Per mostrare che tutti i componenti sono davvero isolati, possiamo creare un componente `App` che renderizza tre `<Clock>`:

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}
```

Ciascun `Clock` imposta il proprio timer e si aggiorna indipendentemente dagli altri.

Nelle applicazioni React, il fatto che un componente sia stateful o stateless è considerato un dettaglio implementativo di quel componente, che potrebbe cambiare nel tempo. Puoi utilizzare componenti stateless all'interno di componenti stateful, e viceversa.

Stili e CSS

Utilizzare inline styling è una cattiva idea?

In linea di massima, l'utilizzo di classi comporta una performance migliore rispetto all'inline styling.

Che cos'è CSS-in-JS?

Con "CSS-in-JS" si intende la pratica di definire styling CSS direttamente all'interno di un file JavaScript, anziché utilizzare dei file esterni.

Tieni presente che questa funzionalità non è parte di React, ma è fornita da librerie esterne. React non fornisce particolari indicazioni su come e dove definire lo styling. Nel dubbio, un buon punto di partenza é quello di definire il tuo styling in un file `*.css` separato, ed utilizzare cio che é definito al suo interno tramite `className`.

Posso utilizzare animazioni in React?

React può essere utilizzato per qualsiasi tipo di animazione. Prova a dare un'occhiata a React Transition Group, React Motion, React Spring o Framer Motion, per esempio.

Gestione degli Eventi

La gestione degli eventi negli elementi React è molto simile alla gestione degli eventi negli elementi DOM. Vi sono alcune differenze sintattiche:

- Gli eventi React vengono dichiarati utilizzando camelCase, anziché in minuscolo.
- In JSX, il gestore di eventi (*event handler*) viene passato come funzione, piuttosto che stringa.

Gestione degli Eventi

Per esempio, l'HTML:

```
<button onclick="attivaLasers()">
  Attiva Lasers
</button>
```

è leggermente diverso in React:

```
<button onClick={attivaLasers}>
  Attiva Lasers
</button>
```

Gestione degli Eventi

Un'altra differenza è che, in React, non è possibile ritornare `false` per impedire il comportamento predefinito. Devi chiamare `preventDefault` esplicitamente. Ad esempio, in un semplice codice HTML per impedire il comportamento predefinito del form nel submit, potresti scrivere:

```
<form onsubmit="console.log('Hai cliccato Invia.'); return false">
  <button type="submit">Invia</button>
</form>
```

In React, invece sarebbe:

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('Hai cliccato Invia.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Invia</button>
    </form>
  );
}
```

In questo esempio, il parametro `e` è un evento sintetico (*synthetic event*). React definisce questi eventi sintetici in base alle specifiche W3C, quindi non hai bisogno di preoccuparti della compatibilità tra browser. Gli eventi React non funzionano esattamente allo stesso modo degli eventi nativi. Consulta la guida di riferimento `SyntheticEvent` per saperne di più.

NO

Usando React, in generale, non dovresti aver bisogno di chiamare `addEventListener` per aggiungere listeners ad un elemento DOM dopo la sua creazione. Invece, basta fornire un listener quando l'elemento è inizialmente renderizzato.

Gestione degli Eventi

Quando definisci un componente usando una classe ES6, un pattern comune è usare un metodo della classe come gestore di eventi. Ad esempio, questo componente `Interruttore` renderizza un pulsante che consente all'utente di alternare gli stati "Acceso" e "Spento":

```
class Interruttore extends React.Component {
  constructor(props) {
    super(props);
    this.state = {acceso: true};

    // Necessario per accedere al corretto valore di `this`
all'interno della callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      acceso: !prevState.acceso
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.acceso ? 'Acceso' : 'Spento'}
      </button>
    );
  }
}
```

Gestione degli Eventi

Fai attenzione al valore di `this` nelle callback JSX. In JavaScript, i metodi delle classi non sono associati (*bound*) di default. Se dimentichi di applicare `bind` a `this.handleClick` e di passarlo a `onClick`, `this` sarà `undefined` quando la funzione verrà effettivamente chiamata.

Questo non è un comportamento specifico in React: è parte di come funzionano le funzioni in JavaScript. In generale, se ti riferisci ad un metodo senza `()` dopo di esso, per esempio `onClick = {this.handleClick}`, potresti aver bisogno di applicare `bind` a quel metodo.

Gestione degli Eventi

Se usare la chiamata al metodo `bind` ti sembra troppo, ci sono due alternative a disposizione. Puoi usare la sintassi proprietà pubbliche delle classi, per associare correttamente le callback:

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  handleClick = () => {
    console.log('this is:', this);
  };
  render() {
    return (
      <button onClick={this.handleClick}>
        Clicca qui
      </button>
    );
  }
}
```

Questa sintassi è abilitata nelle impostazioni predefinite di Create React App.

Gestione degli Eventi

Se non stai usando la sintassi delle proprietà delle classi, è possibile utilizzare una funzione a freccia all'interno della callback:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('Il valore di `this` è: ', this);
  }

  render() {
    // Questa sintassi garantisce che `this` sia associato
correttamente all'interno di handleClick
    return (
      <button onClick={() => this.handleClick()}>
        Clicca qui
      </button>
    );
  }
}
```

Il problema con questa sintassi è che viene creata una callback diversa ogni volta che `LoggingButton` viene renderizzato. Nella maggior parte dei casi, non vi sono problemi. Tuttavia, se questa callback viene passata come prop a componenti inferiori, tali componenti potrebbero eseguire un ulteriore re-renderizzamento. In generale, vi consigliamo di utilizzare `bind` nel costruttore o la sintassi delle proprietà pubbliche nelle classi, per evitare questo tipo di problema di prestazioni.

Gestione degli Eventi

Passare Argomenti ai Gestori di Eventi

All'interno di un ciclo, è comune avere l'esigenza di passare un parametro aggiuntivo ad un gestore di eventi. Ad esempio, avendo `id` come l'identificativo della riga, le seguenti dichiarazioni sarebbero entrambe valide:

```
<button onClick={(e) => this.deleteRow(id, e)}>Elimina riga</button>
<button onClick={this.deleteRow.bind(this, id)}>Elimina riga</button>
```

Le due linee di codice precedenti sono equivalenti e utilizzano le funzioni a freccia e `Function.prototype.bind` rispettivamente.

In entrambi i casi, l'argomento `e`, che rappresenta l'evento React, verrà passato come secondo argomento dopo l'ID. Con la funzione a freccia, devi passarlo esplicitamente, mentre con `bind` qualsiasi altro argomento viene passato automaticamente.

SyntheticEvent

I tuoi event handlers riceveranno istanze di `SyntheticEvent`, un contenitore cross-browser intorno all'evento nativo del browser. Hanno entrambi la stessa interfaccia, compresi `stopPropagation()` e `preventDefault()`, l'eccezione sta nel fatto che gli eventi funzionano in modo identico in tutti i browser.

Se constati di avere bisogno dell'evento del browser sottostante per qualche motivo, puoi ottenerlo semplicemente usando l'attributo `nativeEvent`. Gli eventi sintetici hanno una forma differente rispetto agli eventi nativi del browser. Per esempio: in `onMouseLeave` `event.nativeEvent` punta all'evento `mouseout`. La mappatura specifica non fa parte delle API pubbliche e per questo é soggetta a cambiamenti inaspettati. Ogni `SyntheticEvent` oggetto ha i seguenti attributi:

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
void persist()
DOMEventTarget target
number timeStamp
string type
```