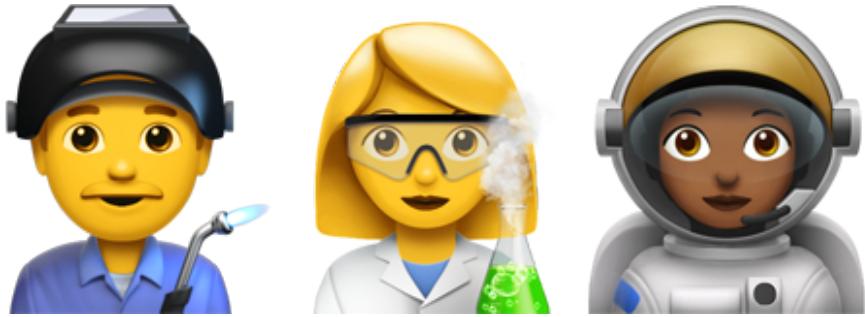


# React

Marius Minia

**Ciao**   
**sono Marius**

# Qual è il vostro background?



# Front-End

HTML



JS



CSS



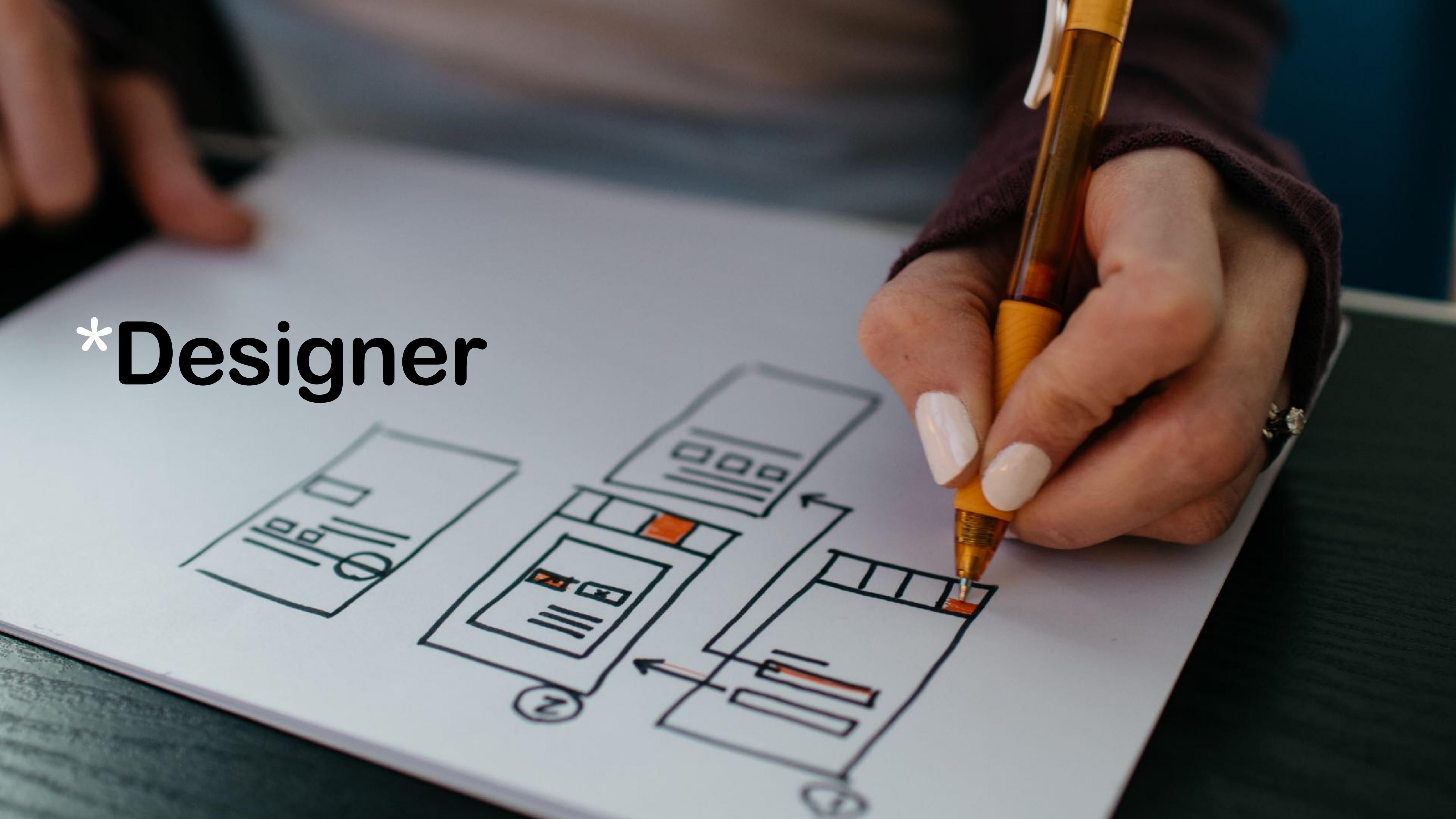
Bootstrap

# Back-End





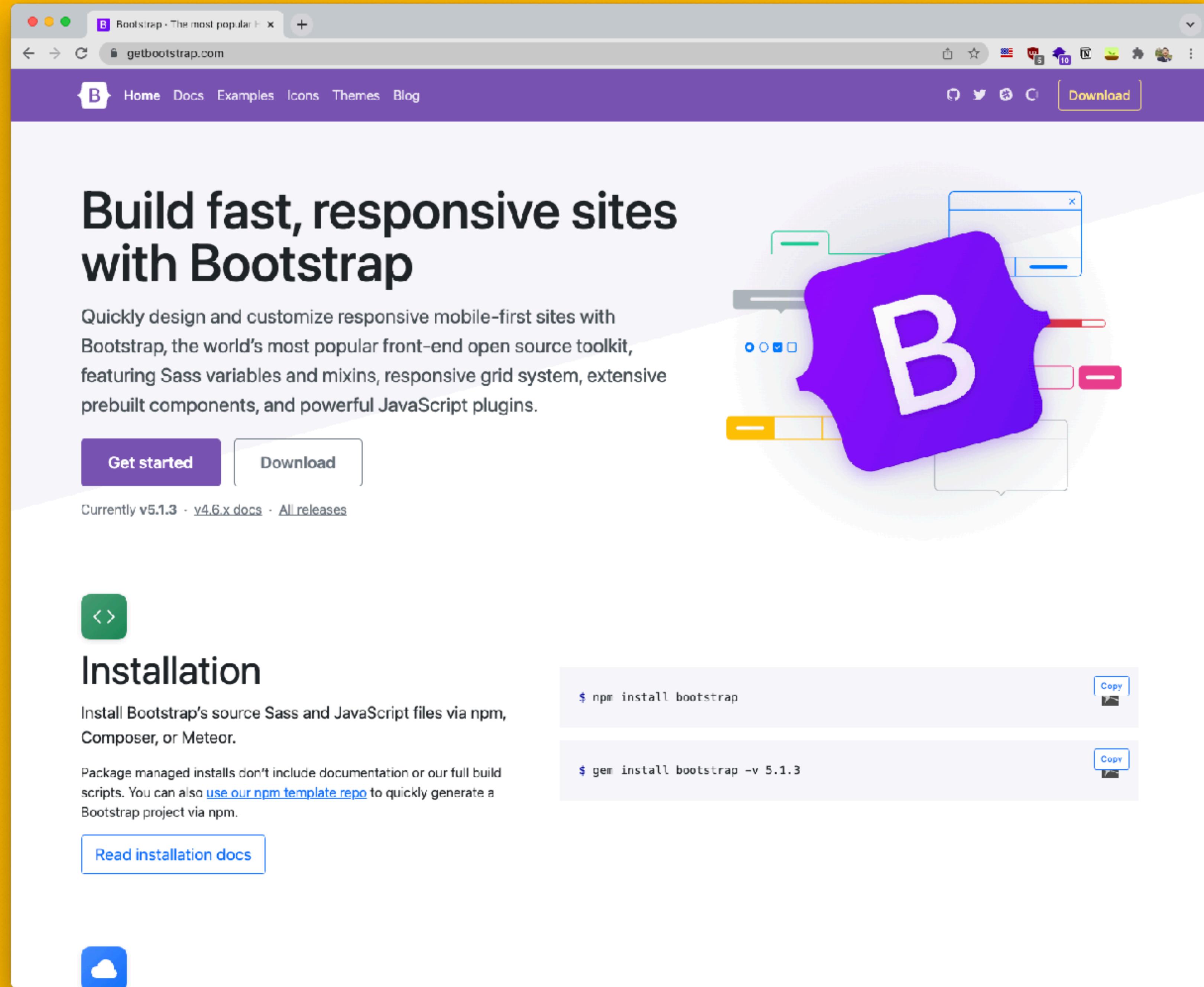
\*Designer



A close-up of Thanos' face and upper body. He has a serious, almost smug expression. His right hand is raised, showing his gauntlet which is glowing with energy. The background consists of vibrant, radiating streaks of light in shades of green, blue, and yellow, creating a sense of power and energy.

Full Stack Developer



A screenshot of a web browser displaying the official Bootstrap website. The page features a purple header with the Bootstrap logo and navigation links for Home, Docs, Examples, Icons, Themes, and Blog. A prominent 'Download' button is located in the top right corner. The main content area has a white background. On the left, there's a large heading 'Build fast, responsive sites with Bootstrap' followed by a descriptive paragraph about the toolkit's features. Below this are two buttons: 'Get started' (purple) and 'Download'. On the right, there's a large purple graphic of the letter 'B' with a white outline, set against a background of blurred mobile device icons. At the bottom, there's a section titled 'Installation' with instructions for using npm, Composer, or Meteor, and a link to 'Read installation docs'. There are also code snippets for installing Bootstrap via npm and gem.

# 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

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 zipped. 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.

**Other Related Projects**

**A Brief Look**

**DOM Traversal and Manipulation**

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

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

**Event Handling**

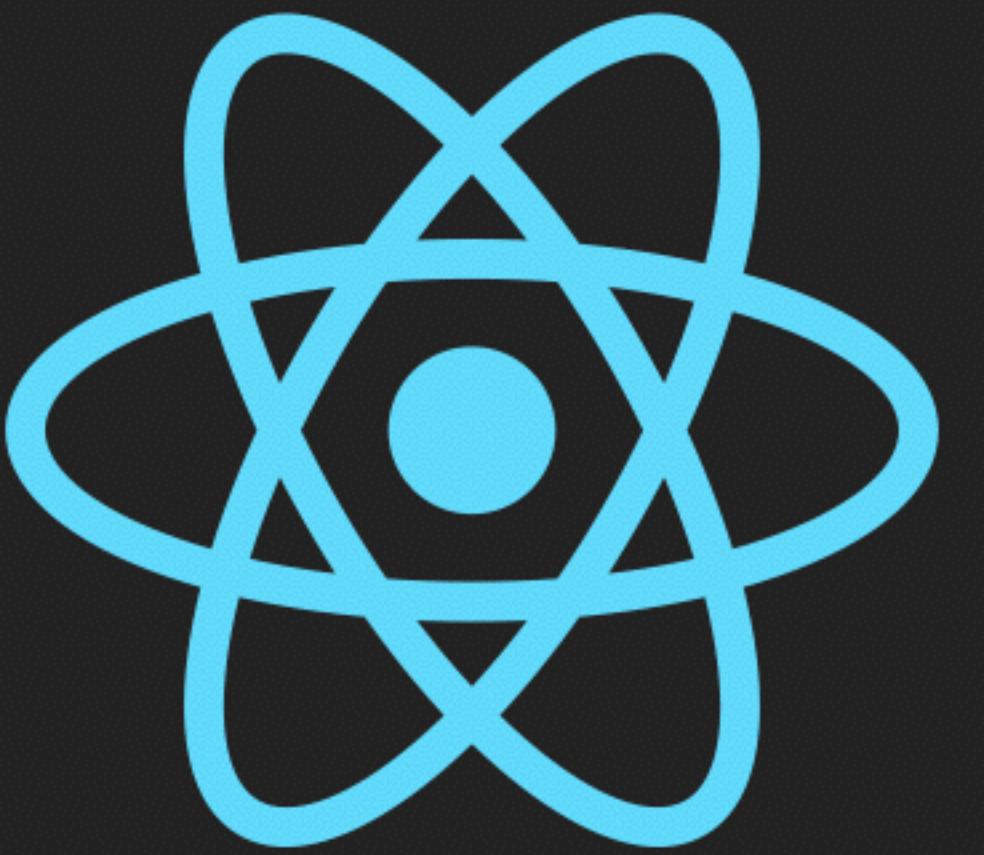
Show the `#banner-message` element that is hidden with `display:none` in its CSS when any button in `#button-container` is clicked.

```
1 | var hiddenBox = $( "#banner-message" );
2 | $( "#button-container button" ).on( "click", function( event ) {
3 |   hiddenBox.show();
4 |});
```

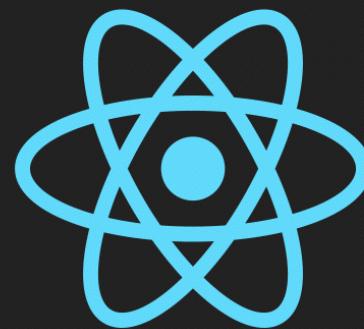


```
$(document).ready(function(){
    $("button").click(function(){
        $("p").hide();
    });
});
```

# 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.

### Componenti

Crea componenti in isolamento e componili per creare UI complesse.

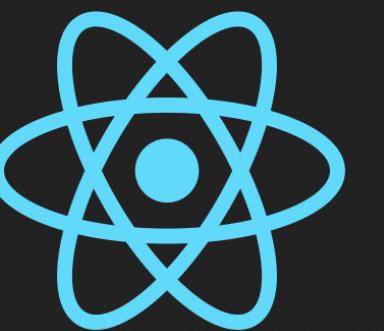
Dato che interazioni e logica per i componenti sono implementate in JavaScript puoi facilmente passare ed accedere strutture dati complesse in vari punti della tua applicazione senza dover salvare informazioni sul DOM.

### 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](#).

# 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" />);
```

RESULT

Ciao Claudia



chrome

[Home page](#) > [Estensioni](#) > [React Developer Tools](#)

## React Developer Tools

[In primo piano](#) 1.395 [i](#)

Strumenti per sviluppatori | 3.000.000+ utenti

[Rimuovi da Google Chrome](#)[Panoramica](#)[Norme di tutela della privacy](#)[Recensioni](#)[Assistenza](#)[Correlati](#)

The screenshot shows the React Developer Tools extension running in a browser. The main window displays a 'todos' application with a header, a list of todos, and a footer. A modal window titled 'Try React DevTools' is open, showing a preview of the todo list. Below the modal, the browser's developer tools are visible, specifically the Components tab. The 'TodoList' component is selected in the tree view on the left. The right panel shows detailed information about this component, including its props (newTodo, onSave, placeholder), hooks (useState), state (State: Try React DevTools), and the components it renders (Header, App). The browser's address bar shows the URL 'https://react-devtools-extension.csb.app/'.



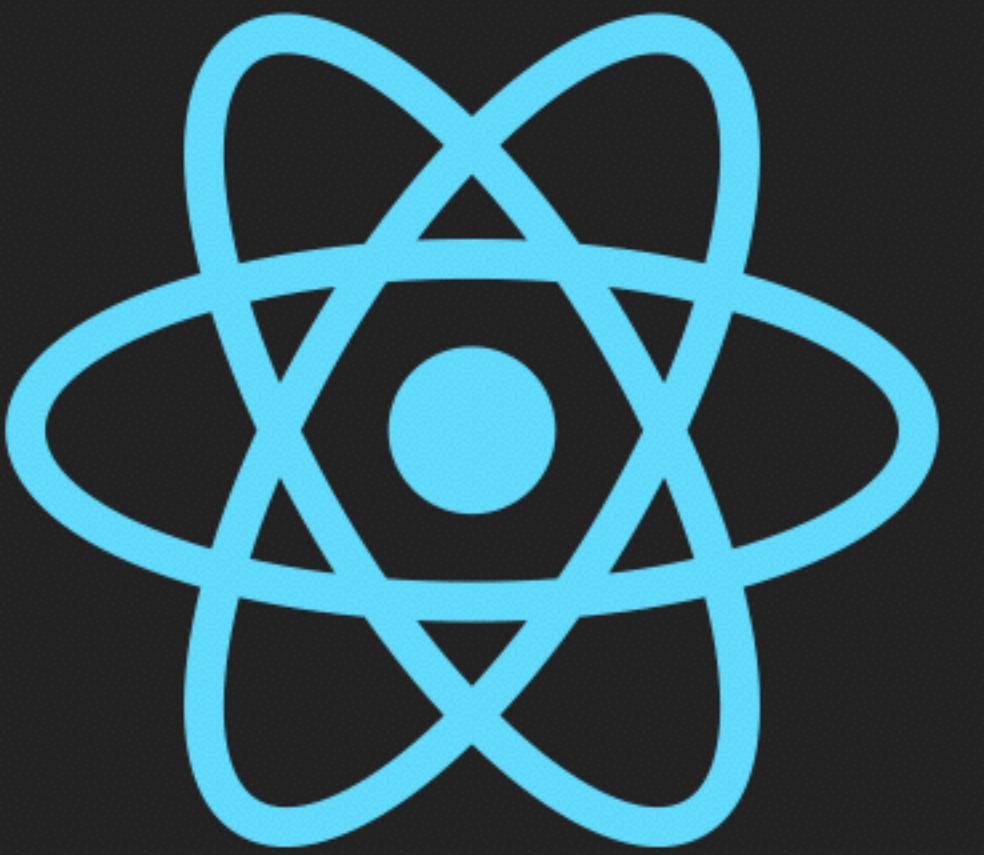
Visual Studio Code



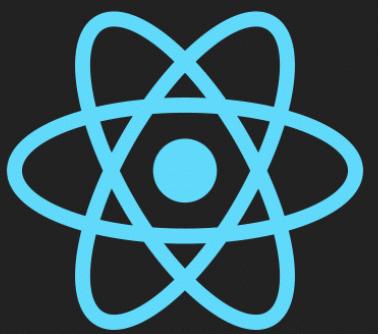
**JS**

{-}  
**ES6**

# React



# Glossario

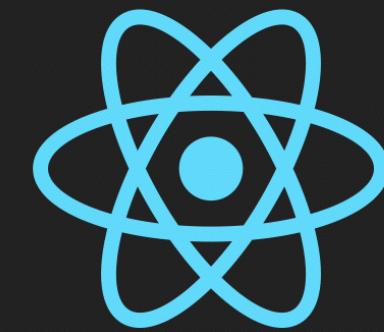


## Applicazione Single-page

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.

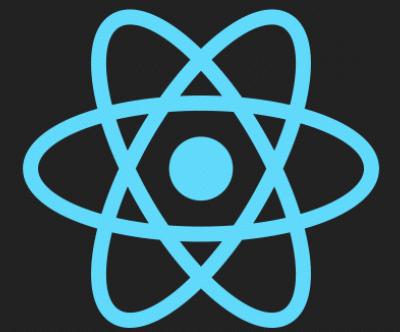
# Glossario



## 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.

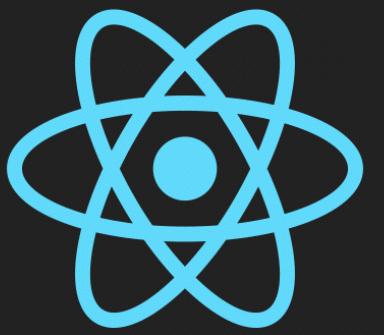
# Glossario



## 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>;
```

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 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.

# 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.

# 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!'
);
```

# 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:

```
// 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 encapsulato in `componenti stateful`.

# 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>;  
}
```

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:

```
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(elemento);
```

# 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(elemento);
```

Ricapitoliamo cosa succede nell'esempio:

1. Richiamiamo `root.render()` con l'elemento `<Ciao nome="Sara" />`.
2. React chiama a sua volta il componente `Welcome` 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

Ad esempio, considera questo componente `Commento`:

```
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 encapsulato. 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. Sostitisci `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`:

```
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:

```
// 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()`:

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

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

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:

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

# 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 fluisce 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

```
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:

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

# 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 è quello di definire il tuo styling in un file `*.css` separato, ed utilizzare ciò che è 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.

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 è 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
```

# Renderizzazione Condizionale

In React, puoi creare componenti distinti che encapsulano il funzionamento di cui hai bisogno. Quindi, puoi renderizzarne solo alcuni, a seconda dello stato della tua applicazione.

La renderizzazione condizionale in React funziona nello stesso modo in cui funzionano le condizioni in JavaScript. Puoi perciò usare operatori come `if` o l'operatore condizionale per creare elementi che rappresentano lo stato corrente cosicché React possa aggiornare la UI di conseguenza.

# Renderizzazione Condizionale

Considera i due componenti:

```
function BenvenutoUtente(props) {
  return <h1>Bentornato/a!</h1>;
}

function BenvenutoOspite(props) {
  return <h1>Autenticati, per favore</h1>;
}
```

Creiamo un componente `Benvenuto` che visualizza l'uno o l'altro dei componenti appena visti a seconda del fatto che l'utente sia autenticato o meno:

```
function Benvenuto(props) {
  const utenteAutenticato = props.utenteAutenticato;
  if (utenteAutenticato) {
    return <BenvenutoUtente />;
  }
  return <BenvenutoOspite />;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
// Prova a cambiare in utenteAutenticato={true}:
root.render(<Benvenuto utenteAutenticato={false} />);
```

# Renderizzazione Condizionale

## Variabili Elemento

Le variabili possono contenere elementi. Ciò ti permette di renderizzare condizionatamente parti del componente mentre il resto dell'output non cambia.

Considera questi due nuovi componenti che rappresentano bottoni di Logout e Login:

```
function BottoneLogin(props) {
  return <button onClick={props.onClick}>Login</button>;
}

function BottoneLogout(props) {
  return <button onClick={props.onClick}>Logout</button>;
}
```

Nell'esempio di seguito, creeremo un componente stateful chiamato **ControlloLogin**.

Esso renderizzerà `<BottoneLogin />` o `<BottoneLogout />` a seconda del suo stato corrente. Renderizzerà inoltre il componente `<Benvenuto />` dell'esempio precedente:

# Renderizzazione Condizionale

```
class ControlloLogin extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {utenteAutenticato: false};
  }

  handleLoginClick() {
    this.setState({utenteAutenticato: true});
  }

  handleLogoutClick() {
    this.setState({utenteAutenticato: false});
  }

  render() {
    const utenteAutenticato = this.state.utenteAutenticato;
    let bottone;

    if (utenteAutenticato) {
      bottone = (
        <BottoneLogout onClick={this.handleLogoutClick} />
      );
    } else {
      bottone = (
        <BottoneLogin onClick={this.handleLoginClick} />
      );
    }

    return (
      <div>
        <Benvenuto utenteAutenticato={utenteAutenticato} />
        {bottone}
      </div>
    );
  }
}

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

# Renderizzazione Condizionale

## Condizione If Inline con Operatore Logico &&

Puoi incorporare espressioni in JSX racchiudendole in parentesi graffe. Lo stesso vale per l'operatore logico JavaScript `&&` che può tornare utile quando vogliamo includere un elemento condizionatamente:

```
function CasellaDiPosta(props) {
  const messaggiNonLetti = props.messaggiNonLetti;
  return (
    <div>
      <h1>Ciao!</h1>
      {messaggiNonLetti.length > 0 && (
        <h2>
          Hai {messaggiNonLetti.length} messaggi non letti.
        </h2>
      )}
    </div>
  );
}

const messaggi = ['React', 'Re: React', 'Re:Re: React'];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<CasellaDiPosta messaggiNonLetti={messaggi} />);
```

# Renderizzazione Condizionale

Funziona perché in JavaScript, `true && espressione` si risolve sempre in `espressione`, mentre `false && espressione` si risolve sempre in `false`.

Per questo, se la condizione è `true`, l'elemento dopo `&&` verrà renderizzato. Se invece è `false`, React lo ignorerà.

Tieni presente che ritornare una espressione `falsy` farà in modo che l'elemento che segue `&&` venga scartato ma ritornerà l'espressione falsy. Nell'esempio di sotto, `<div>0</div>` verrà ritornato dal metodo render.

```
render() {
  const count = 0;
  return (
    <div>
      {count && <h1>Messaggi: {count}</h1>}
    </div>
  );
}
```

# Renderizzazione Condizionale

## Condizioni If-Else Inline con Operatore Condizionale

Un altro metodo per renderizzare condizionatamente elementi inline è quello di usare l'operatore condizionale JavaScript `condizione ? true : false`.

Nell'esempio di seguito, lo useremo per renderizzare condizionatamente un breve blocco di testo.

```
render() {
  const utenteAutenticato = this.state.utenteAutenticato;
  return (
    <div>
      L'utente è <b>{utenteAutenticato ? 'attualmente' : 'non'}</b> autenticato.
    </div>
  );
}
```

# Renderizzazione Condizionale

Può essere usato anche per espressioni più lunghe anche se diventa meno ovvio capire cosa sta succedendo:

```
render() {
  const utenteAutenticato = this.state.utenteAutenticato;
  return (
    <div>
      {utenteAutenticato ? (
        <BottoneLogout onClick={this.handleLogoutClick} />
      ) : (
        <BottoneLogin onClick={this.handleLoginClick} />
      )}
    </div>
  );
}
```

Proprio come in JavaScript, sta a te scegliere lo stile più appropriato a seconda di cosa tu ed il tuo team ritenete più leggibile. Inoltre, ricorda che se le condizioni diventano troppo complesse, potrebbe essere un segnale del fatto che probabilmente è bene estrarre un componente.

# Renderizzazione Condizionale

## Prevenire la Renderizzazione di un Componente

In alcuni rari casi potresti volere che un componente sia nascosto anche se viene renderizzato da un altro componente. Per ottenere questo risultato devi ritornare `null` al posto del suo output di renderizzazione.

Nell'esempio di seguito, il componente `<MessaggioAvviso />` viene renderizzato a seconda del valore della prop chiamata `attenzione`. Se il valore della prop è `false`, il componente non viene renderizzato:

Ritornando `null` dal metodo `render` di un componente, non modifica il comportamento dei metodi di lifecycle del componente. Ad esempio `componentDidUpdate` viene ancora chiamato.

# Renderizzazione Condizionale

```
function MessaggioAvviso(props) {
  if (!props.attenzione) {
    return null;
  }

  return <div className="warning">Attenzione!</div>;
}

class Pagina extends React.Component {
  constructor(props) {
    super(props);
    this.state = {mostraAvviso: true};
    this.handleToggleClick = this.handleToggleClick.bind(
      this
    );
  }

  handleToggleClick() {
    this.setState(state => ({
      mostraAvviso: !state.mostraAvviso,
    }));
  }

  render() {
    return (
      <div>
        <MessaggioAvviso
          attenzione={this.state.mostraAvviso}
        />
        <button onClick={this.handleToggleClick}>
          {this.state.mostraAvviso ? 'Nascondi' : 'Mostra'}
        </button>
      </div>
    );
  }
}

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

# Liste e Chiavi

Prima di iniziare, rivediamo come trasformare le liste in JavaScript.

Nel codice qui sotto, usiamo la funzione `map()` per prendere un array di `numeri` e raddoppiarne i valori. Assegniamo il nuovo array restituito da `map()` alla variabile `lista` e lo stampiamo a console:

```
const numeri = [1, 2, 3, 4, 5];
const lista = numeri.map((numero) => numero * 2);
console.log(lista);
```

Questo codice mostra `[2, 4, 6, 8, 10]` nella console.

Trasformare array in liste di `elementi` con React è quasi identico.

# Liste e Chiavi

## Renderizzare Liste di Componenti

Puoi creare liste di elementi e usarle in JSX usando le parentesi graffe `{}`.

Di seguito, eseguiamo un ciclo sull'array `numeri` usando la funzione JavaScript `map()`.

Ritorniamo un elemento `<li>` per ogni elemento dell'array. Infine, assegniamo l'array risultante a `lista`:

```
const numeri = [1, 2, 3, 4, 5];
const lista = numeri.map((numero) =>
  <li>{numero}</li>
);
```

Includiamo l'intero array `lista` all'interno di un elemento `<ul>`:

```
<ul>{lista}</ul>
```

# Liste e Chiavi

## Semplice Componente Lista

È comune voler renderizzare liste all'interno di un [componente](#).

Possiamo rifattorizzare l'esempio precedente in un componente che accetta un array di [numeri](#) e produce un elenco di elementi.

```
function ListaNumeri(props) {
  const numeri = props.numeri;
  const lista = numeri.map((numero) =>
    <li>{numero}</li>
  );
  return (
    <ul>{lista}</ul>
  );
}

const numeri = [1, 2, 3, 4, 5];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<ListaNumeri numbers={numeri} />);
```

Quando esegui questo codice, appare un *warning* che una chiave (`key`) deve essere fornita per gli elementi della lista. Una "chiave" è una prop speciale di tipo stringa che devi includere quando crei liste di elementi. Discuteremo perché è importante nella prossima sezione.

# Liste e Chiavi

Assegniamo una `key` ai nostri elementi della lista all'interno di `numeri.map()` e risolviamo il problema della chiave mancante.

```
function ListaNumeri(props) {
  const numeri = props.numeri;
  const lista = numeri.map((numero) =>
    <li key={numero.toString()}>
      {numero}
    </li>
  );
  return (
    <ul>{lista}</ul>
  );
}
```

# Liste e Chiavi

## Chiavi

Le chiavi aiutano React a identificare quali elementi sono stati aggiornati, aggiunti o rimossi. Le chiavi dovrebbero essere fornite agli elementi all'interno dell'array per dare agli elementi un'identità stabile:

```
const numeri = [1, 2, 3, 4, 5];
const lista = numeri.map((numero) =>
  <li key={numero.toString()}>
    {numero}
  </li>
);
```

# Liste e Chiavi

Il modo migliore per scegliere una chiave è utilizzare una stringa che identifichi univocamente un elemento della lista tra i suoi elementi adiacenti (*siblings*). L'esempio più comune è usare gli identificativi dei tuoi dati come chiavi:

```
const listaArticoli = articoli.map((articolo) =>
  <li key={articolo.id}>
    {articolo.texto}
  </li>
);
```

# Liste e Chiavi

Quando non disponi di identificativi stabili per gli elementi da renderizzare, puoi assegnare l'indice dell'elemento corrente alla chiave come ultima scelta:

```
const listaArticoli = articoli.map((articolo, indice) =>
  // Fallo solo se gli elementi non hanno identificativi stabili
  <li key={indice}>
    {articolo.texto}
  </li>
);
```

# Liste e Chiavi

## Estrarre Componenti con Chiavi

Le chiavi hanno senso solo nel contesto dell'array circostante.

Per esempio, se estrai un componente `Numero`, dovresti mantenere la chiave sugli elementi `<Numero />` nell'array piuttosto che su l'elemento `<li>` nel `Numero` stesso.

## Esempio: Errato Utilizzo della Chiave

```
function Numero(props) {
  const valore = props.valore;
  return (
    // Sbagliato! Non è necessario specificare la chiave qui:
    <li key={valore.toString()}>
      {valore}
    </li>
  );
}

function ListaNumeri(props) {
  const numeri = props.numeri;
  const lista = numeri.map((numero) =>
    // Sbagliato! La chiave deve essere stata specificata qui:
    <Numero valore={numero} />
  );
  return (
    <ul>
      {lista}
    </ul>
  );
}
```

# Liste e Chiavi

## Esempio: Corretto Utilizzo della Chiave

```
function Numero(props) {
  // Corretto! Non è necessario specificare la chiave qui:
  return <li>{props.valore}</li>;
}

function ListaNumeri(props) {
  const numeri = props.numeri;
  const lista = numeri.map((numero) =>
    // Corretto! La chiave deve essere specificata all'interno dell'array.
    <Numero key={numero.toString()} valore={numero} />
  );
  return (
    <ul>
      {lista}
    </ul>
  );
}
```

# Liste e Chiavi

## Le Chiavi Devono Essere Uniche Tra Gli Elementi Adiacenti

Chiavi usate all'interno degli array dovrebbero essere uniche tra gli elementi adiacenti.

Tuttavia, non hanno bisogno di essere uniche a livello globale. Possiamo usare le stesse chiavi quando creiamo due array diversi:

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.articoli.map((articolo) =>
        <li key={articolo.id}>
          {articolo.titolo}
        </li>
      )}
    </ul>
  );
  const contenuto = props.articoli.map((articolo) =>
    <div key={articolo.id}>
      <h3>{articolo.titolo}</h3>
      <p>{articolo.testo}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      <hr />
      {contenuto}
    </div>
  );
}

const articoli = [
  {id: 1, titolo: 'Ciao Mondo', testo: 'Benvenuto in imparando React!'},
  {id: 2, titolo: 'Installazione', testo: 'Puoi installare React usando npm.'}
];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Blog articoli={articoli} />);
```

# Liste e Chiavi

Le chiavi servono a React come suggerimento, ma non vengono passate ai componenti.

Se hai bisogno di quel valore nel tuo componente, passalo come prop esplicitamente con un nome diverso:

```
const contenuto = articoli.map((articolo) =>
  <Articolo
    key={articolo.id}
    id={articolo.id}
    titolo={articolo.titolo} />
);
```

In questo esempio, il componente `Articolo` può leggere `props.id`, ma non `props.key`.

# Liste e Chiavi

## Incorporare map() in JSX

Nell'esempio di prima abbiamo dichiarato una variabile separata `lista` e l'abbiamo usata nel codice JSX:

```
function ListaNumeri(props) {
  const numeri = props.numeri;
  const lista = numeri.map((numero) =>
    <Numero key={numero.toString()}
      valore={numero} />
  );
  return (
    <ul>
      {lista}
    </ul>
  );
}
```

# Liste e Chiavi

JSX consente di incorporare qualsiasi espressione in parentesi graffe in modo da poter scrivere direttamente il risultato di `map()`:

```
function ListaNumeri(props) {
  const numeri = props.numeri;
  return (
    <ul>
      {numeri.map((numero) =>
        <Numero key={numero.toString()}>
          valore={numero} />
      )}
    </ul>
  );
}
```

# Forms

Gli elementi HTML `form` funzionano in un modo differente rispetto agli altri elementi DOM in React, la motivazione sta nel fatto che gli elementi form mantengono naturalmente uno stato interno. Ad esempio, questo form in puro HTML accetta un singolo nome:

```
<form>
  <label>
    Nome:
    <input type="text" name="nome" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

Questo form si comporta come di consueto, facendo navigare l'utente in una nuova pagina quando viene inviato. In React, se vuoi avere lo stesso comportamento, non c'è bisogno di fare alcuna modifica. Ad ogni modo, potrebbe essere più conveniente avere una funzione JavaScript che gestisce l'invio del form e che ha accesso ai dati inseriti dall'utente. La tecnica standard con cui si può ottenere ciò prende il nome di "componenti controllati".

# Forms

## Componenti Controllati

In HTML, gli elementi di un form come `<input>`, `<textarea>` e `<select>` mantengono tipicamente il proprio stato e lo aggiornano in base all'input dell'utente. In React, lo stato mutabile viene tipicamente mantenuto nella proprietà `state` dei componenti e viene poi aggiornato solo mediante `setState()`.

Possiamo combinare le due cose rendendo lo `state` in React la "singola fonte attendibile" (SSOT). Possiamo poi fare in modo che il componente React che renderizza il form controlli anche cosa succede all'interno del form in risposta agli input dell'utente. In un form, un elemento di input il cui valore è controllato da React in questo modo viene chiamato "componente controllato".

# Forms

Ad esempio, se vogliamo far sì che l'esempio precedente registri il nome inserito, possiamo riscrivere il form sotto forma di componente controllato:

```
class FormNome extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('E\' stato inserito un nome: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Nome:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

# Forms

Dato che l'attributo `value` viene impostato nel nostro elemento form, il valore visualizzato sarà sempre `this.state.value`, rendendo lo stato in React l'unica fonte di dati attendibile. Dato che la funzione `handleChange` viene eseguita ad ogni battitura per aggiornare lo stato di React, il valore visualizzato verrà aggiornato man mano che l'utente preme i tasti.

Con un componente controllato, il valore dell'input viene sempre controllato dallo stato di React. Anche se ciò comporta la battitura di più codice, permette il passaggio del valore anche ad altri elementi della UI, o di resettarlo da altri *event handlers*.

# Forms

## Il Tag Textarea

In HTML, l'elemento `<textarea>` definisce il testo in esso contenuto con i suoi elementi figli:

```
<textarea>
  Nel mezzo del cammin di nostra vita
  mi ritrovai per una selva oscura
  ché la diritta via era smarrita.
</textarea>
```

# Forms

In React, invece, `<textarea>` utilizza l'attributo `value`. Per questo, un form che utilizza una `<textarea>` può essere scritto in modo molto simile a come verrebbe scritto se utilizzasse un semplice input di una sola riga:

```
class FormTema extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Per favore scrivi un tema riguardo il tuo elemento DOM preferito.'
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Un tema è stato inviato: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Tema:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Nota come `this.state.value` viene inizializzato nel costruttore, cosicche la casella di testo è inizializzata con del testo al suo interno.

# Forms

## Il Tag Select

In HTML, `<select>` crea una lista a discesa. Per esempio, questo HTML crea una lista a discesa di gusti:

```
<select>
  <option value="pompelmo">Pompelmo</option>
  <option value="limone">Limone</option>
  <option selected value="cocco">Cocco</option>
  <option value="mango">Mango</option>
</select>
```

Nota come l'opzione Cocco venga preselezionata grazie all'attributo `selected`. React, piuttosto che usare l'attributo `selected`, usa l'attributo `value` dell'elemento radice `select`. Ciò facilita le cose in un componente controllato in quanto bisogna aggiornare lo stato in un posto solo. Ad esempio:

# Forms

```
class FormGusti extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'cocco'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Il tuo gusto preferito è: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Seleziona il tuo gusto preferito:
          <select value={this.state.value}
                  onChange={this.handleChange}>

            <option value="pompelmo">Pompelmo</option>
            <option value="limone">Limone</option>
            <option value="cocco">Cocco</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

# Forms

Ricapitolando, ciò fa sì che `<input type="text">`, `<textarea>` e `<select>` funzionino in modo molto simile - tutti accettano un attributo `value` che puoi utilizzare per implementare un componente controllato.

## **Nota bene**

Puoi passare un array nell'attributo `value`, permettendoti di selezionare opzioni multiple in un tag `select`:

```
<select multiple={true} value={['B', 'C']}>
```

# Forms

## Il Tag Input File

In HTML, un `<input type="file">` permette all'utente di selezionare uno o più file da disco e di inviarli al server o manipolarli in JavaScript mediante le [File API](#).

```
<input type="file" />
```

Dato che il suo valore è in sola-lettura, è un componente **non controllato** in React.

Riprenderemo il discorso riguardo questo ed altri componenti non controllati [in seguito](#).

---

# Forms

## Gestione di Input Multipli

Quando devi gestire diversi elementi `input`, puoi aggiungere un attributo `name` ad ognuno di essi e far sì che la funzione handler controlli cosa fare in base al valore di `event.target.name`.

```
class Prenotazione extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      presente: true,
      numeroOspiti: 2,
    };
    this.handleInputChange = this.handleInputChange.bind(
      this
    );
  }
  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;
    this.setState({
      [name]: value,
    });
  }
  render() {
    return (
      <form>
        <label>
          Sarà presente:
          <input
            name="presente"
            type="checkbox"
            checked={this.state.presente}
            onChange={this.handleInputChange}
          />
        </label>
        <br />
        <label>
          Numero di ospiti:
          <input
            name="numeroOspiti"
            type="number"
            value={this.state.numeroOspiti}
            onChange={this.handleInputChange}
          />
        </label>
      </form>
    );
  }
}
```

# Forms

Nota come abbiamo utilizzato la sintassi ES6 *computed property name* ("nome proprietà calcolato") per aggiornare la rispettiva chiave nello stato a seconda dell'attributo `name` dell'input:

```
this.setState({  
  [name]: value  
});
```

Il che in ES5 corrisponde al codice:

```
var statoParziale = {};  
statoParziale[name] = value;  
this.setState(statoParziale);
```

Inoltre, dato che `setState()` unisce uno stato parziale nello stato corrente automaticamente, dobbiamo chiamarla con le sole parti modificate.

# Forms

## Valore Null in Input Controllati

Specificare la prop `value` in un componente controllato fa sì che l'utente possa cambiare l'input solo quando lo desideri. Se hai specificato un `value` ma l'input è ancora editabile, potresti aver accidentalmente impostato `value` come `undefined` o `null`.

Il codice seguente lo dimostra. (L'input è inizialmente bloccato ma diventa editabile dopo un secondo)

```
ReactDOM.createRoot(mountNode).render(<input value="ciao" />);

setTimeout(function() {
  ReactDOM.createRoot(mountNode).render(<input value={null} />);
}, 1000);
```

# Forms

## Alternative ai Componenti Controllati

Utilizzare componenti controllati può sembrare laborioso a volte, soprattutto perché è necessario scrivere un *event handler* per ogni modo in cui i tuoi dati possono cambiare e perché si deve collegare lo stato di tutti gli input a quello di un componente React. Il tutto diventa particolarmente noioso quando bisogna convertire progetti preesistenti in React, o integrare un'applicazione React con una libreria non-React. In queste situazioni, si potrebbe ricorrere ai [componenti non controllati](#), una tecnica alternativa per implementare forms ed i relativi campi di input.

## Soluzioni Chiavi In Mano

Se stai cercando una soluzione che include la validazione dei dati, il tener traccia dei campi visitati e la sottomissione del form, [Formik](#) è una delle scelte popolari. Comunque, si basa sugli stessi principi dei componenti controllati e della gestione dello stato — ecco perché è bene essere familiari con questi concetti.

# React.Component

# Spostare lo stato

Spesso, l'aggiornamento di diversi componenti dipende dagli stessi dati.

Raccomandiamo di spostare lo stato condiviso in alto nella gerarchia fino al loro antenato più vicino. Vediamo come questo avviene in pratica.

In questa sezione creeremo un calcolatore della temperatura che calcola se l'acqua bolle ad una data temperatura.

Iniziamo con un componente chiamato `VerdettoEbollizione`. Questo, accetta la temperatura tramite la prop `celsius` e ritorna che sia sufficiente a far bollire l'acqua o no:

```
function VerdettoEbollizione(props) {
  if (props.celsius >= 100) {
    return <p>L'acqua bollirebbe.</p>;
  }
  return <p>L'acqua non bollirebbe.</p>;
}
```

# Spostare lo stato

Successivamente, creiamo un componente chiamato `Calcolatore`. Esso renderizza un `<input>` che permette di inserire la temperatura e mantiene il suo valore in `this.state.temperatura`.

Inoltre, restituisce `VerdettoEbollizione` per il valore di input corrente.

```
class Calcolatore extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperatura: ''};
  }

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

  render() {
    const temperatura = this.state.temperatura;
    return (
      <fieldset>
        <legend>Inserisci la temperatura in gradi Celsius:</legend>
        <input
          value={temperatura}
          onChange={this.handleChange} />
        <VerdettoEbollizione
          celsius={parseFloat(temperatura)} />
      </fieldset>
    );
  }
}
```

# Spostare lo stato

## Aggiunta di un secondo input

Il nostro nuovo requisito è che, oltre a un input in gradi Celsius, forniamo un input in gradi Fahrenheit e l'aggiornamento dei due deve essere sincronizzato.

Possiamo iniziare estraendo un componente `InputTemperatura` da `Calcolatore`.

Aggiungiamo una nuova prop `scala` ad esso che può essere "c" o "f":

```
const scale = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

class InputTemperatura extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperatura: ''};
  }

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

  render() {
    const temperatura = this.state.temperatura;
    const scala = this.props.scala;
    return (
      <fieldset>
        <legend>Inserisci la temperatura in gradi {scale[scala]}:</legend>
        <input value={temperatura}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

# Spostare lo stato

Ora possiamo cambiare il `Calcolatore` per renderizzare due input di temperatura separati:

```
class Calcolatore extends React.Component {
  render() {
    return (
      <div>
        <InputTemperatura scala="c" />
        <InputTemperatura scala="f" />
      </div>
    );
  }
}
```

Ora abbiamo due input, ma quando si inserisce la temperatura in uno di essi, l'altro non si aggiorna. Questo non soddisfa il nostro requisito: vogliamo mantenerli sincronizzati.

Inoltre, non possiamo mostrare `VerdettoEbollizione` da `Calcolatore`. Il `Calcolatore` non conosce la temperatura corrente perché è nascosta all'interno di `InputTemperatura`.

# Spostare lo stato

## Scrittura Delle Funzioni Di Conversione

Innanzitutto, scriviamo due funzioni per convertire da Celsius a Fahrenheit e viceversa:

```
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}
```

Queste due funzioni convertono i numeri. Scriviamo un'altra funzione che accetta come argomenti una stringa `temperatura` e una funzione `converti`, e restituisce una stringa. La useremo per calcolare il valore di un input basato su un altro.

La funzione restituisce una stringa vuota per una `temperatura` non valida e arrotonda l'output alla terza cifra decimale:

```
function conversione(temperatura, converti) {
  const input = parseFloat(temperatura);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = converti(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}
```

Ad esempio, `conversione('abc', toCelsius)` restituisce una stringa vuota, e `conversione('10 .22', toFahrenheit)` restituisce `'50.396'`.

# Spostare lo stato

## Spostare lo stato “in alto”

Attualmente, entrambi i componenti `InputTemperatura` mantengono in modo indipendente i loro valori nello stato locale:

```
class InputTemperatura extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperatura: ''};
  }

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

  render() {
    const temperatura = this.state.temperatura;
    // ...
  }
}
```

Tuttavia, vogliamo che i valori di questi due input siano sincronizzati tra loro. Quando aggiorniamo l'input Celsius, l'input Fahrenheit dovrebbe aggiornare la temperatura convertita e viceversa.

# Spostare lo stato

In React, la condivisione dello stato si ottiene spostandolo verso il più vicino antenato comune dei componenti che ne hanno bisogno. Questo processo viene detto "spostare lo stato verso l'alto" (*lifting state up*). Rimuoviamo lo stato locale da `InputTemperatura` e invece lo spostiamo nel `Calcolatore`.

Se il `Calcolatore` possiede lo stato condiviso, diventa la "unica fonte di verità" per la temperatura corrente in entrambi gli input. Può istruire entrambi ad avere valori coerenti l'uno con l'altro. Poiché le props di entrambi i componenti `InputTemperatura` provengono dallo stesso componente `Calcolatore` padre, i due input saranno sempre sincronizzati.

Vediamo come funziona passo dopo passo.

# Spostare lo stato

Vediamo come funziona passo dopo passo.

Per prima cosa sostituiremo `this.state.temperatura` con `this.props.temperatura` nel componente `InputTemperatura`. Per ora, facciamo finta che `this.props.temperatura` esista già, anche se dovremo successivamente passarla dal `Calcolatore`:

```
render() {
  // Prima: const temperatura = this.state.temperatura;
  const temperatura = this.props.temperatura;
  // ...
```

Sappiamo già che le props sono in sola lettura. Quando `temperatura` era nello stato locale, `InputTemperatura` poteva semplicemente chiamare `this.setState()` per cambiarla. Tuttavia, ora che la `temperatura` viene come prop dal componente genitore, `InputTemperatura` non ha alcun controllo su di esso.

In React, questo è solitamente risolto rendendo un componente "controllato". Proprio come nel DOM, `<input>` accetta sia una prop `value` che una prop `onChange`, quindi il componente personalizzato `InputTemperatura` accetta sia la prop `temperatura` che `onChangeTemperatura` dal suo `Calcolatore` padre.

# Spostare lo stato

Ora, quando `InputTemperatura` vuole aggiornare la sua temperatura, chiama `this.props.onChangeTemperatura`:

```
handleChange(e) {
  // Prima: this.setState({temperatura: e.target.value});
  this.props.onChangeTemperatura(e.target.value);
  // ...
```

**Nota:**

Non vi è alcun significato speciale nei nomi delle props `temperatura` o `onChangeTemperatura` nei componenti personalizzati. Avremmo potuto chiamarle in qualsiasi altro modo, come chiamarle `value` e `onChange`, come da convenzione comune.

# Spostare lo stato

La prop `onChangeTemperatura` viene fornita insieme alla prop `temperatura` dal componente padre `Calcolatore`. Gestirà i cambiamenti nel proprio stato locale, ri-renderizzando poi gli input con i nuovi valori. A breve, vedremo la nuova implementazione di `Calcolatore`.

Prima di immergerti nei cambiamenti del `Calcolatore`, ricapitoliamo le modifiche al componente `InputTemperatura`. Abbiamo rimosso lo stato locale e invece di leggere `this.state.temperatura`, ora leggiamo `this.props.temperatura`. Invece di chiamare `this.setState()` quando vogliamo apportare una modifica, ora chiamiamo `this.props.onChangeTemperatura()`, che sarà fornita dal `Calcolatore`:

```
class InputTemperatura extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onChangeTemperatura(e.target.value);
  }

  render() {
    const temperatura = this.props.temperatura;
    const scala = this.props.scala;
    return (
      <fieldset>
        <legend>Inserisci la temperatura in {scale[scala]}:</legend>
        <input value={temperatura}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

# Spostare lo stato

Ora passiamo al componente **Calcolatore**.

Memorizzeremo **temperatura** e **scala** dall'input corrente nel suo stato locale. Questo è lo stato che abbiamo "spostato su" dagli input, e servirà da "unica fonte di verità" per entrambi. È la rappresentazione minima di tutti i dati di cui dobbiamo essere a conoscenza per renderizzare entrambi gli input.

Ad esempio, se inseriamo 37 nell'input Celsius, lo stato del componente **Calcolatore** è:

```
{  
  temperatura: '37',  
  scala: 'c'  
}
```

Se in seguito modifichiamo il campo in Fahrenheit come 212, lo stato del **Calcolatore** è:

```
{  
  temperatura: '212',  
  scala: 'f'  
}
```

Avremmo potuto memorizzare il valore di entrambi gli input, ma non è necessario. È sufficiente memorizzare il valore dell'input modificato più recentemente e la scala che rappresenta. Possiamo quindi dedurre il valore dell'altro input basato sulle sole **temperatura** e **scala** correnti.

# Spostare lo stato

Gli input rimangono sincronizzati perché i loro valori sono calcolati dallo stesso stato:

Ora `this.state.temperatura` e `this.state.scala` in `Calcolatore` si aggiornano indipendentemente dall'input che si modifica. Uno degli input ottiene il valore così com'è, quindi qualsiasi input dell'utente viene mantenuto e l'altro valore di input viene sempre ricalcolato in base ad esso.

```
class Calcolatore extends React.Component {
  constructor(props) {
    super(props);
    this.handleChangeCelsius = this.handleChangeCelsius.bind(this);
    this.handleChangeFahrenheit = this.handleChangeFahrenheit.bind(this);
    this.state = {temperatura: '', scala: 'c'};
  }

  handleChangeCelsius(temperatura) {
    this.setState({scala: 'c', temperatura});
  }

  handleChangeFahrenheit(temperatura) {
    this.setState({scala: 'f', temperatura});
  }

  render() {
    const scala = this.state.scala;
    const temperatura = this.state.temperatura;
    const celsius = scala === 'f' ? conversione(temperatura, toCelsius) :
      temperatura;
    const fahrenheit = scala === 'c' ? conversione(temperatura, toFahrenheit) :
      temperatura;

    return (
      <div>
        <InputTemperatura
          scala="c"
          temperatura={celsius}
          onChangeTemperatura={this.handleChangeCelsius} />
        <InputTemperatura
          scala="f"
          temperatura={fahrenheit}
          onChangeTemperatura={this.handleChangeFahrenheit} />
        <VerdettoEbollizione
          celsius={parseFloat(celsius)} />
      </div>
    );
  }
}
```

# Spostare lo stato

Ricapitoliamo cosa succede quando modifichi un input:

- React chiama la funzione specificata come `onChange` sul DOM `<input>`. Nel nostro caso, questo è il metodo `handleChange` nel componente `InputTemperatura`.
- Il metodo `handleChange` nel componente `InputTemperatura` chiama `this.props.onChangeTemperatura()` con il nuovo valore desiderato. Le sue props, tra cui `onChangeTemperatura`, sono fornite dal suo componente principale, il `Calcolatore`.
- Quando il `Calcolatore` renderizza, esso determina che `onChangeTemperatura` del `InputTemperatura` in Celsius sia il metodo `handleChangeCelsius` di `Calcolatore`, e `onChangeTemperatura` del `InputTemperatura` in Fahrenheit sia il metodo `handleChangeFahrenheit` di `Calcolatore`. Quindi uno di questi due metodi `Calcolatore` viene chiamato a seconda di quale input viene modificato.
- All'interno di questi metodi, il componente `Calcolatore` chiede a React di eseguire nuovamente la renderizzazione chiamando `this.setState()` con il nuovo valore inserito e la scala attuale dell'input appena modificato.
- React chiama il metodo `render` del componente `Calcolatore` per sapere come l'interfaccia utente dovrebbe apparire. I valori di entrambi gli input vengono ricalcolati in base alla temperatura corrente e alla scala attiva. La conversione della temperatura viene eseguita qui.
- React chiama i metodi `render` dei singoli componenti `InputTemperatura` con le nuove props passate dal `Calcolatore`. Vengono a conoscenza di come dovrebbe essere la loro UI.
- React chiama il metodo `render` del componente `VerdettoEbollizione`, passando la temperatura in Celsius come sue props.
- React DOM aggiorna il DOM con il verdetto di ebollizione e abbina i valori di input desiderati. L'input appena modificato riceve il suo valore corrente e l'altro input viene aggiornato con la temperatura dopo la conversione.

Ogni aggiornamento passa attraverso gli stessi passaggi in modo che gli input rimangano sincronizzati.

# Spostare lo stato

Tutti i dati che cambiano in un'applicazione React dovrebbero avere una “unica fonte di verità”. Di solito, lo stato viene prima aggiunto al componente che ne ha bisogno per il rendering. Quindi, se anche altri componenti ne hanno bisogno, puoi spostarlo fino al loro antenato più vicino. Invece di provare a sincronizzare lo stato tra diversi componenti, dovresti affidarti sul [flusso di dati top-down](#).

Spostare lo stato in alto nella gerarchia implica la scrittura di un codice più “standard” rispetto all'approccio *two-way binding* (a doppio senso), ma come vantaggio, trovare e isolare i bug risulta meno laborioso. Poiché ogni stato “vive” in alcuni componenti e solo quel componente può cambiarlo, la fonte di bugs viene notevolmente ridotta. Inoltre, è possibile implementare qualsiasi logica personalizzata per validare o trasformare l'input dell'utente.

Se qualcosa può essere derivato da props o stato, probabilmente non dovrebbe essere nello stato. Ad esempio, invece di memorizzare sia `valoreCelsius` che `valoreFahrenheit`, memorizziamo solo l'ultima `temperatura` modificata e la sua `scala`. Il valore dell'altro input può sempre essere calcolato da loro nel metodo `render()`. Questo ci consente di cancellare o applicare l'arrotondamento all'altro campo senza perdere precisione nell'input dell'utente.

# Spostare lo stato

Quando vedi qualcosa di sbagliato nell'interfaccia utente, puoi utilizzare React Developer Tools per ispezionare le props e spostarti nell'albero finché non si trova il componente responsabile dell'aggiornamento dello stato. Questo ti permette di tracciare i bug alla loro fonte:

The screenshot shows a user interface for a temperature converter. At the top, there are two input fields: "Enter temperature in Celsius:" with a value of "1" and "Enter temperature in Fahrenheit:". Below the inputs, a message says "The water would not boil.".

The React Developer Tools interface is overlaid on the page. The "Elements" tab is selected. In the component tree, the "Calculator" component is expanded, showing its internal structure:

```
<Calculator>
  <div>
    ><TemperatureInput scale="c" temperature="" onTemperatureChange="..."/>
    ><TemperatureInput scale="f" temperature="" onTemperatureChange="..."/>
    ><BoilingVerdict celsius=null>...</BoilingVerdict>
  </div>
</Calculator>
```

The "Calculator" component is highlighted in blue at the bottom of the tree.

On the right side of the tools, the component's state is displayed:

**Props**  
Empty object

**State**  
scale: "c"  
temperature: ""

# Composizione vs Ereditarietà

React ha un potente modello di composizione, raccomandiamo che lo si usi in alternativa all'ereditarietà per riutilizzare codice tra componenti.

In questa sezione, considereremo alcuni problemi nei quali gli sviluppatori che sono ancora agli inizi in React utilizzano l'ereditarietà, mostreremo come si possa invece risolverli con la composizione.

# Composizione vs Ereditarietà

## Contenimento

Esistono componenti che si comportano da contenitori per altri componenti, non possono quindi sapere a priori quali componenti avranno come figli. Si pensi ad esempio a `Sidebar` (barra laterale) oppure `Dialog` (finestra di dialogo) che rappresentano "scatole" generiche.

Raccomandiamo che questi componenti facciano uso della prop speciale `children` per passare elementi figli direttamente nell'output:

```
function BordoFigo(props) {
  return (
    <div className={'BordoFigo BordoFigo-' + props.colore}>
      {props.children}
    </div>
  );
}
```

# Composizione vs Ereditarietà

Ciò permette di passare componenti figli arbitrariamente annidandoli nel codice JSX:

```
function FinestraBenvenuto() {
  return (
    <BordoFigo colore="blue">
      <h1 className="Finestra-titolo">Benvenuto/a!</h1>
      <p className="Finestra-messaggio">
        Ti ringraziamo per questa tua visita nella nostra
        nave spaziale!
      </p>
    </BordoFigo>
  );
}
```

Il contenuto del tag JSX `<BordoFigo>` viene passato nel componente `BordoFigo` come prop `children`. Dato che `BordoFigo` renderizza `{props.children}` all'interno di un `<div>`, gli elementi passati appaiono nell'output finale.

# Composizione vs Ereditarietà

Anche se si tratta di un approccio meno comune, a volte potresti ritrovarti ad aver bisogno di più di un “buco” all’interno di un componente. In questi casi potresti creare una tua convenzione invece di ricorrere all’uso di `children`:

```
function Pannello(props) {
  return (
    <div className="Pannello">
      <div className="Pannello-sinistra">
        {props.sinistra}
      </div>
      <div className="Pannello-destra">{props.destra}</div>
    </div>
  );
}

function App() {
  return (
    <Pannello sinistra={<Contatti />} destra={<Chat />} />
  );
}
```

Gli elementi React `<Contatti />` e `<Chat />` sono dei semplici oggetti, quindi puoi passarli come props esattamente come faresti con altri dati. Questo approccio potrebbe ricordarti il concetto di “slots” in altre librerie, ma non ci sono limitazioni su cosa puoi passare come props in React.

# Composizione vs Ereditarietà

## Specializzazioni

A volte pensiamo ai componenti come se fossero "casi speciali" di altri componenti. Ad esempio, potremmo dire che `FinestraBenvenuto` è una specializzazione di `Finestra`.

In React, ciò si ottiene mediante composizione, dove componenti più "specifici" renderizzano la versione più "generica" configurandola mediante props:

```
function Finestra(props) {
  return (
    <BordoFigo colore="blue">
      <h1 className="Finestra-title">{props.titolo}</h1>
      <p className="Finestra-messaggio">
        {props.messaggio}
      </p>
    </BordoFigo>
  );
}

function FinestraBenvenuto() {
  return (
    <Finestra
      titolo="Benvenuto/a!"
      messaggio="Ti ringraziamo per questa tua visita nella nostra     nave
spaziale!">
    />
  );
}
```

# Composizione

La composizione funziona ugualmente bene per i componenti definiti come classi:

```
function Finestra(props) {
  return (
    <BordoFigo colore="blue">
      <h1 className="Finestra-titolo">{props.titolo}</h1>
      <p className="Finestra-messaggio">
        {props.messaggio}
      </p>
      {props.children}
    </BordoFigo>
  );
}

class FinestraRegistrazione extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Finestra
        titolo="Programma di Esplorazione di Marte"
        messaggio="Qual'è il tuo nome?">
        <input
          value={this.state.login}
          onChange={this.handleChange}
        />
        <button onClick={this.handleSignUp}>
          Registrami!
        </button>
      </Finestra>
    );
  }

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

  handleSignUp() {
    alert(`Benvenuto/a a bordo, ${this.state.login}!`);
  }
}
```

# Composizione vs Ereditarietà

## E per quanto riguarda l'ereditarietà?

In Facebook, usiamo React in migliaia di componenti ma non abbiamo mai avuto alcun caso in cui sarebbe raccomandabile utilizzare gerarchie di ereditarietà per i componenti.

Le props e la composizione ti offrono tutta la flessibilità di cui hai bisogno per personalizzare l'aspetto ed il comportamento di un componente in modo esplicito e sicuro. Ricorda che i componenti possono accettare props arbitrarie, inclusi valori primitivi, elementi React o funzioni.

Se vuoi riutilizzare le funzionalità non strettamente legate alla UI tra componenti, suggeriamo di estrarre tali logiche all'interno di un modulo JavaScript separato. I componenti potranno quindi importarlo ed utilizzare quella funzione, oggetto o classe di cui hanno bisogno, senza dover estendere tale modulo.

# Pensare in React

## Comincia Con Una Bozza

Immaginiamo di avere già a disposizione una API JSON e che il nostro designer ci abbia fornito una bozza come questa:

<input type="text" value="Cerca..."/>	
<input type="checkbox"/> Mostra solo disponibili	
Nome	Prezzo
<b>Attrezzatura Sportiva</b>	
Palla da calcio	\$49.99
Palla da tennis	\$9.99
<b>Palla da canestro</b>	\$29.99
<b>Elettronica</b>	
iPod Touch	\$99.99
<b>iPhone 5</b>	\$399.99
Nexus 7	\$199.99

La nostra API JSON ritorna dati in questa forma:

```
[  
  {categoria: "Attrezzatura Sportiva", prezzo: "$49.99", disponibile: true, nome: "Palla da calcio"},  
  {categoria: "Attrezzatura Sportiva", prezzo: "$9.99", disponibile: true, nome: "Palla da tennis"},  
  {categoria: "Attrezzatura Sportiva", prezzo: "$29.99", disponibile: false, nome: "Palla da canestro"},  
  {categoria: "Elettronica", prezzo: "$99.99", disponibile: true, nome: "iPod Touch"},  
  {categoria: "Elettronica", prezzo: "$399.99", disponibile: false, nome: "iPhone 5"},  
  {categoria: "Elettronica", prezzo: "$199.99", disponibile: true, nome: "Nexus 7"}  
];
```

# Pensare in React



Abbiamo identificato cinque componenti nella nostra applicazione. In corsivo, la parte del modello dati rappresentata da ogni componente. I numeri nell'immagine corrispondono ai numeri di seguito.

1. **TabellaProdottiRicercabile** (arancione): contiene l'intero esempio
2. **BarraRicerca** (blu): riceve tutti gli *input dell'utente*
3. **TabellaProdotti** (verde): visualizza e filtra la *lista dei prodotti* a seconda dell'*input dell'utente*
4. **RigaCategoriaProdotti** (turchese): visualizza una testata per ogni *categoria*
5. **RigaProdotto** (rosso): visualizza una riga per ogni *prodotto*

# Pensare in React



Se dai un'occhiata a `TabellaProdotti`, noterai che la testata della tabella ( contenente le etichette "Nome" e "Prezzo") non rappresenta un componente a se stante. Si tratta di una questione soggettiva, e ci sono argomenti validi in entrambi i sensi. In questo esempio, l'abbiamo lasciata come parte di `TabellaProdotti` perché fa parte della renderizzazione della *lista dei prodotti* che è una responsabilità di `TabellaProdotti`. Comunque, qualora questa testata dovesse diventare complessa (per esempio se volessimo aggiungere la gestione dell'ordinamento per colonna), avrebbe sicuramente senso creare un suo proprio componente `TestataTabellaProdotti`.

# Pensare in React



Adesso che abbiamo identificato i componenti nella nostra bozza, ordiniamoli gerarchicamente. I componenti che appaiono all'interno di un altro componente nella bozza, devono essere loro figli nella gerarchia:

- TabellaProdottiRicercabile
  - BarraRicerca
  - TabellaProdotti
    - RigaCategoriaProdotti
    - RigaProdotto

# Pensare in React

## Passo 2: Sviluppa Una Versione Statica in React

● ● ●

```
1 import React from 'react';
2 import ReactDOM from 'react-dom';
3
4 class RigaCategoriaProdotti extends React.Component {
5   render() {
6     const categoria = this.props.categoria;
7     return (
8       <tr>
9         <th colSpan="2">{categoria}</th>
10      </tr>
11    );
12  }
13}
14
15 class RigaProdotto extends React.Component {
16   render() {
17     const prodotto = this.props.prodotto;
18     const nome = prodotto.disponibile ? (
19       prodotto.nome
20     ) : (
21       <span style={{color: 'red'}}>{prodotto.nome}</span>
22     );
23
24     return (
25       <tr>
26         <td>{nome}</td>
27         <td>{prodotto.prezzo}</td>
28       </tr>
29     );
30   }
31 }
```

```
33 class TabellaProdotti extends React.Component {
34   render() {
35     const righe = [];
36     let ultimaCategoria = null;
37
38     this.props.prodotti.forEach(prodotto => {
39       if (prodotto.categoria !== ultimaCategoria) {
40         righe.push(
41           <RigaCategoriaProdotti
42             categoria={prodotto.categoria}
43             key={prodotto.categoria}
44           />
45         );
46       }
47       righe.push(
48         <RigaProdotto
49           prodotto={prodotto}
50           key={prodotto.nome}
51         />
52       );
53       ultimaCategoria = prodotto.categoria;
54     });
55
56     return (
57       <table>
58         <thead>
59           <tr>
60             <th>Nome</th>
61             <th>Prezzo</th>
62           </tr>
63         </thead>
64         <tbody>{righe}</tbody>
65       </table>
66     );
67   }
68 }
```

# Pensare in React

## Passo 2: Sviluppa Una Versione Statica in React

```
70 class BarraRicerca extends React.Component {  
71   render() {  
72     return (  
73       <form>  
74         <input type="text" placeholder="Cerca..." />  
75         <p>  
76           <input type="checkbox" /> Mostra solo disponibili  
77         </p>  
78       </form>  
79     );  
80   }  
81 }  
82  
83 class TabellaProdottiRicercabile extends React.Component {  
84   render() {  
85     return (  
86       <div>  
87         <BarraRicerca />  
88         <TabellaProdotti prodotti={this.props.prodotti} />  
89       </div>  
90     );  
91   }  
92 }
```

```
94 const PRODOTTI = [  
95   {  
96     categoria: 'Attrezzatura Sportiva',  
97     prezzo: '$49.99',  
98     disponibile: true,  
99     nome: 'Palla da calcio',  
100   },  
101   {  
102     categoria: 'Attrezzatura Sportiva',  
103     prezzo: '$9.99',  
104     disponibile: true,  
105     nome: 'Palla da tennis',  
106   },  
107   {  
108     categoria: 'Attrezzatura Sportiva',  
109     prezzo: '$29.9',  
110     disponibile: false,  
111     nome: 'Palla da canestro',  
112   },  
113   {  
114     categoria: 'Elettronica',  
115     prezzo: '$99.99',  
116     disponibile: true,  
117     nome: 'iPod Touch',  
118   },  
119   {  
120     categoria: 'Elettronica',  
121     prezzo: '$399.99',  
122     disponibile: false,  
123     nome: 'iPhone 5',  
124   },  
125   {  
126     categoria: 'Elettronica',  
127     prezzo: '$199.99',  
128     disponibile: true,  
129     nome: 'Nexus 7',  
130   },  
131 ];  
132  
133 ReactDOM.render(  
134   <TabellaProdottiRicercabile prodotti={PRODOTTI} />,  
135   document.getElementById('container')  
136 );
```

# Pensare in React

Adesso che hai la gerarchia dei componenti, è ora di implementare la tua applicazione. Il modo più facile è quello di sviluppare una versione che riceve dati dal modello e renderizza la UI senza alcuna interattività. È bene mantenere separati questi processi perché implementare una versione statica richiede la scrittura di molto codice non complesso, aggiungere interattività, al contrario, richiede di spremersi le meningi e la scrittura di poco codice. Vediamo perché.

Per costruire una versione statica della tua applicazione che renderizza il tuo modello dati, devi implementare componenti che riutilizzano altri componenti e passano dati usando *props*. Le *props* sono un modo di passare dati da genitore a figlio. Se sei familiare con il concetto di *state*, **evita di utilizzarlo del tutto** nella costruzione di una versione statica. Lo stato (nella forma della proprietà `state`) è riservato all'interattività, ovvero, dati che cambiano nel tempo. Dato che si tratta di una versione statica, non ne abbiamo bisogno.

# Pensare in React

Puoi cominciare partendo dall'alto (top-down) o dal basso (bottom-up). Il che significa che puoi cominciare con l'implementazione dei componenti più un alto nella gerarchia (per esempio cominciando da `TabellaProdottiRicercabile`) o con quelli più in basso al suo interno (`RigaProdotto`). Nei casi più semplici, è meglio preferire l'approccio top-down, nei progetti più grandi, è più facile l'approccio bottom-up e la contestuale scrittura di test mano che si implementa.

Alla fine di questo passo, avrai una libreria di componenti riutilizzabili che renderizza il tuo modello dati. I componenti avranno solo metodi `render()` dati che è una versione statica dell'applicazione. Il componente al vertice della gerarchia (`TabellaProdottiRicercabile`) riceverà il tuo modello dati come prop. Qualora tu cambiassi il modello dati sottostante, richiamando `root.render()` di nuovo, la UI verrà aggiornata. Vedrai come la UI si aggiorna al fine di individuare i necessari cambiamenti dato che quanto sta accadendo non è per nulla complicato. Il **flusso dati monodirezionale** di React (noto anche come *one-way binding*) mantiene ogni cosa modulare e veloce.

# Pensare in React

## Passo 3: Identifica la Minima (ma completa) Rappresentazione dello Stato della UI

Per rendere la tua UI interattiva, devi fare in modo che sia possibile alterare il modello dati sottostante. React rende ciò possibile grazie all'uso di **state**.

Al fine di implementare l'applicazione correttamente, devi innanzitutto pensare alla minima parte di stato mutabile del quale la tua applicazione ha bisogno. La chiave qui è DRY: Don't Repeat Yourself. Individua la minima rappresentazione di stato richiesta dall'applicazione e calcola tutto il resto al bisogno. Ad esempio, se stai costruendo una "Lista delle cose da fare"; mantieni solo l'array degli elementi della lista; non mantenere una variabile separata nello stato per il conteggio, utilizza semplicemente la proprietà `length` dell'array.

# Pensare in React

Pensa a tutte le parti di dati nell'applicazione d'esempio. Abbiamo:

- La lista originale dei prodotti
- Il testo di ricerca inserito dall'utente
- Il valore della checkbox
- La lista filtrata dei prodotti

Vediamoli uno alla volta al fine di individuare quali rappresentano stato. Poniti tre domande per ognuno:

1. Viene ricevuto da un genitore via props? In tal caso, probabilmente non si tratta di stato.
2. Rimane invariato nel tempo? Se sì, probabilmente non si tratta di stato.
3. Puoi derivarlo in base ad altre parti di state o props nel tuo componente? Se è così, non è stato.

La lista originale dei prodotti viene ricevuta come props, non si tratta quindi di stato. Il testo di ricerca e la checkbox sembrano far parte dello stato dato che cambiano nel tempo e non possono essere derivati da nulla. Infine, la lista filtrata dei prodotti non fa parte dello stato visto che più essere derivata dalla lista originale dei prodotti, dal testo di ricerca e dal valore della checkbox.

# Pensare in React

## Passo 4: Identifica Dove Posizionare Il tuo Stato

OK, abbiamo individuato la minima rappresentazione di stato dell'applicazione. Adesso, dobbiamo trovare quale componente muta, o *possiede*, questo stato.

Ricorda: React è del tutto basato sul concetto di flusso dati unidirezionale dall'alto verso il basso nella gerarchia dei componenti. Può non risultare immediatamente chiaro quale componente deve mantenere quale parte di stato. **Questa è la parte più complicata da capire per chi è agli inizi**, segui questi passi come linea guida:

Per ogni parte di stato nella tua applicazione:

- Identifica ogni componente che renderizza qualcosa in base a quello stato.
- Identifica un componente proprietario comune (un singolo componente al di sopra di tutti i componenti che richiedono quello stato nella gerarchia).
- Lo stato dovrà risiedere nel proprietario comune oppure in un altro componente più in alto nella gerarchia.
- Se non riesci ad individuare facilmente il componente che sensatamente dovrebbe mantenere lo stato, crea un nuovo componente per fare ciò e posizionalo da qualche parte nella gerarchia al di sopra del componente proprietario comune.

# Pensare in React

Applichiamo questa strategia nella nostra applicazione:

- `TabellaProdotti` ha bisogno di filtrare la lista dei prodotti in base allo stato e `BarraRicerca` deve visualizzare il testo di ricerca e lo stato della checkbox.
- Il componente proprietario comune è `TabellaProdottiRicercabile`.
- Concettualmente possiamo dire che ha senso mantenere il testo di ricerca e lo stato della checkbox all'interno di `TabellaProdottiRicercabile`

Bene, abbiamo deciso che il nostro stato vive in `TabellaProdottiRicercabile`. Prima di tutto, aggiungi una proprietà d'istanza `this.state = {testoRicerca: '', soloDisponibili: false}` nel `constructor` di `TabellaProdottiRicercabile` al fine di riflettere lo stato iniziale della tua applicazione. Poi, passa `testoRicerca` e `soloDisponibili` a `TabellaProdotti` e `BarraRicerca` come prop. Infine, usa queste props per filtrare le righe in `TabellaProdotti` e impostare i valori nel form in `BarraRicerca`.

Puoi cominciare a vedere come si comporta l'applicazione: imposta `testoRicerca` a "Palla" e aggiornala. Vedrai che la tabella dati è stata aggiornata correttamente.

# Pensare in React

## Passo 5: Invertire il Flusso Dati

Fino ad ora, abbiamo implementato una applicazione che renderizza correttamente con props e state che fluiscono in basso nella gerarchia. Adesso è il momento di supportare il flusso inverso: dobbiamo fare in modo che i componenti form più in basso nella gerarchia, possano aggiornare lo stato in `TabellaProdottiRicercabile`.

React rende questo flusso dati esplicito in modo da facilitare la comprensione del funzionamento del programma, tuttavia richiede la scrittura di un po' più codice rispetto ad altre soluzioni con flusso dati bidirezionale (*two-way data binding*).

Se provi a scrivere qualcosa o a selezionare la casella nella precedente versione (passo 4), noterai che React ignora completamente il tuo input. Si tratta di un fatto intenzionale, abbiamo infatti impostato la prop `value` dell' `input` per essere sempre uguale allo `state` che riceve da `TabellaProdottiRicercabile`.

Proviamo a pensare a cosa vogliamo far sì che avvenga. Vogliamo fare in modo che a seconda di come l'utente alteri il form, lo stato verrà alterato di conseguenza. Dato che i componenti possono alterare solamente il proprio stato, `TabellaProdottiRicercabile` passerà `callbacks` a `BarraRicerca` che verranno invocate ogni volta lo stato deve essere aggiornato. Possiamo utilizzare l'evento `onChange` degli `input` per ricevere tale notifica. Le callbacks passate da `TabellaProdottiRicercabile` chiameranno `setState()` facendo sì che la applicazione venga aggiornata.

# Pensare in React

# Pensare in React

# Pensare in React

# Pensare in React

# Frammenti

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

```
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}
```