Capitolo 1

La prima mezz'ora con React

Il primo contatto con una nuova tecnologia è sempre un momento critico, perché può causare il nostro amore o odio verso questa. Lunghe configurazioni, istruzioni poco chiare o troppe informazioni da assorbire tutte in una volta possono farci perdere l'entusiasmo iniziale. Inoltre, avvicinarsi a React richiede anche di conoscere già alcune nozioni, senza le quali non sarebbe poi chiaro che cosa stiamo facendo o che cosa sta accadendo nel browser.

Per evitare tutti questi problemi, di solito, si mostra solo come qualcosa deve essere fatta per funzionare, tralasciando le spiegazioni e i dettagli troppo complicati.

In questo capitolo (e in generale in tutto il libro) cercherò, invece, di seguire un'altra via: verranno introdotti i concetti uno alla volta in un contesto pratico, affiancati a spiegazioni e brevi approfondimenti. In questo modo potrete acquisire le conoscenze in modo graduale mentre le applicate, così da avere una maggiore consapevolezza di quello che state facendo e di quello sta accadendo.

Per semplificare e velocizzare al massimo la nostra esperienza, in questo capitolo useremo un editor online già configurato, così da poter sperimentare subito le funzionalità base di React senza nessuno sforzo aggiuntivo.

Alla fine di questo capitolo avrete acquisito le conoscenze minime necessarie per comprendere come funziona React che vi permetteranno di continuare questo percorso di sviluppo di un'applicazione e di scoperta delle tante sue caratteristiche.

- Il codice necessario per eseguire un'applicazione React in una pagina HTML.

In questo capitolo

- Che cosa sono e come usare gli elementi React.
- Le caratteristiche degli elementi: attributi, stile, immagini.
- La differenza con gli elementi HTML.
- La differenza tra DOM e virtual DOM.
- Come usare JSX e perché è utile.
- Come creare componenti e la differenza con gli elementi.
- Come usare altri componenti all'interno di un componente.

Alla fine del capitolo, oltre ad aver compreso tutti questi concetti, avremo anche creato i primi componenti per mostrare un messaggio di benvenuto. Questi componenti rappresentano i primi tasselli dell'applicazione che svilupperemo.

Il primo rendering

Per provare React senza dover installare nulla sul computer useremo *Codepen*, che è un ambiente di sviluppo online. Accedendo all'indirizzo `https://codepen.io/lfoo/pen/xxLyEwJ` si aprirà una pagina con l'editor già pronto per eseguire il nostro primo esperimento con React. La pagina, che vediamo rappresentata nella Figura 1.1, ha due sezioni con del codice e una con l'anteprima del risultato (il titolo dell'applicazione *Todo app*).

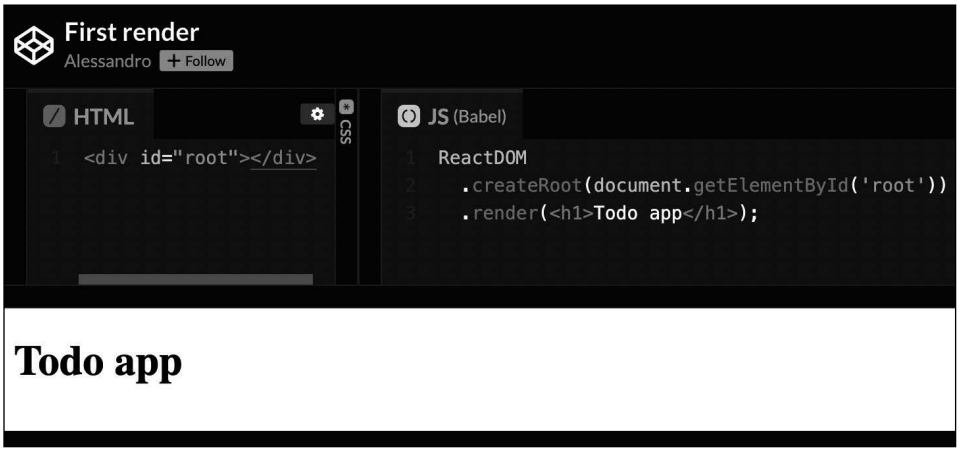

Figura 1.1 La prima versione della nostra applicazione.

Nel blocco HTML abbiamo il contenuto dell'elemento `<body>` della pagina in cui vogliamo visualizzare l'applicazione React.

> **NOTA**
>
> Codepen inserisce per noi il resto della struttura della pagina HTML, così da permetterci di scrivere solo la parte che ci interessa. Grazie a questo automatismo possiamo rimanere concentrati solo sugli elementi necessari all'applicazione.

La sezione HTML contiene un solo elemento `div` con id `root`, che useremo per far inserire a React il suo output.

Listato 1.1 Il codice HTML.

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

La sezione con il codice JavaScript è più interessante, perché il codice è quello che genera l'output *Todo app* che vediamo nello spazio dell'anteprima.

Listato 1.2 Il codice JavaScript.

```
ReactDOM
    .createRoot(document.getElementById('root'))
    .render(<h1>Todo app</h1>);
```

Modificando il testo possiamo vedere che il risultato si aggiorna in tempo reale: ogni volta che il codice nel blocco HTML o JavaScript viene cambiato, Codepen ricarica la pagina per noi, eseguendo di nuovo tutto il codice e producendo un risultato aggiornato.

ATTENZIONE

All'interno della sezione JavaScript di Codepen abbiamo scritto del codice che usa l'oggetto ReactDOM, senza, però, che questo sia stato definito da nessuna parte. Il motivo per cui non vediamo nessun errore è dato dalla configurazione di questa pagina di Codepen. Infatti, è stata configurata per caricare la libreria di React e renderla disponibile nel codice. Nel prossimo capitolo importeremo questa libreria in modo esplicito.

Dall'oggetto `ReactDOM` viene richiamata la funzione `createRoot` che crea, nell'elemento selezionato del DOM (il `div` con id `root`), la *radice* (il contenitore) in cui visualizzare gli elementi generati da React. La funzione `render` poi, mostra all'interno di questa radice l'elemento passato come parametro (in questo caso `<h1>Todo app</h1>`), permettendoci di vedere il risultato nel browser.
L'elemento del DOM in cui visualizzare l'output di React viene recuperato con l'apposita funzione `document.getElementById` fornita dal browser.

ATTENZIONE

Un *elemento* React (come `<h1>Todo app</h1>`) non è un normale elemento del DOM, ma è un oggetto JavaScript; React si occuperà di mostrarlo per noi nel DOM del browser, quando necessario.

Dov'è finita la funzione ReactDOM.render()?

Nelle versioni di React precedenti alla 18, il rendering dell'applicazione veniva effettuato tramite la funzione `ReactDOM.render`, in modo più breve rispetto a quanto si fa ora.

```
ReactDOM.render(
    <h1>Todo app</h1>,
    document.getElementById('root')
);
```

Quando si utilizza la vecchia sintassi con React 18 (o superiore) la console del browser mostra un warning.

```
Warning: ReactDOM.render is no longer supported in React 18. Use createRoot instead.
Until you switch to the new API, your app will behave as if it's running React 17. Learn
more: https://reactjs.org/link/switch-to-createroot
```

Le ragioni di questo cambiamento nelle API esposte da React sono diverse:

- rendere esplicita la creazione di una root;
- supportare meglio le funzionalità future di React;
- uniformare l'importazione di ReactDom nel client e nel server (per chi utilizza SSR).

Potete trovare tutte le motivazioni e spiegazioni approfondite di questa modifica a questo indirizzo: `https://github.com/reactwg/react-18/discussions/5`.

Da React al browser

Per mostrare una pagina web, il browser trasforma il codice HTML in un modello chiamato DOM (*Document Object Model*) `https://developer.mozilla.org/en-US/docs/Glossary/DOM`; dopo aver creato questo modello, lo interpreta e ne effettua il rendering, cioè lo trasforma in quello che vediamo sullo schermo (caratteri, immagini e così via).
Prima dell'avvento di React, la maggior parte delle librerie JavaScript per costruire interfacce utente manipolava in modo diretto gli elementi del DOM, aggiungendoli, rimuovendoli e modificandoli. Questo approccio, anche se pratico, è limitato dalle prestazioni del DOM che, di solito, non sono elevate. La conseguenza principale della manipolazione diretta del DOM è la scarsa reattività delle interfacce, che risulteranno sempre molto lente nell'aggiornarsi dopo le interazioni dell'utente.
Anche React manipola il DOM, ma in modo diverso, usando la tecnica del *virtual DOM*. Invece di modificare direttamente il contenuto del DOM, React mantiene in memoria un modello del DOM (il virtual DOM) e gli applica tutte le modifiche quando l'interfaccia cambia. Questo modello contiene al suo interno tutti gli elementi dell'interfaccia, rappresentati come normali oggetti JavaScript. Dopo aver modificato questo modello in memoria, tramite uno speciale algoritmo React calcola la differenza tra il DOM del browser e il suo modello. Nel DOM del browser verranno aggiornate solo le parti che risultano diverse tra i due modelli, con un notevole guadagno dal punto di vista delle prestazioni.

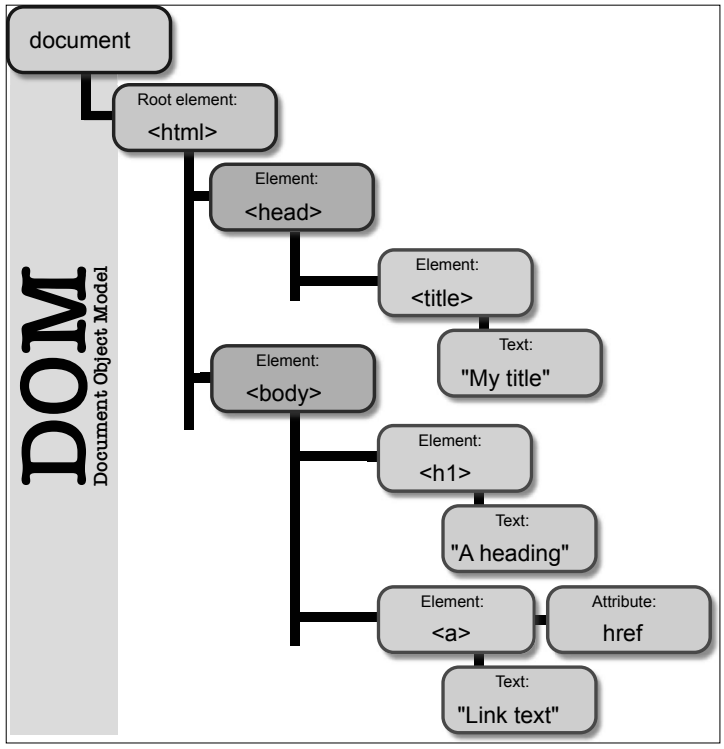

Figura 1.2 Una rappresentazione grafica del DOM - Birger Eriksson CC BY-SA 3.0 https://commons.wikimedia.org/wiki/File:DOM-model.svg.

Più avanti toccheremo con mano le conseguenze positive derivanti dall'uso del virtual DOM (soprattutto in termini di velocità di aggiornamento) e ne approfondiremo il funzionamento. Quello che ci interessa sapere al momento è che questo modello ideale dalla nostra interfaccia utente viene usato per modificare il DOM del browser solo quando è necessario, attraverso una fase di *rendering*. Nel nostro caso, questa operazione è gestita dalla funzione `render` che abbiamo visto poco fa.

JSX

Possiamo riscrivere il nostro codice in modo che l'elemento React da mostrare sia assegnato a una costante e che la funzione di rendering la riceva come parametro.

```
const helloElement = <h1>Todo app</h1>;
ReactDOM
    .createRoot(document.getElementById('root'))
    .render(helloElement);
```

Ora dovrebbe essere più evidente che `<h1>Todo app</h1>` non solo è un elemento React, ma è anche codice JavaScript valido, perché viene eseguito senza errori. In realtà si trat-

ta di codice *JSX*, cioè una speciale estensione di JavaScript che ci permette di scrivere codice simil-HTML all'interno di JavaScript.
Visto che gli elementi React sono semplici oggetti, per crearli dovremo usare una funzione o costruirli a mano (conoscendone la struttura interna). Usando JSX, invece, usiamo una scorciatoia che ci evita di chiamare una funzione, perché JSX lo fa implicitamente al posto nostro. Senza JSX avremmo dovuto creare il nostro elemento React in modo meno "simile" al codice HTML, usando la funzione `React.createElement`. La differenza tra i due metodi è visibile nei due elementi creati qui sotto che, pur essendo creati in modo differente, sono equivalenti:

```
const helloElementJSX = <h1>Todo app</h1>;
const helloElementNoJSX = React.createElement(
    'h1',
    {},
    'Todo app'
);
```

È facile intuire che usando JSX possiamo definire le nostre interfacce in JavaScript in modo simile a quello che faremmo usando HTML. Proviamo quindi ad aggiungere un paragrafo con un messaggio di benvenuto per l'utente dell'applicazione:

```
const helloElement = <div>
    <h1>Todo app</h1>
    <p>Benvenuto!</p>
</div>;

ReactDOM
    .createRoot(document.getElementById('root'))
    .render(helloElement);
```

Non appena completata la modifica, vedremo apparire i due elementi nella sezione dedicata all'anteprima.

> **SUGGERIMENTO**
>
> Se tutti questi *tag* rendono difficile capire dove inizia e dove finisce un elemento, possiamo sempre aggiungere delle parentesi intorno alle espressioni JSX per delimitarle: `const helloElement = (<div>...</div>)`. Possiamo farlo perché, anche se JSX è un'estensione di JavaScript, sempre di codice JavaScript si tratta.

JSX, XHP, E4X

JSX non è un'idea completamente nuova di React, perché ha almeno due parenti noti nel recente passato.
Il più recente è XHP (`https://github.com/hhvm/xhp-lib`), una libreria sviluppata nel 2010 sempre all'interno di Facebook, che permette di scrivere codice XML all'interno di codice Hack (prima PHP). XHP a sua volta è stato ispirato da E4X (`https://engineering.fb.com/2010/02/09/developer-tools/xhp-a-new-way-to-write-php`).

ECMAScript for XML, o E4X, è un'estensione di JavaScript creata nel 2002 per utilizzare e manipolare XML nel codice JavaScript. Dopo qualche anno, fu rimosso dall'unico browser che lo supportava (Firefox). Potete leggere di più nella pagina Wikipedia dedicata (`https://en.wikipedia.org/wiki/ECMAScript_for_XML`) e nella *issue* di Mozilla per la sua rimozione da Firefox (`https://bugzilla.mozilla.org/show_bug.cgi?id=695577`).
JSX è stato influenzato da E4X, tanto da essere stato etichettato come *E4X The Good Parts* (`https://blog.vjeux.com/2013/javascript/jsx-e4x-the-good-parts.html`); nel repository di React si trovano tracce della loro parentela (`https://github.com/facebook/react/search?q=e4x&type=issues`).

JSX e attributi

Come per i normali elementi HTML, anche per quelli definiti in JSX è possibile usare degli attributi: `<h1 id="title">Todo app</h1>`. Tuttavia, essendo JSX un'estensione di JavaScript (e non di HTML), segue la convenzione (comune nel mondo JavaScript) di avere le proprietà composte da più parole scritte in *camelCase* (*`nomeAttributo`*) invece del modo *`nome-attributo`* usato in HTML (separato da un trattino). Inoltre, anche alcuni attributi che in HTML sono composti da una sola parola, in JSX sono trasformati in camelCase. Per esempio, l'attributo `tabindex` usato in HTML, diventa `tabIndex` in JSX.
In React è quindi possibile utilizzare tutti i normali attributi HTML, ma alcuni di questi richiedono un diverso utilizzo rispetto alle loro controparti HTML.

className

Un caso speciale è quello dell'attributo HTML `class`, che in JSX diventa `className` anche se nella versione originale non è composto da due parole.

```
const helloElement = <div>
    <h1 className="title">Todo app</h1>
    <p className="message">Benvenuto!</p>
</div>;
```

Se provassimo a usare `class` in JSX riceveremmo un warning nella console del browser. Con questo messaggio, React ci avverte che stiamo usando l'attributo sbagliato.

> **NOTA**
>
> Il motivo della differenza di nomi tra HTML e JSX è la volontà di React di adeguarsi alla proprietà `className` dell'interfaccia `Element` del DOM `https://developer.mozilla.org/en-US/docs/Web/API/Element/className#Notes`.

style

Per dare uno stile *inline* a un elemento JSX, possiamo usare l'attributo `style` al posto di `className`, così da scrivere solo le proprietà CSS senza dover creare una classe:

```
<p style={{color: 'red'}}>Benvenuto!</p>
```

La sintassi `style={{ }}` con le doppie graffe è necessaria perché, rispetto a quanto fatto con `className`, qui non stiamo passando una stringa come valore dell'attributo. La coppia più esterna di graffe indica a JSX che il valore dell'attributo sarà quello prodotto dal codice JavaScript contenuto al suo interno. La coppia più interna, invece, è quella che usiamo per definire un normale oggetto JavaScript con le sue proprietà. In questo caso l'oggetto ha una sola proprietà `color` con valore `'red'` (una normale stringa).
Nell'oggetto `style` possono essere usate tutte le normali proprietà CSS, ma i loro nomi vanno convertiti in *camelCase* (come per gli attributi HTML). Quindi, per esempio, `font-size` diventerà `fontSize`.

```
// HTML
<p style="color: red; font-size: 12px; line-height: 2">Benvenuto!</p>

// REACT/JSX
<p style={{color: 'red', fontSize: '12px', lineHeight: 2}}>Benvenuto!</p>
```

Se il valore della proprietà è un numero, possiamo passarlo come tale, altrimenti dobbiamo inserirlo in una stringa, come nel caso di `'12px'`.

> **NOTA**
>
> Anche se la scrittura di codice CSS inline può sembrare una pratica obsoleta e dannosa, più avanti vedremo che non è sempre così, e scopriremo perché può essere utile farlo.

Possiamo anche definire un oggetto JavaScript con le proprietà dello stile e poi passarlo all'attributo `style`.

```
const helloStyle = { color: 'red', fontSize: '12px', lineHeight: 2 };

const helloElement = <div>
    <p style={helloStyle}>Benvenuto!</p>
</div>;
```

<img> e self-closing tag

Proviamo ora ad aggiungere un'immagine nella nostra pagina, usando l'elemento `<img src="...">` di HTML.

```
const helloElement = <div>
    <h1>Todo app</h1>
    <p>Benvenuto!</p>
    <img src="https://via.placeholder.com/728x90.png?text=Todo+app">
</div>;
```

Nella finestra del codice JavaScript di Codepen noterete un piccolo punto esclamativo rosso che segnala un problema: viene segnalato l'errore `Unterminated JSX contents`. Nonostante lo standard HTML richieda (`https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img#technical_summary`) che il tag `<img>` non debba avere un tag di chiusura, JSX lo richiede in quanto parente stretto di XML.

```
const helloElement = <div>
    <h1>Todo app</h1>
    <p>Benvenuto!</p>
    <img src="https://via.placeholder.com/728x90.png?text=Todo+app">
</div>;
Unterminated JSX contents (5:6)

ReactDOM
  .createRoot(document.getElementById('root'))
  .render(helloElement);
```

Figura 1.3 L'errore con l'elemento <img> non correttamente chiuso.

Inoltre, visto che l'elemento `<img></img>` è vuoto (non ha altri elementi al suo interno), possiamo scriverlo anche usando la forma abbreviata, chiamata *self closing tag*, `<img src="..." />`. I due modi riportati qui sotto per inserire immagini nel codice JSX sono entrambi validi.

```
const helloElement = <div>
    <h1>Todo app</h1>
    <p>Benvenuto!</p>
    <img src="https://via.placeholder.com/728x90.png?text=Todo+app"/>
    <img src="https://via.placeholder.com/728x90.png?text=Todo+app"></img>
</div>;

ReactDOM
    .createRoot(document.getElementById('root'))
    .render(helloElement);
```

Dagli elementi ai componenti

I componenti React ci permettono di dividere le nostre interfacce in parti indipendenti e riutilizzabili, e rappresentano quindi il passo successivo rispetto ai semplici elementi che abbiamo visto finora.

Per definire un componente è sufficiente creare una normale funzione JavaScript che restituisca elementi React: ogni volta che la funzione viene eseguita, viene generato e restituito l'elemento definito al suo interno. Possiamo quindi usare JSX per definire l'elemento restituito dalla funzione, cioè il `<div>` con all'interno gli altri elementi, che, per ora, sarà la pagina principale della nostra applicazione.

Listato 1.3 Il nostro primo componente.

```
function App() {
    return <div>
        <h1>Todo app</h1>
        <p>Benvenuto!</p>
```

```
    </div>;
};

ReactDOM
    .createRoot(document.getElementById('root'))
    .render(<App />);
```

La funzione `App` è il nostro primo componente React, e come tale possiamo usarlo nel codice JSX usando il tag `<App></App>` o `<App />` (il *self closing tag* che abbiamo visto poco fa).

NOTA

Usare `<App />` al posto di `<App></App>` non è proprio la stessa cosa, ma lo vedremo più avanti.

I nomi dei componenti, e quindi le funzioni che li definiscono, devono sempre iniziare con una lettera maiuscola; questo è necessario per indicare a React che non si tratta di normali elementi HTML (per esempio, `<p>`) ma di componenti.
I componenti possono essere composti, a loro volta, da altri componenti, così da poter realizzare interfacce complesse lavorando in blocchi singoli e separati che possono anche essere riutilizzati.

Listato 1.4 Un'applicazione con più componenti.

```
function AppTitle () {
    return <h1>Todo app</h1>;
}

function AppMessage() {
    return <p>Benvenuto!</p>;
}

function App() {
    return <div>
        <AppTitle />
        <AppMessage />
    </div>;
}

ReactDOM
    .createRoot(document.getElementById('root'))
    .render(<App />);
```

DEFINIZIONE

Un componente è una funzione JavaScript che restituisce elementi React definiti al suo interno o generati attraverso l'utilizzo di altri componenti.

Riepilogo

Questo capitolo ha permesso di toccare con mano alcune delle caratteristiche principali di React. Siamo partiti dalle istruzioni chiave per collegare React alla pagina HTML fino ad arrivare a creare i nostri primi componenti. JSX, con le sue peculiarità, è il principale elemento di novità mostrato in queste pagine. Il rapporto tra il DOM e React è stretto, ma allo stesso React ci isola completamente da esso, permettendoci di ragionare in termini di elementi e componenti, e lasciando a lui la responsabilità di mostrarli nella pagina. Anche se abbiamo creato solo pochi semplici componenti, dovrebbe esserci già chiaro quanto sia potente questo meccanismo, e quali vantaggi ci siano nel costruire le interfacce attraverso di essi.

Nonostante l'ambiente di lavoro online, tutto quello che abbiamo imparato sarà utilizzabile anche nei prossimi capitoli, quando svilupperemo il resto dell'applicazione lavorando in locale (sul PC), così da usare React in modo reale.