



# Fondamenti del Web

## Ingegneria del Software e Fondamenti Web

Corso di Laurea in Ingegneria  
Informatica e dell'Automazione

Anno Accademico 2023/2024

Prof. Antonio Ferrara

# 10 Frontend e React



# Evoluzione dei frontend

Nei primi anni del Web, i **siti web** erano serviti **staticamente** come **pagine HTML**

Le interfacce di frontend utilizzate dagli utenti erano sostanzialmente statiche e l'interazione poteva avvenire generalmente solo tramite form

Quando è nata la prima generazione di applicazioni web, gli utenti potevano iniziare a vedere **contenuti generati dinamicamente lato server**

Soltanamente venivano utilizzati approcci a **template** che permettevano di realizzare delle “viste” mescolando codice HTML con la logica applicativa

Questo modello è migliorato quando lo sviluppo di applicazioni web ha iniziato a seguire il pattern **MVC**, che ha permesso di disaccoppiare la logica applicativa dalla presentazione del contenuto (es. quello che abbiamo realizzato con Node.js e il rendering delle viste)

Successivamente, le web app sono diventate molto più simili ad applicazioni desktop capaci di funzionare **all'interno del browser** mediante JavaScript e che connettono i modelli scaricati da un server con le viste realizzate (e gestite) client-side

Questa nuova generazione di frontend è realizzata mediante framework come React, Vue.js e Angular(v2) e si basa su viste assemblate mediante componenti riusabili piuttosto che semplici pagine

# Cosa è e perché React

React è una **libreria JavaScript** per lo sviluppo dichiarativo di complesse **interfacce utente basate su componenti**

React si occupa principalmente della **gestione dello stato** dei componenti e del **rendering** di tali componenti all'interno del DOM

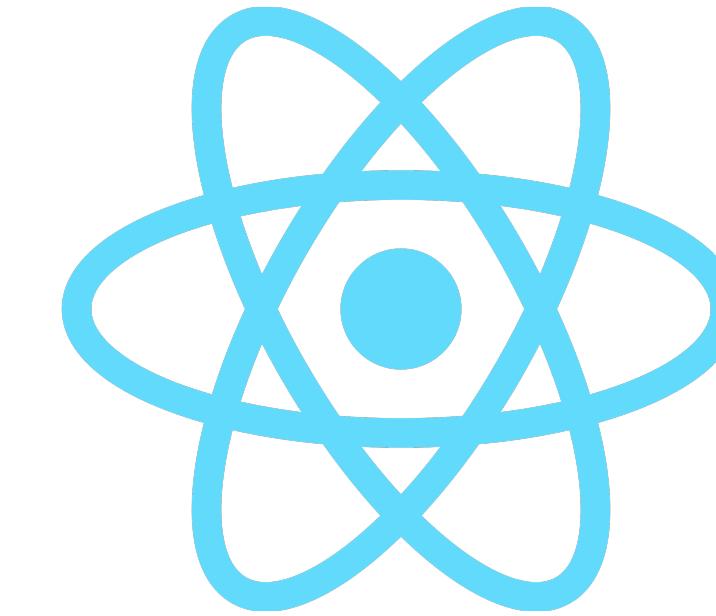
La logica di realizzazione di componenti di React permette l'**isolamento**, la **composizione** ed il **riuso del codice** per lo sviluppo di UI

React permette di conservare lo stato dell'applicazione e di aggiornare in maniera efficiente, ad ogni cambio di stato, solo le parti della UI che dipendono da tali dati

Ciò avviene grazie al concetto di **Virtual DOM**, una rappresentazione interna della UI composta da elementi React che viene volta per volta sincronizzata con il DOM reale mediante un processo detto **rinconciliazione**

L'uso del Virtual DOM permette a React di interpretare in maniera efficiente le differenze fra stati successivi dell'interfaccia e aggiornare, dopo ogni re-rendering, **solo le parti del DOM che necessitano di un cambiamento**

Questo processo rende l'aggiornamento del DOM molto più veloce di vanilla JavaScript



React è **open-source** e mantenuto da **Meta**: ciò significa che migliora ed evolve costantemente

# La prima app in React

```
<html>
  <head>
    <title>React App</title>
    <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>
    <script>
      const greeting = React.createElement("h1", { id: "recipe-0" }, "Hello World!");
      ReactDOM.render(greeting, document.getElementById("root"));
    </script>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

Il Virtual DOM di React è fatto di elementi React

**Elementi del DOM HTML ed elementi React non sono la stessa cosa:** gli elementi React sono una descrizione di come gli elementi del DOM HTML dovrebbero apparire

Il ReactDOM contiene gli strumenti necessari per renderizzare gli elementi React nel browser

Al posto di "Hello World!"  
Possiamo anche inserire diversi elementi React  
Il secondo argomento prende il nome di **props**, e le sue proprietà possono essere utilizzate nel contenuto dell'elemento

# La prima app in React

React permette di **semplificare** notevolmente la classica creazione di **elementi React** mediante lo pseudolinguaggio **JSX** (che viene convertito in chiamate a funzioni JavaScript), molto simile a quella di HTML

```
const element = React.createElement('h1', null, `Hello World!`) // senza JSX
const element = <h1>Hello World!</h1> // con JSX
```

Mediante JSX è inoltre possibile **incorporare espressioni JavaScript** all'interno di {}, come nell'esempio seguente

```
const name = 'Mario Rossi';
const element = <h1>Hello, {name}</h1>;
```

JSX non può specificare due **elementi adiacenti** se questi non sono racchiusi in un **unico elemento**

```
ReactDOM.render(<h1>Hello World!</h1><p>Benvenuti!</p>, document.getElementById("root"));
// sbagliato
```

```
ReactDOM.render(<div><h1>Hello World!</h1><p>Benvenuti!</p></div>,
document.getElementById("root")); // corretto
```

# La prima app in React

```
<html>
  <head>
    <title>React App</title>
    <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>
    <script src="https://unpkg.com/babel-standalone/babel.min.js"></script>
    <script type="text/babel">
      ReactDOM.render(<h1>Hello World!</h1>, document.getElementById("root"));
    </script>
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

Il codice JSX viene “**compilato**” mediante il **preprocessore Babel** e trasformato in **elementi React**

# La prima app in React

Il secondo argomento di `ReactDOM.render()` specifica il nodo del DOM all'interno del quale vogliamo renderizzare un elemento React

Il React DOM si occupa di tenere aggiornato il DOM del browser affinché sia **consistente** con gli elementi React

```
function tick() {  
  const element = (  
    <div>  
      <h1>Hello, world!</h1>  
      <h2>It is {new Date().toLocaleTimeString()}.</h2>  
    </div>  
  );  
  ReactDOM.render(element, document.getElementById('root'));  
}  
setInterval(tick, 1000);
```

Gli elementi React sono **oggetti semplici** (non derivati da altri prototipi, ma direttamente da Object) e quindi veloci da creare

L'elemento da renderizzare ogni secondo rappresenta l'intero albero della UI, ma il React DOM lo confronta volta per volta con il precedente e aggiorna solo i nodi necessari nel DOM reale (in questo caso, il nodo testo in **h2**)

# Componenti in React

Il primo argomento di `ReactDOM.render()` ci ha permesso di inserire codice JSX che rappresenta il contenuto che vogliamo renderizzare

Chiaramente, aumentando la complessità della vista potremmo ottenere centinaia di elementi all'interno di questo argomento, rendendo impossibile la manutenzione del codice

Uno dei motivi per cui React è popolare è la possibilità di creare **componenti funzionali** realizzabili, appunto, mediante **funzioni che ritornano un elemento React**

```
ReactDOM.render(  
  <ul>  
    <li>Elemento 1</li>  
    <li>Elemento 2</li>  
  </ul>,  
  document.getElementById("root")  
)
```

I nomi dei componenti  
seguono la notazione  
PascalCase

```
function MyApp() {  
  return (  
    <ul>  
      <li>Elemento 1</li>  
      <li>Elemento 2</li>  
    </ul>  
  );  
}
```

```
ReactDOM.render(<MyApp />,  
  document.getElementById("root"));
```

Successivamente,  
nell'organizzazione di  
un'app in React, andremo  
ad inserire ogni  
componente funzionale in  
un file separato

# Esercizi

## Esercizio 10.1

Creare e renderizzare un componente funzionale **MyInfo** che rappresenti l'interfaccia di una card biografica con i seguenti elementi:

- Un elemento **h3** con il proprio nome
- Un paragrafo con una breve bio
- Una lista non ordinata (es. skill principali)

# Transpiling e toolchain

React è eseguibile direttamente dentro il browser grazie all'importazione di **Babel**, che effettua il **transpiling** di JSX

Nonostante ciò sia possibile, è preferibile che il transpiling di un'applicazione React in produzione venga effettuato **offline** per diverse ragioni:

- Evitare che gli utenti scarichino ogni volta lo script di Babel, che è nell'ordine di qualche megabyte
- Il processo di compilazione richiede del tempo
- Il risultato della compilazione non viene cachato dai browser

Il transpiling offline è possibile grazie a Node.js (utilizziamo Node come interprete di JavaScript che ci consente di potenziare il nostro ambiente di sviluppo e non per realizzare un server!)

```
npm init -y
npm install babel-cli@6 babel-preset-react-app@3
```

```
npx babel --watch src --out-dir . --presets react-app/prod
```

# Transpiling e toolchain

Esistono una serie di **toolchain** che aiutano a scalare su molti file, utilizzare librerie di terze parti, individuare errori, visualizzare le modifiche in tempo reale, ottimizzare l'output per la produzione

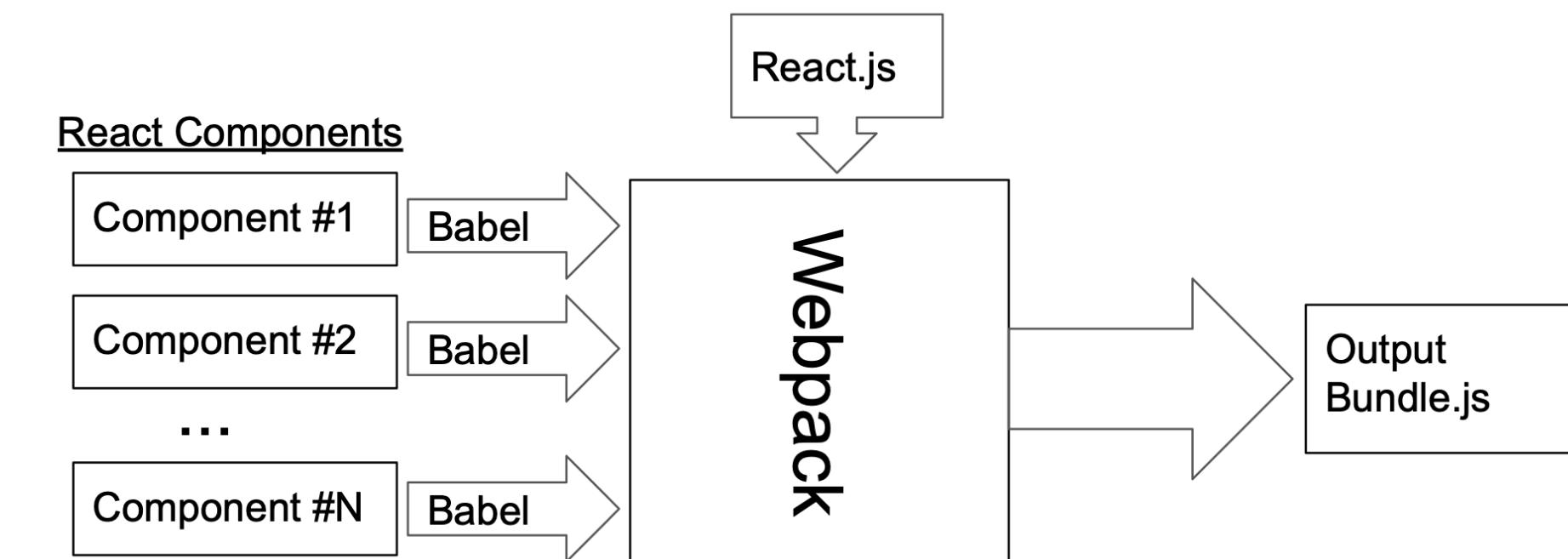
Ad esempio, Create React App supporta lo sviluppo di Single Page Application in React e risulta utile nella fase di apprendimento di React

```
npx create-react-app mia-app  
cd mia-app  
npm start
```

Create React App al suo interno utilizza **Babel** per il transpiling e **webpack** per creare un pacchetto di asset direttamente utilizzabile nel browser a partire da un codice sorgente strutturato in file, moduli e dipendenze

Il deployment di una applicazione creata con Create React App prevede realizzare una build che possa essere servita staticamente da un server

```
npm run build
```



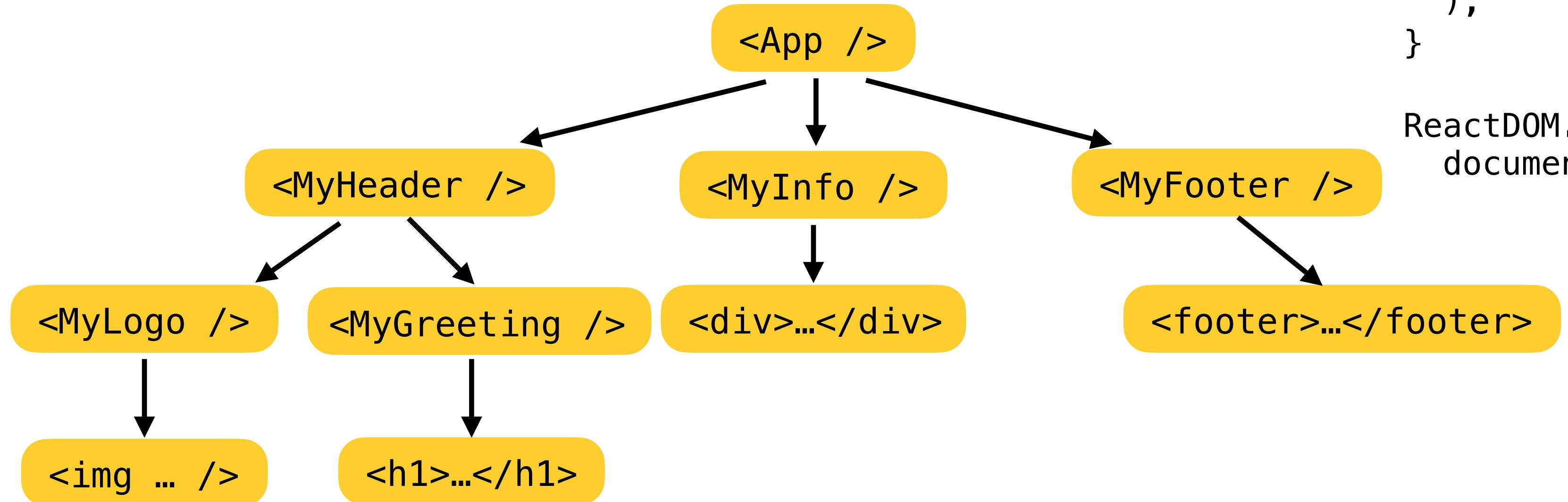
Su Replit viene utilizzato Vite anziché webpack, che utilizza Esbuild (scritto in Go) per fare pre-bundling

# Composizione di componenti

Negli esempi precedenti, abbiamo creato componenti semplici che restituivano un semplice elemento React

I componenti, però, possono essere fra loro **composti**, facendo in modo che un componente faccia riferimento ad altri componenti nel suo output

In questo modo, React permette di **separare le responsabilità** dei diversi componenti



```
function App() {  
  return (  
    <div>  
      <MyHeader />  
      <MyInfo />  
      <MyFooter />  
    </div>  
  );  
}
```

```
ReactDOM.render(<App />,  
  document.getElementById("root"));
```

# Esercizi

## Esercizio 10.2

Creare la UI della slide precedente utilizzando una toolchain come Create React App (ad esempio con WebStorm) o quella offerta da Replit e organizzando i componenti in moduli separati

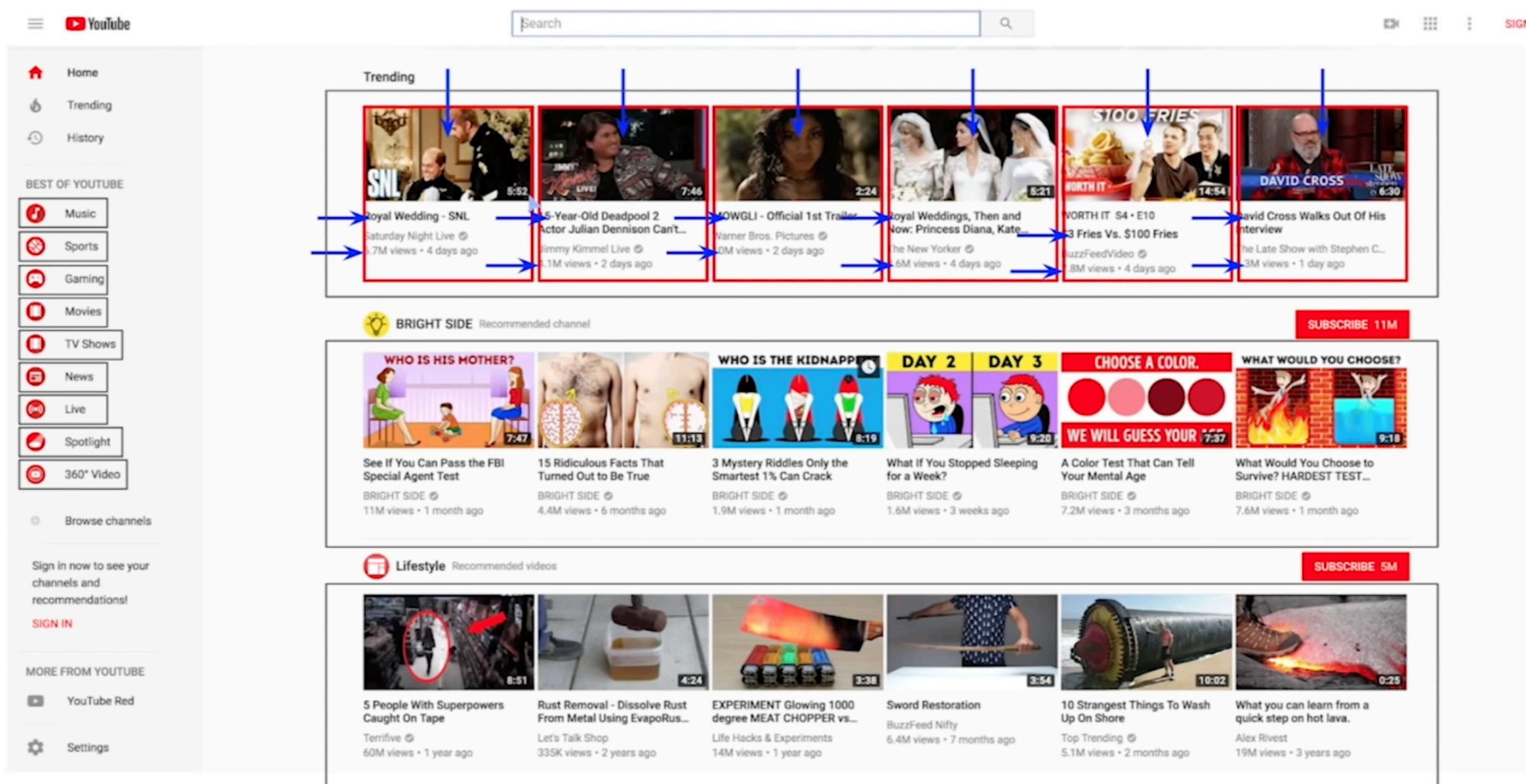
In particolare, riutilizzare il componente **MyInfo** realizzato nell'Esercizio 10.1

Inoltre, il componente **MyGreeting** dovrà salutare con “Buongiorno” o “Buonasera” in base all’ora del giorno

Per far ciò si consideri di creare un oggetto della classe **Date** e utilizzare su di esso il metodo **getHours()**

Su un elemento React utilizza l’attributo JSX **className** per specificare il nome della classe di stile!  
Per specificare stili inline possiamo utilizzare l’attributo **style** a cui possiamo passare un oggetto contenente le proprietà CSS e i relativi valori, ad esempio  
**style={{display: none, backgroundColor: '#FF2D00'}}**

# Componenti e proprietà



La realizzazione di **componenti** è di estrema utilità nei framework frontend in quanto permettono di realizzare **parti riusabili** dell'interfaccia

Nell'esempio a sinistra, possiamo individuare diversi componenti, come i tile dei video (con le info), le barre contenenti set di video, o i pulsanti con i link ai generi più popolari

Chiaramente, per far ciò, è necessario che ciascun componente possa essere dinamicamente “istanziato” con delle **proprietà** diverse

# Componenti e proprietà

In React è possibile rendere “diverso” ciascuna istanza di un componente mediante le sue **proprietà**: quando viene inserito nella pagina un componente, infatti, ad esso possono essere passate delle proprietà mediante l’oggetto **props**, che può essere ricevuto come argomento dalla funzione che renderizza il componente

In **JSX**, tali proprietà vengono passate al nuovo componente come fossero attributi HTML

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

```
function InfoCard(props) {  
  return (  
    <div className="info-card">  
      <img src={props.imgUrl} />  
      <h3>{props.name}</h3>  
      <p>Phone: {props.phone}</p>  
      <p>E-mail: {props.email}</p>  
    </div>  
  );  
}
```

```
function App() {  
  return (  
    <div className="cards">  
      <InfoCard name="Mr. Whiskerson"  
        imgUrl="..." phone="..." email="..." />  
      <InfoCard name="Fluffykins"  
        imgUrl="..." phone="..." email="..." />  
      <InfoCard name="Mr. Whiskerson"  
        imgUrl="..." phone="..." email="..." />  
    </div>  
  );  
}
```



**Mr. Whiskerson**

Phone: (212) 555-1234  
Email: mr.whiskaz@catnap.meow



**Fluffykins**

Phone: (212) 555-2345  
Email: fluff@me.com



# Liste e chiavi

React ci permette di lavorare liberamente con vanilla JavaScript, anche per gestire la logica di rendering di componenti React

Ad esempio, se le informazioni relative alle info card vengono scaricate da un server mediante API oppure sono presenti all'interno di un file JSON, possiamo

**mappare l'array** di informazioni **con componenti di React**

Successivamente, React ci permette di **renderizzare** direttamente l'array di componenti (sempre racchiuso all'interno di un altro elemento)

```
import infoData from './infoData';
```

```
function App() {
  const cards = infoData.map(data => <InfoCard data={data} />);

  return (
    <div>
      {cards}
    </div>
  );
}
```

È buona pratica associare ad ogni elemento/componente della lista un attributo **key** univoco fra i diversi sibling. Questo aiuta React a fare più velocemente i **confronti tra due alberi** quando deve aggiornare la UI.

Nell'esempio seguente, nel primo tipo di confronto dovremmo fare una mutazione per ogni elemento; nel secondo tipo di confronto React capisce che è stato solo aggiunto un nuovo elemento in cima

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

```
<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

---

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

```
<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

# Esercizi

## Esercizio 10.3

Realizzare un'applicazione capace di mostrare le ricette presenti in recipes.json mediante una lista di componenti

### **Recipe**

Ogni componente possederà il contenuto della ricetta, eventualmente suddiviso in diversi componenti (es.

### **IngredientsList e Instructions**

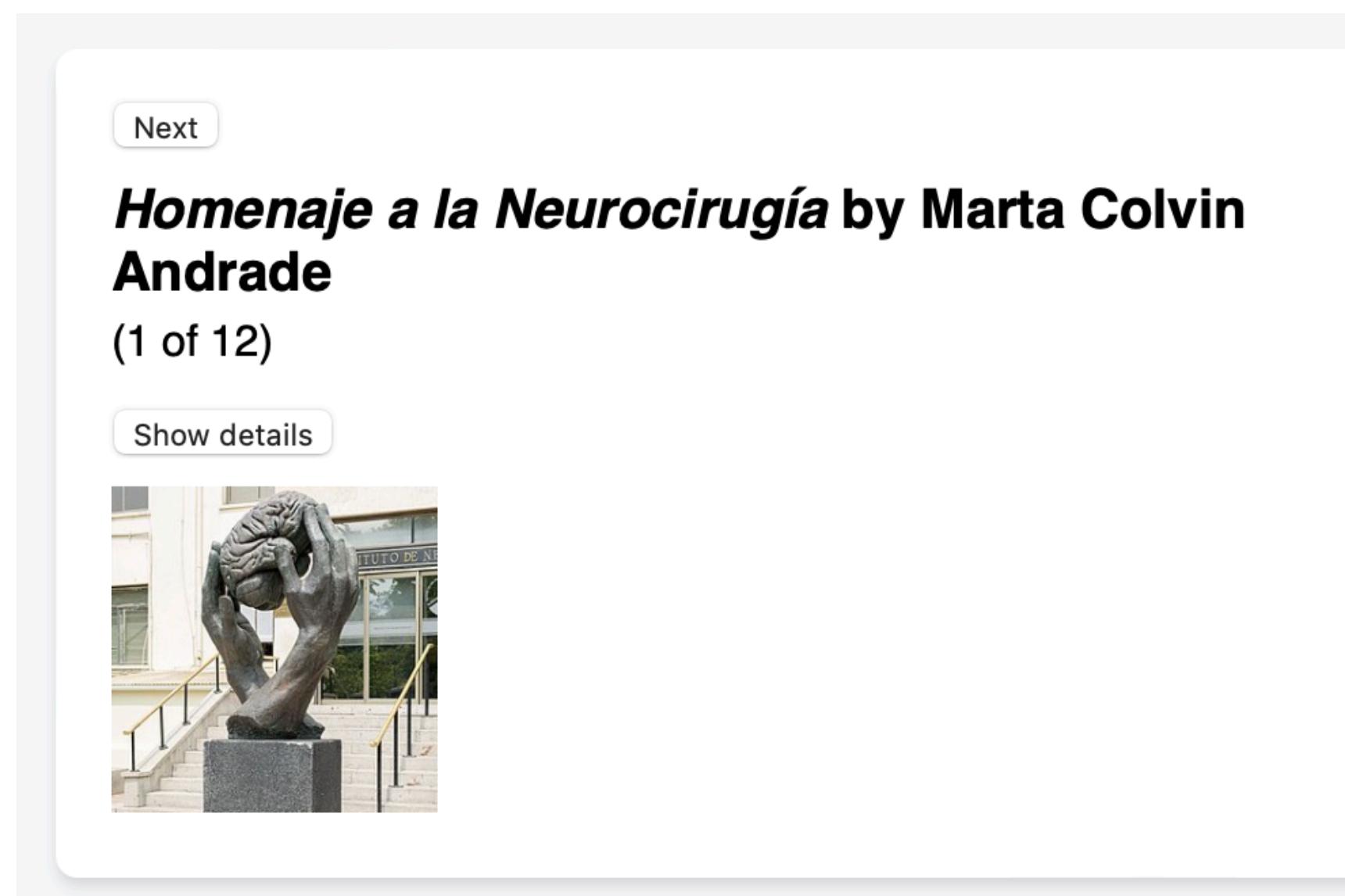
# Componenti e stato

I dati sono ciò che danno vita ai componenti (si pensi all'array di ricette nell'esercizio precedente)

Abbiamo visto come in React i dati possono fluire nella gerarchia di componenti come proprietà

Tuttavia, le proprietà sono solo una faccia della medaglia, in quanto esse sono “**immutabili**”, ovvero i componenti devono comportarsi come funzioni pure rispetto alle loro proprietà

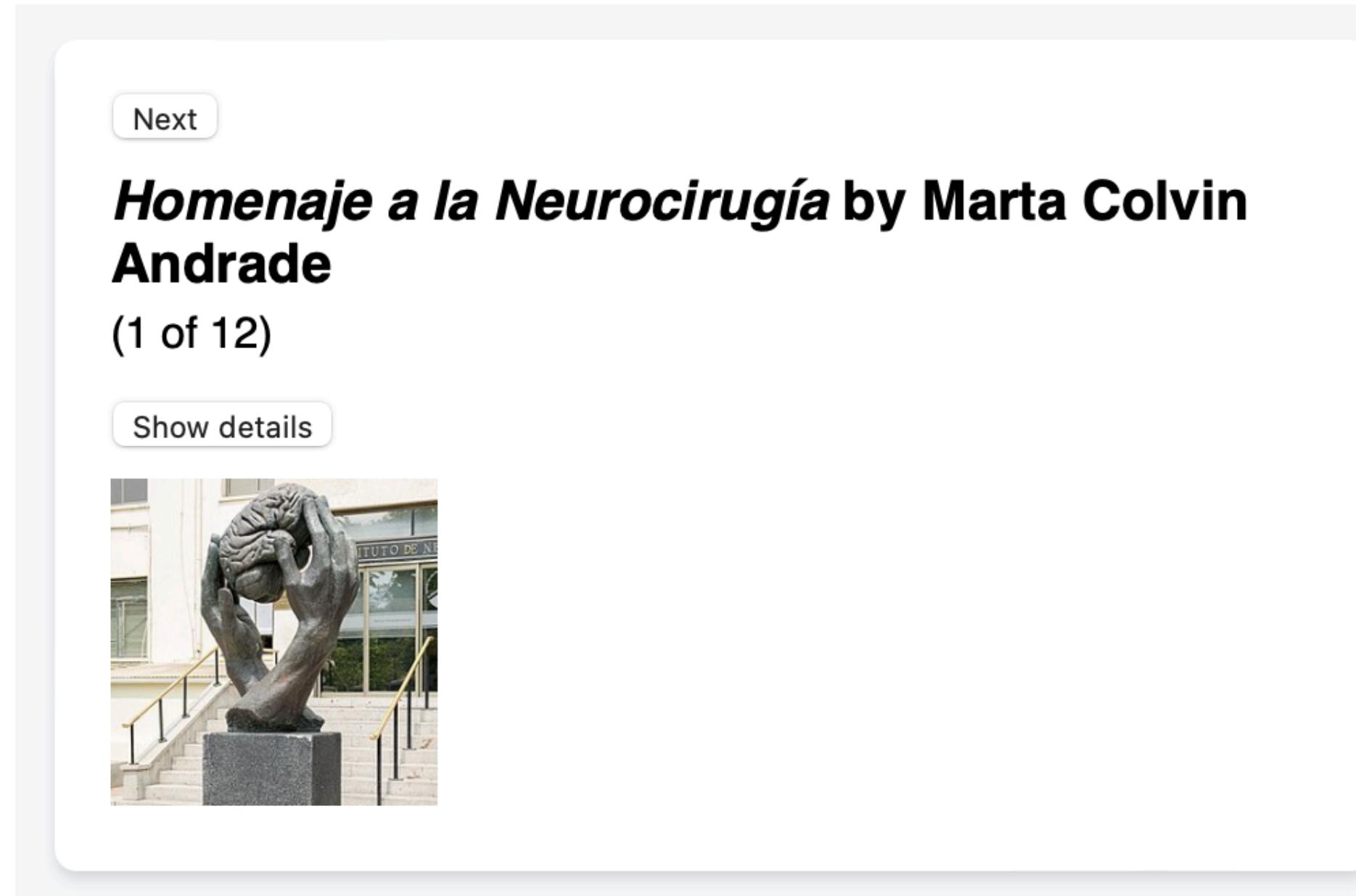
L'altra faccia della medaglia, allora, è lo **stato** di un componente, che ha la possibilità di **cambiare**



Si pensi all'esempio a sinistra

Come potremmo tenere conto della foto attuale da visualizzare (1 of 12, 2 of 12, etc.) o come potremmo tenere conto del fatto che l'utente voglia o meno visualizzare i messaggi?

# Componenti e stato



Possiamo pensare di dichiarare delle variabili locali alla funzione, ma questo approccio non può funzionare per almeno due motivi

Quali?

```
function Gallery() {
  let index = 0;
  let showDetails = false;

  const handleNext = () => { index = index + 1 };
  const handleDetails = () => { showDetails = !showDetails }

  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleNext}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
        by {sculpture.artist}
      </h2>
      <h3>
        ({index + 1} of {sculptureList.length})
      </h3>
      <button onClick={handleDetails}>
        {showDetails ? 'Hide details' : 'Show details'}
      </button>
      <div><img src={sculpture.url} /></div>
      {showDetails && <p>{sculpture.description}</p>}
    </>
  );
}
```

# Componenti e stato

La funzione restituisce due oggetti: uno stato e una funzione dello stato, definita in questo caso `setCount --> set + nome dello stato.`

Abbiamo quindi bisogno di una **variabile di stato** (e di un suo **setter**) che:

- **Conservi i dati** fra due rendering
- Ogni volta che viene **settata a un nuovo valore**, viene triggerato il **re-rendering** del componente (e dei suoi figli)

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

export default function Counter() {
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }

  return (
    <button onClick={handleClick}>
      You pressed me {count} times
    </button>
  );
}
```

Nelle parentesi tonde va il valore di assegnazione dello stato (`count + 1` si usa per incrementare). Lo stato non può essere assegnato, si cambia solo mediante il suo setter (`count = count + 1 --> ERRORE`)

`alert('premuto')` è una chiamata alla funzione. invece bisogna trasformare la funzione in una callback.  
es. `onbuttonclick = () => alert('premuto')`

La funzione `useState` è un **React Hook** che permette di aggiungere una variabile di stato a un componente

Il primo valore di ritorno è lo **stato corrente**, che nel primo render corrisponde al valore iniziale passato alla funzione

Il secondo valore di ritorno è una **funzione setter** che permette di cambiare lo stato corrente

Gli **hook** sono speciali funzioni che permettono di “agganciarsi” a particolari feature di React durante il rendering

**Non** usare gli hook in statement condizionali o loop, ma solo nel top-level del componente

# Esercizi

## Esercizio 10.4

Correggere il componente **Gallery** affinché utilizzi correttamente le due variabili di stato necessarie per il suo funzionamento

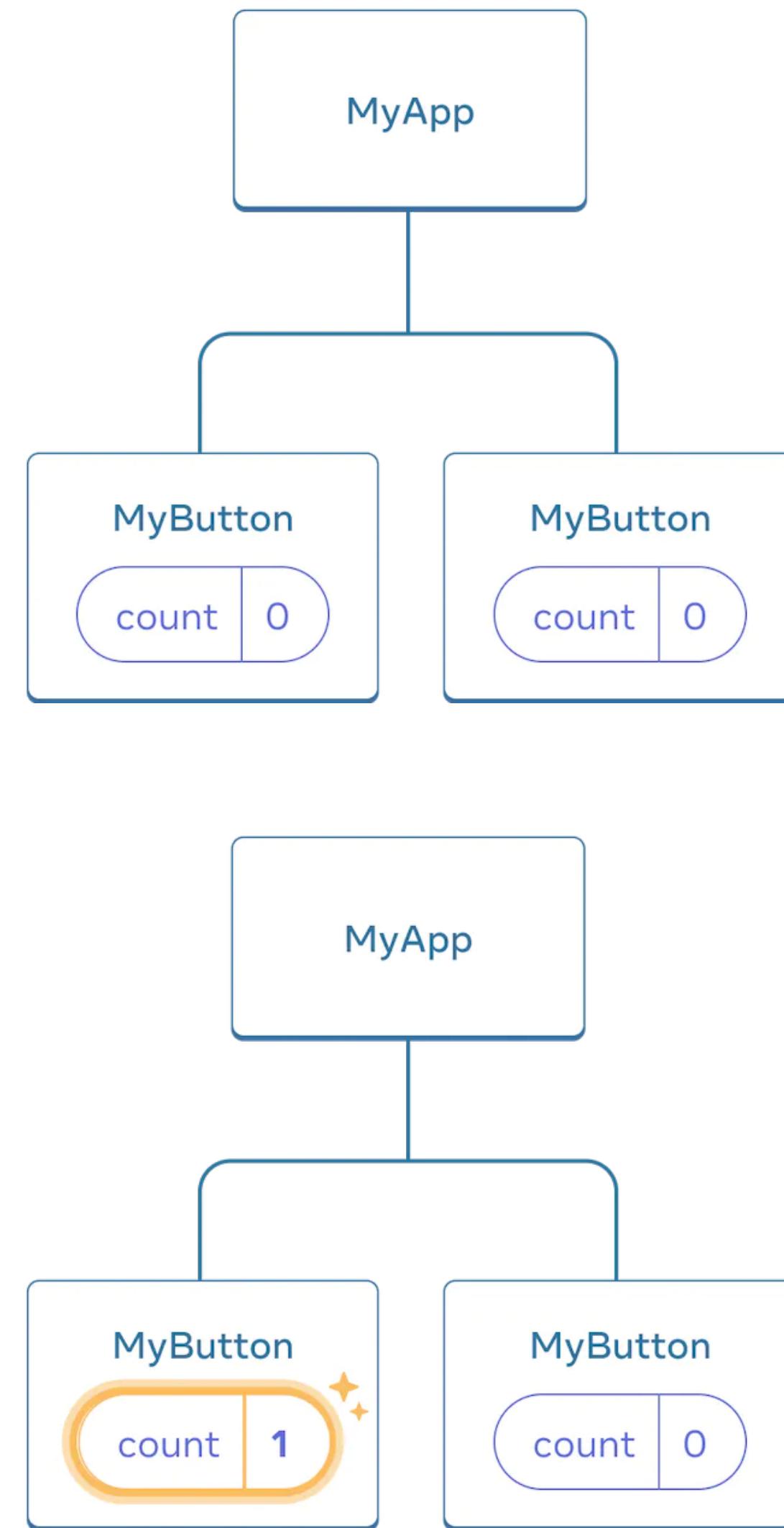
# Componenti e stato

Si consideri il seguente esempio, in cui i due componenti **MyButton** possiedono stati indipendenti

```
export default function MyApp() {
  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton />
      <MyButton />
    </div>
  );
}

function MyButton() {
  const [count, setCount] = useState(0);

  return (
    <button onClick={() => setCount(count + 1)}>
      Clicked {count} times
    </button>
  );
}
```



# Componenti e stato

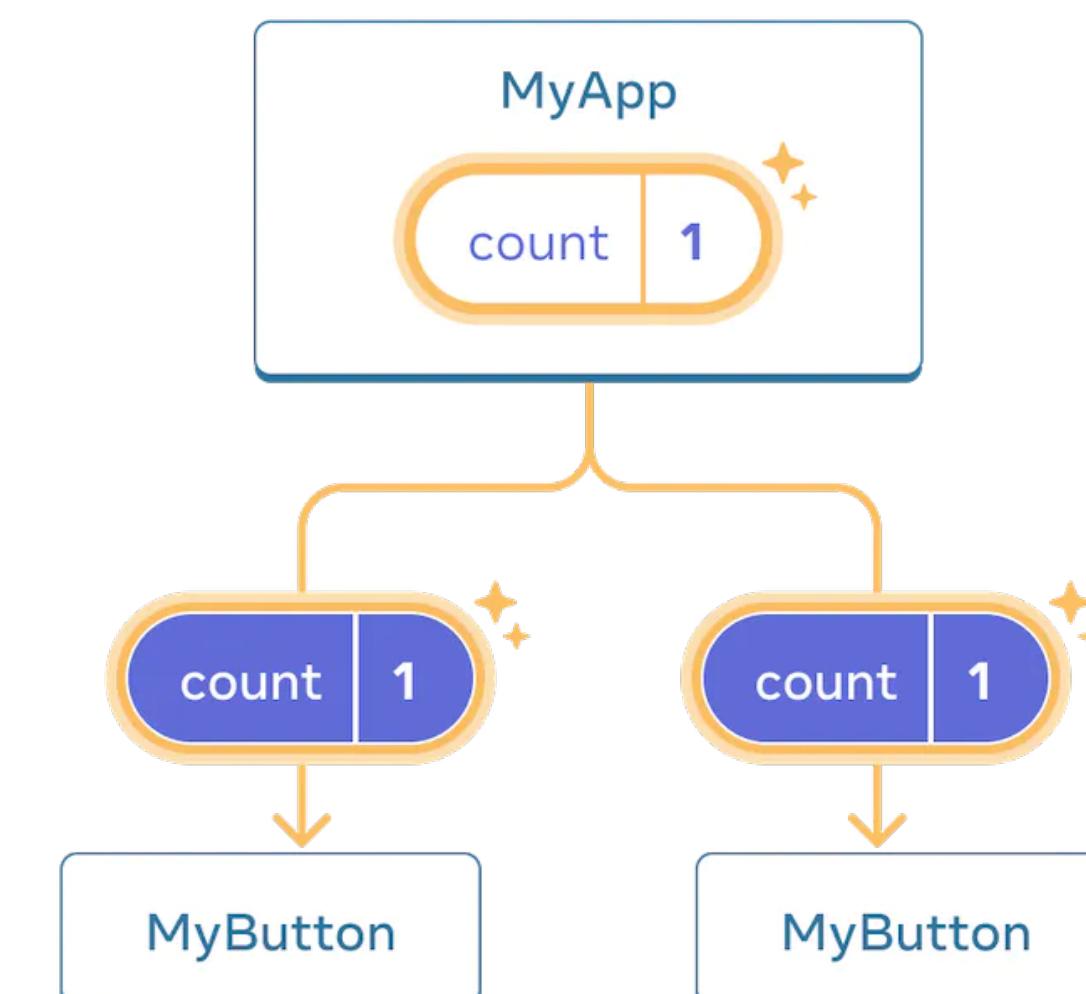
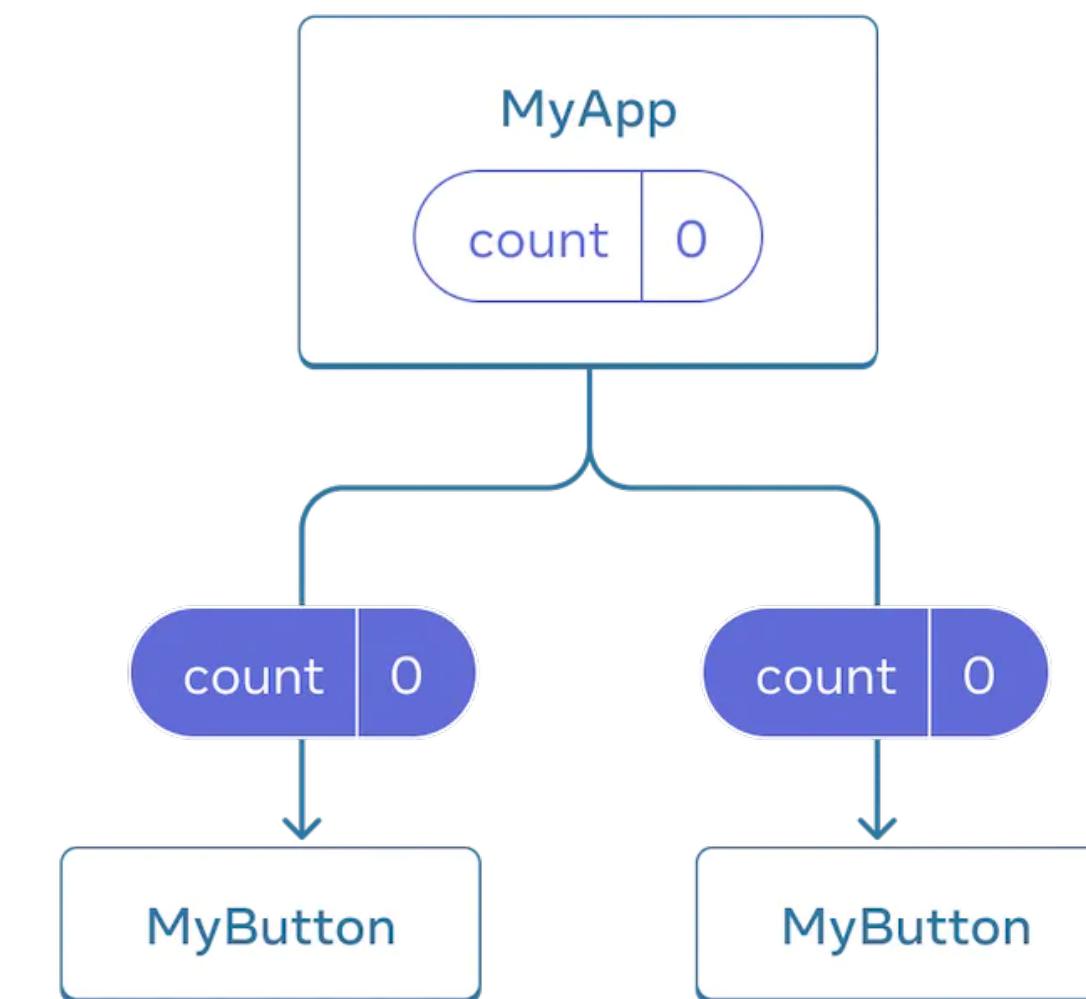
Come potremmo, invece, fare in modo che i due pulsanti si aggiornino “insieme”?

In generale, come facciamo a far fluire uno stato unico verso componenti a livello più basso della gerarchia?

Spostiamo lo stato ad un livello più alto della gerarchia e lo passiamo come **proprietà**. Ogni volta che lo stato in alto cambierà, i componenti che dipendono da questo verranno renderizzati nuovamente.

Come faranno, quindi, i componenti più interni ad agire sulle funzioni che cambiano lo stato di un componente più in alto?

Semplicemente, anche la funzione che gestisce l’evento dovrà essere passata come proprietà ai componenti più interni.



Lo stato sopravvive al re-rendering di un componente e se il componente viene reindirizzato lo stato non cambia.

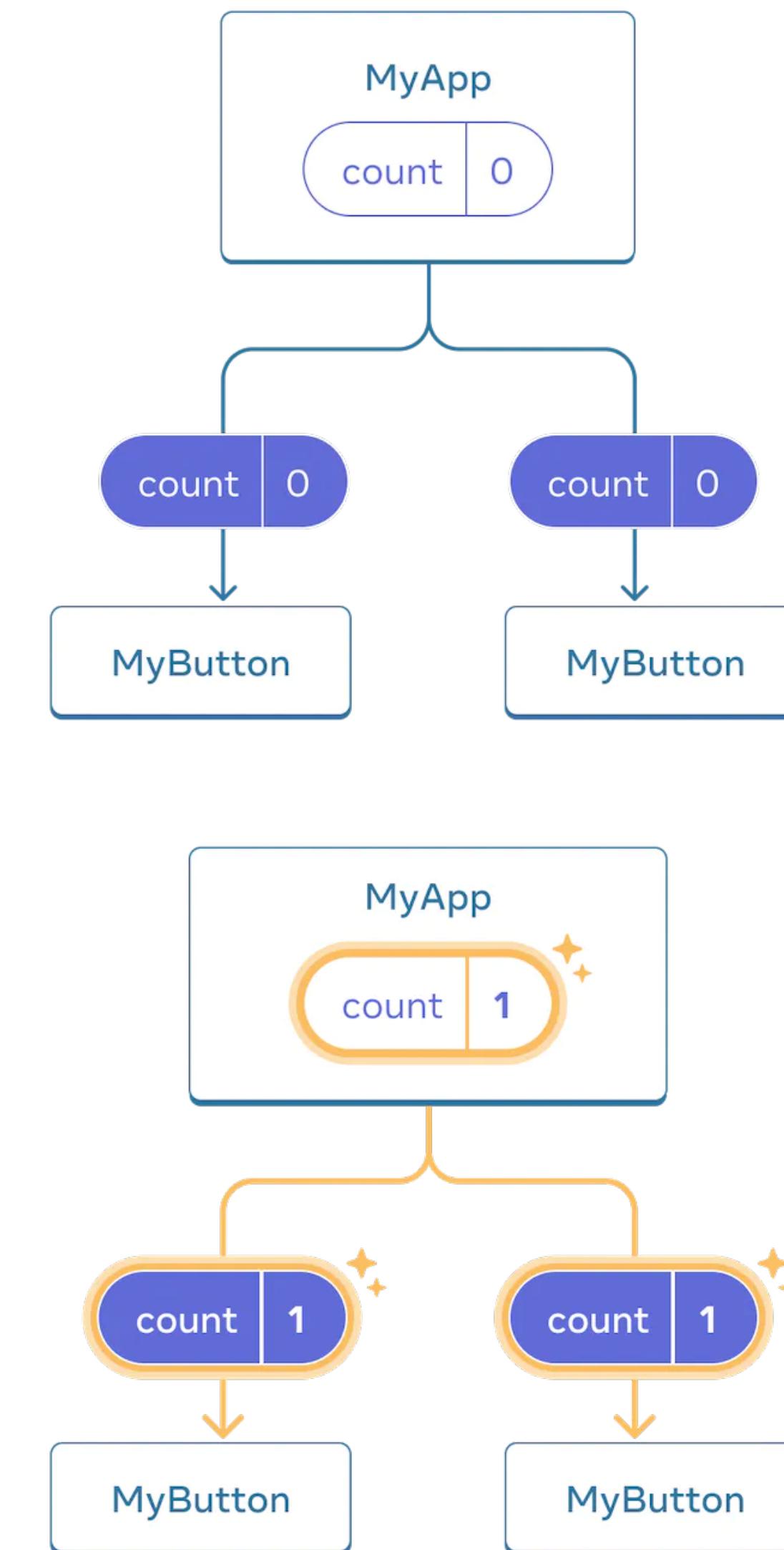
# Componenti e stato

```
export default function MyApp() {
  const [count, setCount] = useState(0);
  const handleClick = () => setCount(count + 1);

  return (
    <div>
      <h1>Counters that update separately</h1>
      <MyButton count={count} onClick={handleClick} />
      <MyButton count={count} onClick={handleClick} />
    </div>
  );
}

function MyButton({ count, onClick }) {

  return (
    <button onClick={onClick}>
      Clicked {count} times
    </button>
  );
}
```



# Componenti e stato

Di seguito sono mostrati due codici che permettono di comprendere alcune particolarità su come React gestisce lo stato e il rendering

```
export default function Counter() {
  const [number, setNumber] = useState(0);

  return (
    <>
      <h1>{number}</h1>
      <button onClick={() => {
        setNumber(number + 1);
        setNumber(number + 1);
        setNumber(number + 1);
      }}>+3</button>
    </>
  )
}
```

Nota come `number` viene incrementato una volta per click  
In questo render `number` è **0** (tutte e tre le volte) e viene  
settato a **1** per il render successivo

```
export default function Form() {
  const [to, setTo] = useState('Alice');
  const [message, setMessage] = useState('Hello');

  function handleSubmit(e) {
    e.preventDefault();
    setTimeout(() => {
      alert(`You said ${message} to ${to}`);
    }, 5000);
  }

  return (
    <form onSubmit={handleSubmit}>
      <label>
        To:' '
        <select
          value={to}
          onChange={e => setTo(e.target.value)}>
          <option value="Alice">Alice</option>
          <option value="Bob">Bob</option>
        </select>
      </label>
      <textarea
        placeholder="Message"
        value={message}
        onChange={e => setMessage(e.target.value)}>
      </>
      <button type="submit">Send</button>
    </form>
  );
}
```

Il valore di una variabile di stato non cambia mai durante il rendering

[...arr, 6] = spread operator, serve per fare l'unpacking di tutti i valori che stanno nell'array = spaccettare i valori di un array in un array. [1...arr]  
{...obj} : copia le proprietà di un oggetto

Lo stato si può cambiare mediante il suo setter al quale non si può passare una versione modificata dell'array, ma un nuovo oggetto uguale a quello di partenza.  
students.push('d') --> NO

# Componenti e stato

```
setpoints([...points, {x:10, y:8}])  
function removeygreater (val) {setpoints(points.filter(p => p.y<=val))}
```

```
const [studs, setstuds] = useState([ { _id :1, name : 'Att', matr: 54, _id :2, name:'andu', matr: 49}])  
function handlechangename(id, name)  
setstuds(studs.map(s=>s._id==id? {...s, name: newName}:s))
```

Facciamo attenzione a stati che non contengono variabili semplici, ma **array** oppure **oggetti**

```
const [position, setPosition] = useState({ x: 0, y: 0 });
```

non dovrebbe essere modificato con **position.x = 10**, ma sempre utilizzando il setter apposito affinché venga triggerato un re-rendering

In altre parole, anche gli oggetti devono essere trattati come dati immutabili

Per far ciò, è necessario **creare un nuovo oggetto e passarlo alla funzione setter**

```
setPosition({ x: 10, y: 0 });    setpoint([...points, {x:8, y:8}]).  
                                setpoint(points.map(p =>{...p, y:p.y +10})
```

```
const [points, setpoints] = useState([{name: 'p1', coo:  
{x:10, y:20}, active = true}, {name: 'p2', coo : {x:40, y:50},  
active = false}])  
function handleactivepoints(1...x, y)  
setpoints([...points, {name.points.length+1} coo: {x:x, y:y}  
active = false}])
```

Se volessimo lasciare invariate tutte le proprietà dell'oggetto di partenza eccetto che alcune potremmo usare lo spread operator

```
setPosition({ ...position, x: 10 });
```

Allo stesso modo, gli array possono essere ritornati mediante lo spread operator, oppure ottenuti mediante **filter**, **map** o **slice**

# Rendering condizionale

Abbiamo già incontrato la possibilità di effettuare **rendering condizionale** all'interno dell'Esercizio 10.2

Si tratta, in generale, di una tecnica estremamente utilizzata in React, specialmente quando è coinvolto lo stato di un componente

```
function MyApp() {  
  [isLoggedIn, setIsLoggedIn] = useState(false);  
  
  return (  
    <div>  
      { isLoggedIn ? <AdminPanel /> : <LoginForm /> }  
    </div>  
  )  
}  
  
function MyApp() {  
  [isLoggedIn, setIsLoggedIn] = useState(false);  
  
  return (  
    <div>  
      {isLoggedIn && <AdminPanel />}  
    </div>  
  )  
}
```

OGNI PUNTO DEVE ESSERE TRASFORMATO. ESEMPIO:  
const[points, setPoints] = useState([{ esempio di prima }])  
function handlechangex(name, x )  
{setPoints(points.map(p=>p.name==name? :p)) {...p, coo:  
{...p.coo, x:x}}}

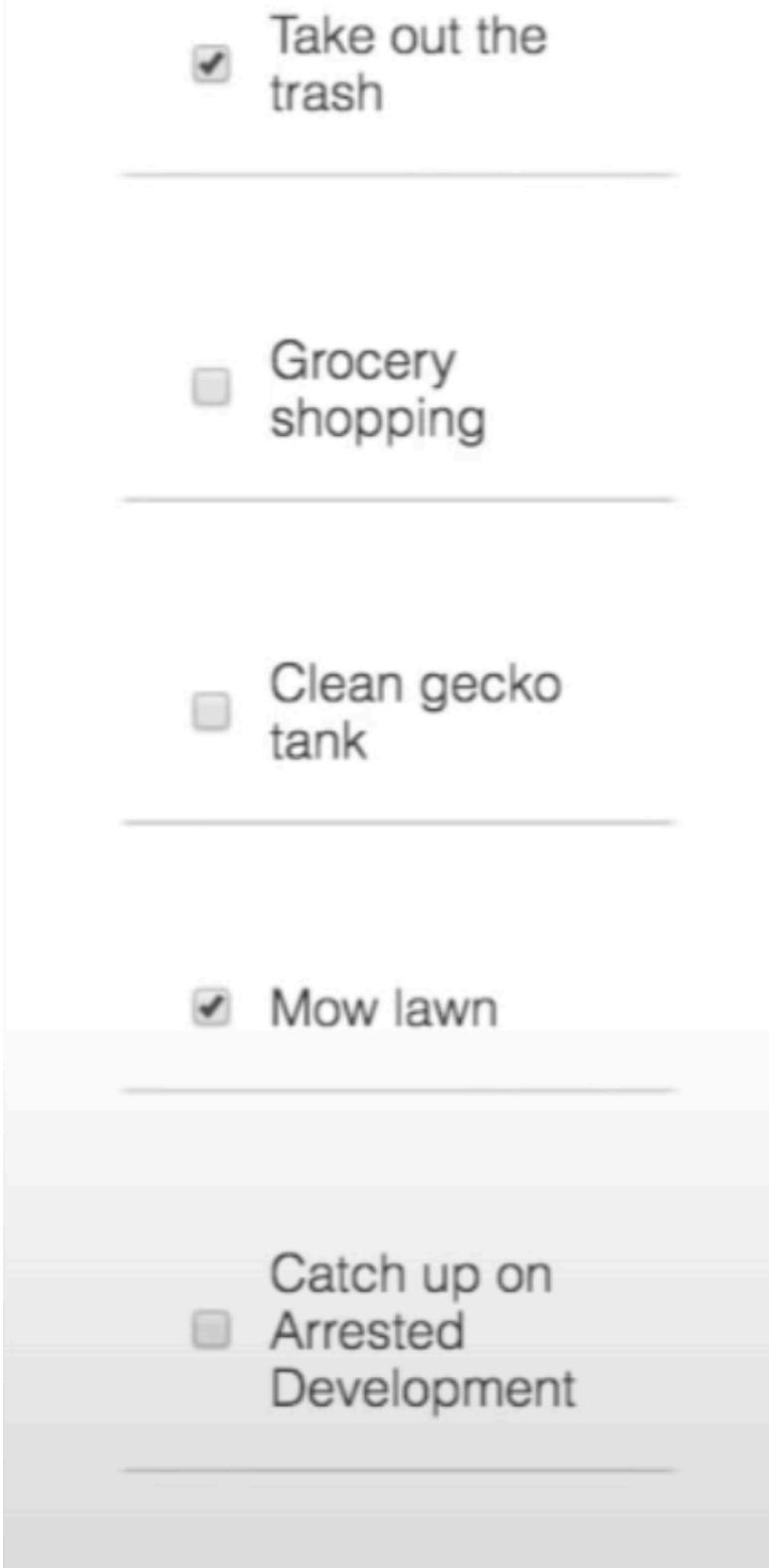
# Esercizi

## Esercizio 10.5

Realizzare un componente **ToDoList** che renderizzi all'interno un componente **ToDoItem** (ad esempio un div) per ogni elemento contenuto in un array presente nel suo stato

In particolare, il componente **ToDoItem** dovrà mostrare una checkbox e un testo relativo all'item

Non dimenticare la chiave!



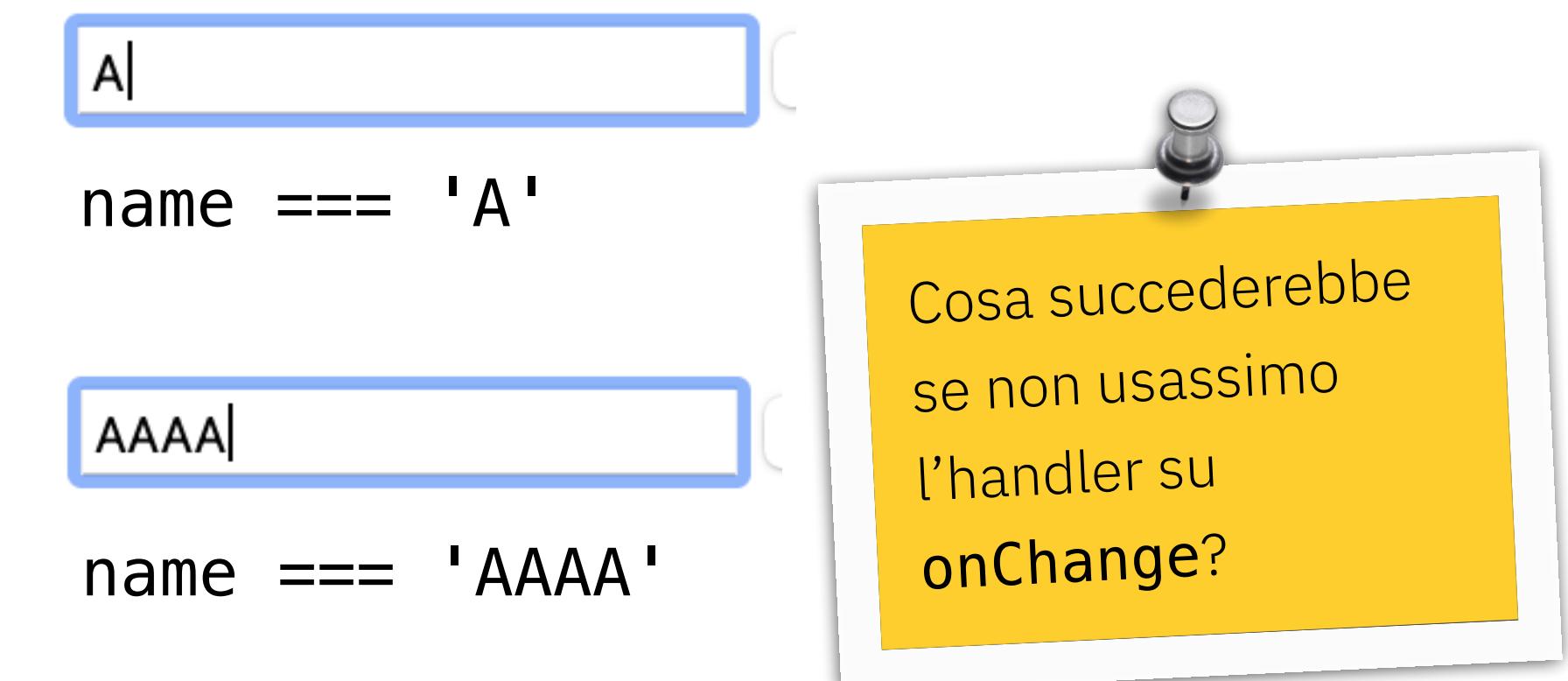
# Form in React

L'idea con cui viene realizzato un form in React è che il **valore** dei suoi elementi venga mantenuto all'interno dello **stato** del componente che lo realizza – ciò permette di rispettare il cosiddetto principio di **Single Source Of Truth** (unica fonte attendibile)

Lo stato del componente che realizza un form, pertanto, verrà aggiornato **ad ogni modifica** dei valori del form: dobbiamo, cioè, guardare tutte le modifiche che l'utente effettua

A sua volta, il form verrà renderizzato riflettendo lo stato attuale

```
function InputBox() {  
  const [name, setName] = useState('');  
  
  return (  
    <>  
      <input  
        value={name}  
        onChange={e => setName(e.target.value)}  
      />  
    </>  
  );  
}
```



# Form in React

Per una questione di consistenza e di praticità, alcuni elementi HTML sono riscritti in maniera diversa, ad esempio:

```
<textarea>Valore</textarea> // in HTML  
<textarea value="Valore" /> // in React
```

L'uso di **checkbox** presenta il problema dell'uso dell'attributo **checked** anziché dell'attributo **value**

Tale differenza con gli altri elementi dovrà essere appositamente gestita nell'**handleChange**

Per i **radio button**, invece, possiamo continuare ad utilizzare lo stesso handler, ricordandoci che la proprietà **checked** proverrà dal valore dello stato

```
const [myForm, setMyForm] = useState({firstName: '', isFriendly: false, gender: 'male'})  
  
function handleChange(event) {  
  event.target.type === "checkbox" ?  
    setMyForm({...myForm, [event.target.name]: event.target.checked}) :  
    setMyForm({...myForm, [event.target.name]: event.target.value})  
}  
  
return ( <>  
  <input type="text" name="firstName" value={myForm.firstName} onChange={handleChange} />  
  <input type="checkbox" name="isFriendly" checked={myForm.isFriendly} onChange={handleChange} />  
  <input type="radio" name="gender" value="male"  
        checked={myForm.gender === "male"} onChange={handleChange} />  
  <input type="radio" name="gender" value="female"  
        checked={myForm.gender === "female"} onChange={handleChange} /> </> )
```

# Esercizi

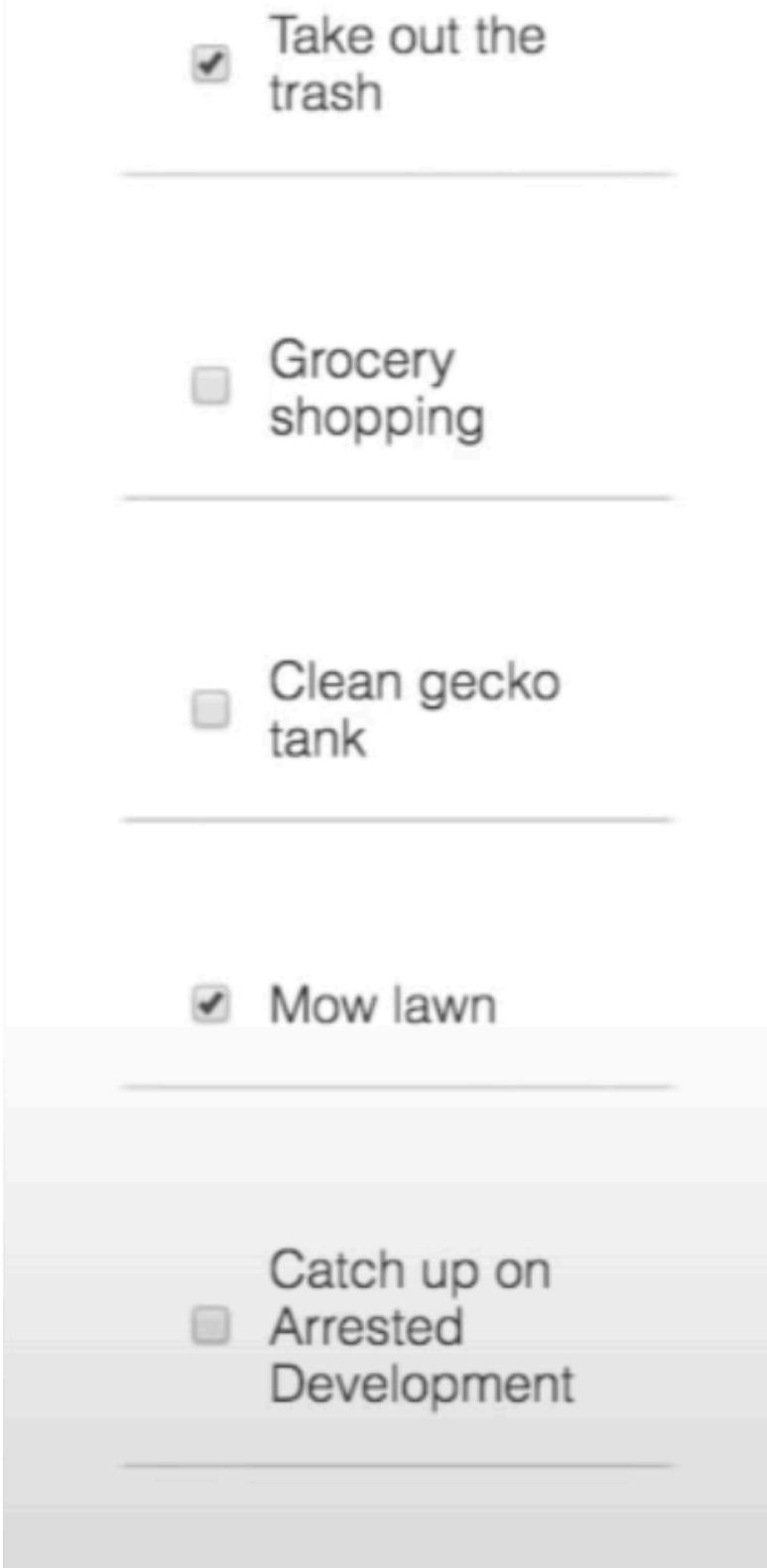
## Esercizio 10.5 (continua)

Continuare l'Esercizio 10.5 inserendo un gestore per l'evento **change** sul componente **ToDoItem** che, nel caso di un click su un item, aggiorni lo stato del componente **ToDoList** per tenere conto della variazione

Suggerimento

Il gestore dell'evento deve cambiare **ToDoList**, pertanto va definito come metodo di questo componente

Tuttavia, esso deve essere utilizzato come event handler sull'evento **change** di **ToDoItem**, pertanto dovrà essere passato come proprietà a quest'ultimo



# Esercizi

## Esercizio 10.6

Si realizzi un'applicazione che contiene una variabile di stato **unreadMessages**, ovvero un array di messaggi non letti

Si realizzi, quindi, un componente che venga mostrato se sono presenti messaggi non letti

Tale componente deve a sua volta mostrare i nuovi messaggi, ciascuno con un pulsante per leggerlo e rimuoverlo dall'array di messaggi non letti

# Form in React

Per inoltrare un form, potremmo inserire un pulsante ed ascoltare l'evento **submit** sul form e agire, eventualmente con un handler asincrono

```
export default function Form() {
  const [answer, setAnswer] = useState('');
  const [error, setError] = useState(null);
  const [status, setStatus] = useState('typing');

  if (status === 'success') { return <h1>That's right!</h1> }

  async function handleSubmit(e) {
    e.preventDefault();
    setStatus('submitting');
    try {
      await sendForm(answer);
      setStatus('success');
    } catch (err) {
      setStatus('typing');
      setError(err);
    }
  }

  return ( <>
    <p>In quale città si trova il miglior Politecnico d'Italia?</p>
    <form onSubmit={handleSubmit}>
      <textarea value={answer} onChange={e => setAnswer(e.target.value)} disabled={status === 'submitting'} />
      <button disabled={answer.length === 0 || status === 'submitting'}>Submit</button>
      {error !== null && <p className="Error">{error.message}</p>}
    </form>
  </> );
}
```

Come ragionare per realizzare la nostra UI interattiva?

1. Identifica i diversi stati "visivi" del componente
2. Determina che cosa può cambiare tali stati (es. input dell'utente, invio del form, ricezione di una risposta)
3. Rappresenta lo stato in memoria mediante useState
4. Connotti gli event handler allo stato

# Esercizi

## Esercizio 10.7

Realizzare un'applicazione React che consenta di inviare le informazioni per la partecipazione a un viaggio mediante un form come quello mostrato a destra. In particolare, si conservino le informazioni relative alle preferenze alimentari all'interno di un oggetto con tre proprietà booleane

The form consists of the following fields:

- First Name
- Last Name
- Age (dropdown menu showing "-- Please Choose a destination --")
- Gender selection (radio buttons for Male and Female)
- Destination (dropdown menu showing "-- Please Choose a destination --")
- Dietary preferences (checkboxes):
  - Vegan?
  - Kosher?
  - Lactose Free?
- Submit button

# Sincronizzazione con gli Effect

Alcuni componenti hanno necessità di sincronizzazione con sistemi esterni (ad esempio con un server per ricevere dati o inviare un log) quando il componente appare sullo schermo

Gli Effect permettono di eseguire codice subito dopo il primo rendering e dopo il re-rendering relativo ad alcune dipendenze

```
function MyComponent() {  
  useEffect(() => {  
    // Codice da eseguire  
  });  
  return <div />;  
}
```

Di default, infatti, gli Effect vengono eseguiti dopo ogni rendering

In realtà, la maggior parte degli Effect devono essere eseguiti solo in alcuni casi

Per esempio, un'animazione fade-in deve essere eseguita solo la prima volta che il componente appare

Una connessione ad una chat deve essere, invece, eseguita ogni volta che si cambia "stanza" della chat

Per far ciò basta usare un secondo argomento che specifica le props o le variabili di stato il cui cambiamento deve triggerare l'Effect

```
useEffect(() => {...}, [state1, state2, props4]);
```