REACT

React è una libreria JavaScript open-source sviluppata da Facebook per la creazione di interfacce utente (UI) dinamiche e interattive. È ampiamente utilizzato per la creazione di applicazioni web moderne, in particolare per la costruzione di interfacce utente complesse e scalabili.

```
// Importazione del modulo React e del modulo ReactDOM
import React from 'react';
import ReactDOM from 'react-dom';

// Definizione del componente Hello che restituisce un elemento
function Hello() {
  return <h1>Hello, world!</h1>;
}

// Rendering del componente Hello nell'elemento con ID "root" ne
ReactDOM.render(
  <React.StrictMode>
    <Hello />
  </React.StrictMode>,
  document.getElementById('root')
);
```

1. **Componenti**: React si basa sul concetto di componenti riutilizzabili. Un componente React è un blocco di costruzione autonomo che racchiude logica, struttura e stile associati a un'area specifica dell'interfaccia utente. I componenti possono essere composti insieme per creare interfacce complesse.

   ```
   import React from 'react';

   // Definizione del componente Saluto
   function Saluto(props) {
   ```

```
  // Estraiamo il nome dalle props
  const { nome } = props;

  // Restituiamo un elemento JSX che visualizza il messaggio
  return <h1>Ciao, {nome}!</h1>;
}

// Utilizzo del componente Saluto
function App() {
  return (
    <div>
      {/* Utilizziamo il componente Saluto passando il nome c
      <Saluto nome="Mario" />
      <Saluto nome="Luigi" />
      <Saluto nome="Peach" />
    </div>
  );
}

export default App;
```

2. **Virtual DOM**: React utilizza un'astrazione chiamata "Virtual DOM" per ottimizzare le prestazioni dell'applicazione. Il Virtual DOM è una rappresentazione leggera e virtuale della struttura del DOM reale. Quando lo stato di un componente cambia, React aggiorna il Virtual DOM, confronta la nuova rappresentazione con quella precedente e applica solo le modifiche necessarie al DOM reale. Questo approccio consente di migliorare le prestazioni e l'efficienza delle applicazioni React.

```
// Codice React
const element = (
  <div>
    <h1>Hello, world!</h1>
    <p>This is a paragraph.</p>
```

```
  </div>
);

ReactDOM.render(element, document.getElementById('root'));
```

3. **JSX**: JSX è una sintassi di estensione di JavaScript che consente di scrivere codice HTML all'interno di file JavaScript. Questo permette agli sviluppatori di definire la struttura dell'interfaccia utente in modo dichiarativo e intuitivo direttamente nel codice JavaScript. JSX viene quindi tradotto in chiamate di funzione React per creare gli elementi UI.

```
// Codice React con JSX
const element = <h1>Hello, world!</h1>;

ReactDOM.render(element, document.getElementById('root'));
```

4. **Stato e proprietà**: I componenti React possono avere stato interno (state) e possono ricevere dati esterni sotto forma di proprietà (props). Lo stato rappresenta i dati locali del componente che possono cambiare nel tempo, mentre le proprietà sono dati passati da componenti genitori a componenti figli.

```
// Componente React con stato e proprietà
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  render() {
    return <div>Count: {this.state.count}</div>;
```

```
  }
}

// Utilizzo del componente Counter
ReactDOM.render(<Counter />, document.getElementById('root'));
```

5. **Ciclo di vita del componente**: I componenti React passano attraverso un ciclo di vita predefinito che comprende diversi metodi che vengono invocati in diverse fasi del ciclo di vita del componente, come il montaggio, l'aggiornamento e lo smontaggio. Questi metodi consentono agli sviluppatori di eseguire azioni specifiche in determinati momenti durante il ciclo di vita del componente.

```
class LifecycleDemo extends React.Component {
  constructor(props) {
    super(props);
    console.log('Componente montato');
  }

  componentDidMount() {
    console.log('Componente aggiunto al DOM');
  }

  componentDidUpdate() {
    console.log('Componente aggiornato');
  }

  componentWillUnmount() {
    console.log('Componente rimosso dal DOM');
  }

  render() {
    return <div>Lifecycle Demo</div>;
  }
```

```
}

ReactDOM.render(<LifecycleDemo />, document.getElementById('
```

6. **Gestione degli eventi**: React supporta la gestione degli eventi attraverso un sistema simile a quello dei browser, consentendo agli sviluppatori di gestire eventi come clic, input, modifiche di stato e altro ancora.

```
class Button extends React.Component {
  handleClick() {
    console.log('Button clicked');
  }

  render() {
    return <button onClick={this.handleClick}>Click me</butt
  }
}

ReactDOM.render(<Button />, document.getElementById('root'))
```

7. **Librerie e strumenti aggiuntivi**: Oltre alla libreria principale React, esistono numerose librerie e strumenti aggiuntivi che estendono le funzionalità di React, come React Router per la gestione delle route, Redux per la gestione dello stato dell'applicazione, Axios per le richieste HTTP, e molti altri.

Regole JSX

1. **Tag di chiusura obbligatori**: Ogni tag deve essere chiuso, anche se non contiene alcun contenuto. Ad esempio, `<div></div>` anziché `<div />`.
2. **Elementi JSX racchiusi in un unico elemento**: Quando si restituiscono più elementi JSX da una funzione o un componente, devono essere racchiusi in un unico elemento padre. Ad esempio:

```
// Corretto
return (
  <div>
    <h1>Hello</h1>
    <p>World</p>
  </div>
);

// Errato
return (
  <h1>Hello</h1><p>World</p>
);
```

3. **Utilizzo di `className` anziché `class`**: Quando si definiscono classi CSS in JSX, è necessario utilizzare l'attributo `className` anziché `class`, poiché `class` è una parola chiave riservata in JavaScript. Ad esempio:

```
<div className="container">
  <p className="text">Hello, world!</p>
</div>
```

4. **Utilizzo delle parentesi graffe per inserire espressioni JavaScript**: Per inserire espressioni JavaScript all'interno di JSX, utilizzare le parentesi graffe `{}`. Ad esempio:

```
const name = 'World';
return <h1>Hello, {name}!</h1>;
```

5. **Gestione degli attributi**: Gli attributi in JSX devono essere in formato camelCase, ad esempio `onClick` anziché `onclick`, e gli attributi booleani devono essere inclusi con il valore `true` senza specificare il valore dell'attributo. Ad esempio:

```
<button onClick={handleClick} disabled={true}>Click me</bu
```

```
tton>
```

È anche possibile omettere `true` per gli attributi booleani, in questo caso l'attributo viene considerato vero:

```
<button disabled>Click me</button>
```

6. **Commenti JSX**: I commenti all'interno di JSX devono essere racchiusi tra parentesi graffe `{/* */}`. Ad esempio:

```
return (
  <div>
    {/* Questo è un commento JSX */}
    <h1>Hello, world!</h1>
  </div>
);
```

Composizione dei componenti

Concetto fondamentale in React che permette di creare interfacce utente complesse e riutilizzabili componendo insieme più componenti più piccoli. La composizione è il processo di combinare e annidare componenti per creare una struttura gerarchica di interfacce utente.

1. **Creazione di componenti riutilizzabili**: Si definiscono componenti più piccoli e riutilizzabili che rappresentano parti specifiche dell'interfaccia utente. Questi componenti possono essere pensati come "blocchi di costruzione" che possono essere combinati in modi diversi per creare interfacce più complesse.
2. **Composizione gerarchica**: I componenti possono essere composti gerarchicamente, dove un componente più grande può contenere e combinare uno o più componenti più piccoli come suoi figli. Questo consente di creare

una struttura ad albero di componenti, con componenti figli che possono essere riutilizzati in più parti dell'applicazione.

3. **Passaggio delle proprietà**: I componenti possono comunicare tra loro passando dati da componenti genitori a componenti figli tramite props. Questo permette di personalizzare il comportamento e l'aspetto dei componenti figli in base alle esigenze specifiche di ciascun contesto in cui vengono utilizzati.
4. **Combinazione dei componenti**: Utilizzando la composizione dei componenti, è possibile combinare componenti diversi per creare interfacce utente complesse e dinamiche. Ad esempio, si possono combinare componenti di layout con componenti di presentazione e componenti di logica per creare una pagina web completa.

Ecco un esempio di come funziona la composizione dei componenti in React:

```
// Definizione del componente Button
function Button(props) {
  return <button onClick={props.onClick}>{props.label}</butto
n>;
}

// Definizione del componente App che compone il componente B
utton
function App() {
  function handleClick() {
    console.log('Button clicked');
  }

  return (
    <div>
      <h1>My App</h1>
      <Button onClick={handleClick} label="Click me" />
    </div>
  );
}
```

```
// Rendering del componente App nell'elemento root del DOM
ReactDOM.render(<App />, document.getElementById('root'));
```

In questo esempio, abbiamo un componente `Button` che rappresenta un pulsante. Il componente `App` compone il componente `Button` insieme ad altri elementi per creare l'interfaccia utente completa. Il componente `Button` viene personalizzato attraverso le props `onClick` e `label`, che vengono passate dal componente `App`. Questo è un esempio semplice di composizione dei componenti in React.

Rendering condizionale

Utilizzo di operatore ternario (Conditional Rendering): Questo è il metodo più comune per eseguire il rendering condizionale in React. Si utilizza l'operatore ternario per valutare una condizione e restituire un elemento JSX in base al risultato della condizione. Ad esempio:

Consente di mostrare o nascondere elementi o componenti in base a una condizione. Ci sono diversi modi per eseguire il rendering condizionale in React, che dipendono dalle esigenze specifiche dell'applicazione.

```
function ComponenteConCondizione(props) {
  const mostraElemento = props.condizione;
  return (
    <div>
      {mostraElemento ? <p>Elemento visibile</p> : <p>Element
o nascosto</p>}
    </div>
  );
}
```

Utilizzo di `&&` (Conditional AND Operator): Invece dell'operatore ternario, si può utilizzare l'operatore logico `&&` per eseguire il rendering condizionale in React.

Questo è utile quando si desidera rendere un elemento solo se una determinata condizione è vera. Ad esempio:

```
function ComponenteConCondizione(props) {
  const mostraElemento = props.condizione;
  return (
    <div>
      {mostraElemento && <p>Elemento visibile</p>}
    </div>
  );
}
```

Utilizzo di `if` all'interno della funzione di render: In React è possibile utilizzare istruzioni `if` all'interno della funzione di render per eseguire il rendering condizionale. Questo approccio è utile quando la logica di rendering è più complessa e richiede più istruzioni condizionali. Ad esempio:

```
function ComponenteConCondizione(props) {
  if (props.condizione) {
    return <p>Elemento visibile</p>;
  } else {
    return <p>Elemento nascosto</p>;
  }
}
```

Rendering condizionale con un operatore switch: In alcuni casi, potrebbe essere utile utilizzare un'istruzione switch per il rendering condizionale. Ad esempio, se si hanno più condizioni da gestire. Ecco un esempio:

```
function ComponenteConCondizione(props) {
  switch (props.valore) {
    case 'A':
      return <p>Opzione A selezionata</p>;
```

```
    case 'B':
      return <p>Opzione B selezionata</p>;
    default:
      return <p>Altra opzione selezionata</p>;
  }
}
```

Javascript e JSX

Interpolazione di JavaScript in JSX: È possibile incorporare espressioni JavaScript all'interno di JSX utilizzando le parentesi graffe `{}`. Questo consente di inserire dinamicamente valori e espressioni JavaScript all'interno del codice JSX. Ad esempio:

```
const name = 'World';
const element = <h1>Hello, {name}!</h1>;
```

Componenti in JSX: I componenti React possono essere utilizzati direttamente all'interno di JSX come se fossero elementi HTML. Ad esempio:

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

const element = <Welcome name="World" />;
```

Attributi in JSX: Gli attributi in JSX sono scritti utilizzando la sintassi camelCase, ad esempio `className` invece di `class`. Questo perché `class` è una parola chiave riservata in JavaScript. Ad esempio:

```
const element = <div className="container">Hello, world!</div
```

```
>;
```

Elementi JSX multilinea: Gli elementi JSX multilinea devono essere racchiusi tra parentesi. Ad esempio:

```
const element = (
  <div>
    <h1>Hello</h1>
    <p>World</p>
  </div>
);
```

Commenti in JSX: I commenti in JSX devono essere racchiusi tra parentesi graffe `{/* */}`. Ad esempio:

```
const element = (
  <div>
    {/* Questo è un commento JSX */}
    <h1>Hello, world!</h1>
  </div>
);
```

Utilizzo di JavaScript puro
: È comunque possibile utilizzare JavaScript puro all'interno di JSX. Ad esempio:

```
const message = 'Hello, world!';
const element = <h1>{message}</h1>;
```

Passare dati ai componenti

In React, è possibile passare dati da un componente padre a un componente figlio utilizzando le props (abbreviazione di "properties", proprietà). Le props sono un meccanismo fondamentale per la comunicazione tra i componenti in React e consentono di personalizzare il comportamento e l'aspetto dei componenti figli in base alle esigenze specifiche.

Definizione delle props: Le props vengono passate come attributi agli elementi JSX quando si utilizzano i componenti. Ad esempio, se si ha un componente `Saluto` che accetta una prop `nome`, si può passare il nome come attributo quando si utilizza il componente:

```
<Saluto nome="Mario" />
```

Accesso alle props: All'interno del componente figlio, le props vengono accessibili come un oggetto JavaScript chiamato `props`. Ad esempio, per accedere alla prop `nome` all'interno del componente `Saluto`, si utilizza `this.props.nome`.

Utilizzo delle props: Le props possono essere utilizzate all'interno del componente figlio per personalizzare il rendering del componente in base ai dati passati dal componente padre. Ad esempio, nel componente `Saluto`, si può utilizzare la prop `nome` per visualizzare un messaggio di saluto personalizzato:

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

Passaggio di dati complessi: Le props possono contenere dati complessi come oggetti o array. È possibile passare qualsiasi tipo di dato valido come prop a un componente.

Ecco un esempio completo che illustra il passaggio dei dati ai componenti tramite props:

```
// Definizione del componente Saluto
function Saluto(props) {
  return <h1>Ciao, {props.nome}!</h1>;
}

// Utilizzo del componente Saluto passando una prop nome
ReactDOM.render(
  <Saluto nome="Mario" />,
  document.getElementById('root')
);
```

Eventi

In React, è possibile gestire eventi utilizzando la sintassi degli eventi simile a quella dei browser standard, ma con alcune differenze sintattiche.

1. **Aggiunta di gestori eventi**: Per aggiungere un gestore evento a un elemento JSX, si utilizza la sintassi `onNomeEvento`, dove `NomeEvento` è il nome dell'evento JavaScript. Ad esempio, `onClick` per il clic del mouse, `onChange` per il cambiamento di valore di un input, ecc.

2. **Definizione dei gestori eventi**: I gestori eventi sono delle funzioni JavaScript definite all'interno del componente React. Queste funzioni verranno chiamate quando si verifica l'evento corrispondente. Per esempio:

```
function handleClick() {
  console.log('Button clicked');
}

<button onClick={handleClick}>Click me</button>
```

3. **Accesso all'evento**: Quando si definisce un gestore evento, è possibile accedere all'evento associato tramite un parametro. Il parametro di default è di solito chiamato `event`, ma può essere nominato diversamente. Ad esempio:

```
function handleChange(event) {
  console.log('Input value:', event.target.value);
}

<input type="text" onChange={handleChange} />
```

In questo esempio, `event` è l'oggetto dell'evento associato all'evento `onChange`. `event.target` si riferisce all'elemento DOM che ha scatenato l'evento, e `event.target.value` contiene il valore dell'elemento (nel nostro caso, il valore dell'input).

4. **Prevenire l'azione predefinita**: In React, per prevenire l'azione predefinita di un evento (come il ricaricamento della pagina dopo un clic su un link), è possibile chiamare il metodo `preventDefault()` sull'oggetto evento. Ad esempio:

```
function handleClick(event) {
  event.preventDefault();
  console.log('Link clicked');
}

<a href="#" onClick={handleClick}>Click me</a>
```

5. **Fare riferimento all'elemento**: In alcuni casi, può essere necessario fare riferimento direttamente all'elemento DOM all'interno del gestore evento. In React, si può utilizzare la funzione `ref` per ottenere un riferimento all'elemento. Ad esempio:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.inputRef = React.createRef();
  }

  handleClick() {
    console.log('Input value:', this.inputRef.current.value);
  }

  render() {
    return (
      <div>
        <input type="text" ref={this.inputRef} />
        <button onClick={this.handleClick}>Get value</button>
      </div>
    );
  }
}
```

In questo esempio, `this.inputRef` contiene un riferimento all'elemento input, che può essere utilizzato all'interno del gestore evento `handleClick`.

React Hooks

Sono una funzionalità introdotta in React 16.8 che permette di utilizzare lo state e altre funzionalità di React all'interno di componenti funzionali, permettendo loro di avere uno stato interno e di sfruttare il ciclo di vita del componente. Prima dell'introduzione dei React Hooks, le funzionalità di stato e ciclo di vita potevano essere utilizzate solo nei componenti di classe.

I React Hooks sono funzioni speciali che consentono di "agganciarsi" a funzionalità specifiche di React all'interno di componenti funzionali.

useState(): Per aggiungere lo stato locale a un componente funzionale.

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

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</
button>
    </div>
  );
}
```

useEffect(): Per eseguire effetti collaterali in un componente funzionale, come chiamate API, abbonamenti a eventi, ecc.

```
import React, { useState, useEffect } from 'react';

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</
button>
    </div>
```

```
  );
}
```

useContext()
: Per accedere al contesto React all'interno di un componente funzionale.

```
import React, { useContext } from 'react';
import MyContext from './MyContext';

function MyComponent() {
  const value = useContext(MyContext);
  return <p>{value}</p>;
}
```

useReducer()
: Per gestire lo stato complesso utilizzando un riduttore, simile a Redux.

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

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0
});
```

```
  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}
>Increment</button>
      <button onClick={() => dispatch({ type: 'decrement' })}
>Decrement</button>
    </div>
  );
}
```

`useState`

Consente di aggiungere uno stato locale a un componente funzionale. Lo stato aggiunto con `useState` è privato al componente in cui è definito e non è condiviso con altri componenti.

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

function Counter() {
  // Definizione dello stato locale "count" con valore iniziale
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>Count: {count}</p>
      {/* Aggiornamento dello stato "count" quando viene clicca
      <button onClick={() => setCount(count + 1)}>Increment</bu
    </div>
  );
}
```

`setState`

Metodo utilizzato per aggiornare lo stato di un componente. È una funzione asincrona che accetta un nuovo stato come argomento e informa React di aggiornare il componente con il nuovo stato.

Quando viene chiamato `setState`, React pianifica un aggiornamento del componente e lo rende effettivo al termine del ciclo di vita corrente, garantendo che l'interfaccia utente venga aggiornata in modo coerente e senza perdita di prestazioni.

Utilizzo di oggetti: `setState` può essere chiamato con un oggetto contenente gli aggiornamenti dello stato. Ad esempio:

```
this.setState({ count: this.state.count + 1 });
```

Utilizzo di funzioni
: `setState` può essere chiamato con una funzione che riceve lo stato precedente come argomento e restituisce un nuovo stato. Questo approccio è consigliato quando si aggiorna lo stato basandosi sullo stato precedente per evitare problemi di concorrenza. Ad esempio:

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

Passaggio di un oggetto a setState all'interno di un loop: Questo è un errore comune che si verifica quando si chiama `setState` all'interno di un loop, passando lo stesso oggetto di stato in ogni iterazione.

Questo comportamento non comporterà gli aggiornamenti desiderati dello stato. Invece, si dovrebbe utilizzare il secondo formato, passando una funzione che riceve lo stato precedente e restituisce il nuovo stato, in modo da garantire che ciascuna chiamata a `setState` utilizzi l'ultimo stato corretto.

Chiamata a setState all'interno di useEffect senza dipendenze

: Se si utilizza `setState` all'interno di un effetto React (utilizzando il hook `useEffect`), è importante specificare le dipendenze corrette per l'effetto. Se `setState` dipende dallo stato o da altre variabili, è necessario elencarle come dipendenze nell'array di dipendenze di `useEffect`. In caso contrario, si potrebbero verificare problemi di rendering e loop infiniti.

Chiamata a setState in un componente non montato
: Se si chiama `setState` in un componente che non è ancora montato o che è stato smontato, si può incorrere in un errore. È importante assicurarsi che il componente sia montato prima di chiamare `setState`, ad esempio verificando `this.mounted` o utilizzando gli hook di montaggio come `useEffect`.

Stato asincrono
: `setState` è asincrono e non garantisce un aggiornamento immediato dello stato. Se si dipende dallo stato attualmente impostato, è necessario utilizzare la funzione di aggiornamento dello stato per assicurarsi di avere lo stato più recente. Ad esempio:

```
// Questo potrebbe non funzionare come previsto
this.setState({ count: this.state.count + 1 });

// Questo garantisce che lo stato sia aggiornato correttament
e
this.setState((prevState) => ({
  count: prevState.count + 1
}));
```

`useEffect`

Utilizzato per eseguire effetti collaterali in componenti funzionali. Gli effetti collaterali includono operazioni asincrone, accesso a risorse esterne come API, manipolazione del DOM e altro ancora. `useEffect` viene eseguito dopo ogni renderizzazione del componente e può essere utilizzato per effettuare operazioni che non devono influenzare direttamente il rendering del componente.

Ecco una panoramica su come `useEffect` funziona e come viene utilizzato:

1. **Definizione dell'effetto collaterale**: Si definisce un effetto collaterale all'interno di un componente funzionale utilizzando il hook `useEffect`. L'effetto collaterale è una funzione che verrà eseguita dopo che il componente è stato renderizzato. Ad esempio:

```
useEffect(() => {
  // Effetto collaterale
  console.log('Component rendered');
});
```

2. **Gestione delle dipendenze**: È possibile specificare le dipendenze dell'effetto collaterale passando un array come secondo argomento di `useEffect`. Le dipendenze sono variabili o stati che l'effetto collaterale dipende. Se una dipendenza cambia tra renderizzazioni, l'effetto collaterale verrà eseguito di nuovo. Ad esempio:

```
const [count, setCount] = useState(0);

useEffect(() => {
  console.log('Count changed:', count);
}, [count]); // Dipendenza
```

In questo caso, l'effetto collaterale verrà eseguito solo quando il valore di `count` cambierà.

3. **Effetto collaterale senza dipendenze**: Se non ci sono dipendenze, è possibile passare un array vuoto come secondo argomento di `useEffect`. In questo modo, l'effetto collaterale verrà eseguito solo una volta dopo la prima renderizzazione del componente. Ad esempio:

```
useEffect(() => {
  console.log('Component mounted');
```

```
}, []); // Nessuna dipendenza
```

4. **Cleanup dell'effetto collaterale**: L'effetto collaterale può anche restituire una funzione di cleanup che verrà eseguita quando il componente viene smontato o prima di eseguire il prossimo effetto collaterale. Ad esempio, per annullare sottoscrizioni a eventi o pulire risorse. Ecco un esempio:

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log('Timer tick');
  }, 1000);

  return () => {
    clearInterval(timer);
    console.log('Timer cleared');
  };
}, []); // Nessuna dipendenza
```

In questo caso, il timer verrà cancellato quando il componente viene smontato o prima di eseguire il prossimo effetto collaterale.

Come costruire un'applicazione in React:

1. Dividere la UI in una gerarchia di componenti
2. Costruire una versione statica in react
3. Identificare lo stato dell'applicazione
4. Identificare dove dovrebbe stare lo stato
5. Gestire il flusso inverso dei dati per la gestione del cambiamento di stato