



# Ripasso React

usando esempi guidati

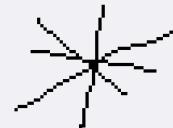
# Modulo 1

i Fondamenti

# Modulo 1: Ripasso e Consolidamento dei Fondamenti

## Workshop Pratico: Costruzione di una Todo List Interattiva

- Un po`di Teoria veloce: Ripasso di componenti funzionali, JSX, props e useState
- Esercizio guidato:
  - Implementazione della struttura base dell'app
  - Creazione degli stati per gli elementi della lista
  - Funzioni per aggiungere/completare/eliminare tasks
  - Styling di base con CSS modules



# Componenti Funzionali e JSX

- Struttura di un componente funzionale

```
1 import React from 'react';
2
3 function Greeting({ name }) {
4   return (
5     <div className="greeting">
6       <h1>Ciao, {name}!</h1>
7     </div>
8   );
9 }
10
11 export default Greeting;
```



# Componenti Funzionali e JSX

## Regole di JSX

- Usa parentesi graffe `{}` per inserire JavaScript
- Attributi HTML in camelCase: `className` invece di `class`
- Self-closing tags richiedono `/`: `<img />`
- Ogni componente deve ritornare un singolo element

```
● ● ●  
1 function UserProfile() {  
2   return (  
3     <>  
4       <h2>Profilo Utente</h2>  
5       <p>Informazioni personali</p>  
6     </>  
7   );  
8 }
```



# Componenti Funzionali e JSX

## Importanza della scomposizione

- Componenti piccoli e focalizzati
- Riutilizzabilità e manutenibilità
- Separa logica di presentazione e business logic
- Facilità di testing



# Props e Gestione degli Stati

- Props: passaggio di dati

```
1 // Componente genitore
2 function App() {
3   return <UserCard name="Marco" role="Developer" isAdmin={true} />;
4 }
5
6 // Componente figlio
7 function UserCard({ name, role, isAdmin }) {
8   return (
9     <div className="card">
10       <h3>{name}</h3>
11       <p>Ruolo: {role}</p>
12       {isAdmin && <span className="badge">Admin</span>}
13     </div>
14   );
15 }
```



# Props e Gestione degli Stati

- useState: creazione e aggiornamento

```
1 import React, { useState } from 'react';
2
3 function Counter() {
4   const [count, setCount] = useState(0);
5
6   return (
7     <div>
8       <p>Conteggio: {count}</p>
9       <button onClick={() => setCount(count + 1)}>
10         Incrementa
11       </button>
12     </div>
13   );
14 }
```

# Props e Gestione degli Stati

## Confronto stato locale vs globale

### Stato locale:

- Isolato al componente
- Ideale per UI temporanea (form, toggle)
- Più semplice da gestire

### Stato globale:

- Condiviso tra componenti
- Per dati applicativi (utente, carrello)
- Richiede context, Redux o altre soluzioni



# Props e Gestione degli Stati

Pattern per aggiornare correttamente

Per oggetti:

```
1 const [user, setUser] = useState({ name: 'Mario', age: 25 });
2
3 // ✗ Errato: sovrascrive solo una proprietà
4 setUser({ name: 'Luigi' });
5
6 // ✓ Corretto: preserva le altre proprietà
7 setUser({ ...user, name: 'Luigi' });
```



# Props e Gestione degli Stati

Pattern per aggiornare correttamente

Per array::

```
1 const [items, setItems] = useState(['mela', 'banana']);
2
3 // Aggiungere elemento
4 setItems([...items, 'arancia']);
5
6 // Rimuovere elemento
7 setItems(items.filter(item => item !== 'banana'));
8
9 // Aggiornare elemento
10 setItems(items.map(item =>
11   item === 'mela' ? 'mela verde' : item
12 ));
```



# Props e Gestione degli Stati

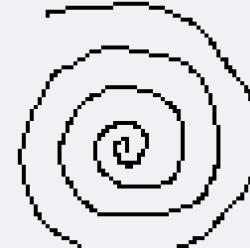
**Best practice: useState vs refs/effect**

Usa `useState` quando:

- Il dato deve influenzare il rendering
- Vuoi reagire ai cambiamenti del dato

Usa `useEffect` quando:

- Devi sincronizzare con sistemi esterni
- Devi eseguire side effect in risposta a cambiamenti



# Esercizio 1: Costruzione di Todo List Interattiva

## Implementazione della Struttura Base

- Creazione dei componenti principali:
  - `App.jsx`: Componente radice
  - `TodoList.jsx`: Contenitore delle attività
  - `TodoItem.jsx`: Singola attività
  - `AddTodoForm.jsx`: Form per aggiungere nuovi task



# Esercizio 1: Costruzione di Todo List Interattiva

## Implementazione della Struttura Base

```
1 // App.jsx - Struttura iniziale
2 import { useState } from 'react';
3 import TodoList from './components/TodoList';
4 import AddTodoForm from './components/AddTodoForm';
5 import styles from './App.module.css';
6
7 function App() {
8   // Definiremo qui gli stati e le funzioni principali
9
10  return (
11    <div className={styles.container}>
12      <h1>Todo List App</h1>
13      <AddTodoForm />
14      <TodoList />
15    </div>
16  );
17}
18
19 export default App;
```

# Esercizio 1: Costruzione di Todo List Interattiva

## Creazione degli Stati

- Implementazione dello stato principale nell'App
- Struttura dati per i todo (array di oggetti con id, testo, stato)

```
1 // App.jsx - Con stato
2 import { useState } from 'react';
3 import TodoList from './components/TodoList';
4 import AddTodoForm from './components/AddTodoForm';
5 import styles from './App.module.css';
6
7 function App() {
8   const [todos, setTodos] = useState([
9     { id: 1, text: 'Imparare React', completed: false },
10    { id: 2, text: 'Creare una Todo List', completed: true }
11  ]);
12
13  // Aggiungeremo qui le funzioni per manipolare lo stato
14
15  return (
16    <div className={styles.container}>
17      <h1>Todo List App</h1>
18      <AddTodoForm />
19      <TodoList todos={todos} />
20    </div>
21  );
22}
23
24 export default App;
```



# Esercizio 1: Costruzione di Todo List Interattiva

Implementazione delle Funzioni (15 min)

- Funzione `addTodo`: aggiunta di un nuovo task
- Funzione `toggleTodo`: completamento/incompletamento di un task
- Funzione `deleteTodo`: rimozione di un task
- Passaggio delle funzioni ai componenti figli tramite props



```
1 // App.jsx - Con funzioni complete
2 import { useState } from 'react';
3 import TodoList from './components/TodoList';
4 import AddTodoForm from './components/AddTodoForm';
5 import styles from './App.module.css';
6
7 function App() {
8   const [todos, setTodos] = useState([
9     { id: 1, text: 'Imparare React', completed: false },
10    { id: 2, text: 'Creare una Todo List', completed: true }
11  ]);
12
13 // Generatore di ID unici semplice
14 const generateId = () => Math.floor(Math.random() * 10000);
15
16 // Funzione per aggiungere un nuovo todo
17 const addTodo = (text) => {
18   if (text.trim() === '') return;
19
20   const newTodo = {
21     id: generateId(),
22     text,
23     completed: false
24   };
25
26   setTodos([...todos, newTodo]);
27 };
28
29 // Funzione per cambiare lo stato di un todo
30 const toggleTodo = (id) => {
31   setTodos(todos.map(todo =>
32     todo.id === id ? { ...todo, completed: !todo.completed } : todo
33   ));
34 };
35
36 // Funzione per eliminare un todo
37 const deleteTodo = (id) => {
38   setTodos(todos.filter(todo => todo.id !== id));
39 };
40
41 return (
42   <div className={styles.container}>
43     <h1>Todo List App</h1>
44     <AddTodoForm addTodo={addTodo} />
45     <TodoList
46       todos={todos}
47       toggleTodo={toggleTodo}
48       deleteTodo={deleteTodo}
49     />
50   </div>
51 );
52 }
53
54 export default App;
```

# Esercizio 1: Costruzione di Todo List Interattiva

Completamento dei Componenti Figli (10 min)

- Implementazione di `TodoList`, `TodoItem` e `AddTodoForm`
- Utilizzo delle props per connettere i componenti

```
1 // TodoItem.jsx
2 import styles from './TodoItem.module.css';
3
4 function TodoItem({ todo, toggleTodo, deleteTodo }) {
5   return (
6     <li className={`${$styles.item} ${todo.completed ? styles.completed : ''}`}>
7       <input
8         type="checkbox"
9         checked={todo.completed}
10        onChange={() => toggleTodo(todo.id)}
11        className={styles.checkbox}
12       />
13       <span className={styles.text}>{todo.text}</span>
14       <button
15         onClick={() => deleteTodo(todo.id)}
16         className={styles.deleteBtn}
17       >
18         Elimina
19       </button>
20     </li>
21   );
22 }
23
24 export default TodoItem;
```



```
1 // TodoList.jsx
2 import TodoItem from './TodoItem';
3 import styles from './TodoList.module.css';
4
5 function TodoList({ todos, toggleTodo, deleteTodo }) {
6   return (
7     <ul className={styles.list}>
8       {todos.length === 0 ? (
9         <p>Nessuna attività da svolgere.</p>
10      ) : (
11        todos.map(todo => (
12          <TodoItem
13            key={todo.id}
14            todo={todo}
15            toggleTodo={toggleTodo}
16            deleteTodo={deleteTodo}
17          />
18        ))
19      )}
20     </ul>
21   );
22 }
23
24 export default TodoList;
```

```
1 // AddTodoForm.jsx
2 import { useState } from 'react';
3 import styles from './AddTodoForm.module.css';
4
5 function AddTodoForm({ addTodo }) {
6   const [text, setText] = useState('');
7
8   const handleSubmit = (e) => {
9     e.preventDefault();
10    addTodo(text);
11    setText(''); // Reset input dopo l'aggiunta
12  };
13
14  return (
15    <form onSubmit={handleSubmit} className={styles.form}>
16      <input
17        type="text"
18        value={text}
19        onChange={(e) => setText(e.target.value)}
20        placeholder="Aggiungi una nuova attività"
21        className={styles.input}
22      />
23      <button type="submit" className={styles.button}>
24        Aggiungi
25      </button>
26    </form>
27  );
28 }
29
30 export default AddTodoForm;
```

# Esercizio 1: Costruzione di Todo List Interattiva

Un po` di stile!

```
1 /* App.module.css */
2 .container {
3   max-width: 600px;
4   margin: 0 auto;
5   padding: 20px;
6   font-family: 'Arial', sans-serif;
7 }
```

```
1 /* TodoList.module.css */
2 .list {
3   list-style-type: none;
4   padding: 0;
5   margin: 20px 0;
6 }
```



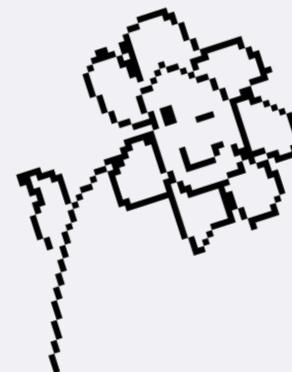
```
1 /* AddTodoForm.module.css */
2 .form {
3   display: flex;
4   margin-bottom: 20px;
5 }
6
7 .input {
8   flex-grow: 1;
9   padding: 10px;
10 border: 1px solid #ddd;
11 border-radius: 4px 0 0 4px;
12 font-size: 16px;
13 }
14
15 .button {
16   background-color: #4caf50;
17   color: white;
18   border: none;
19   border-radius: 0 4px 4px 0;
20   padding: 10px 15px;
21   cursor: pointer;
22   transition: background-color 0.2s;
23 }
24
25 .button:hover {
26   background-color: #45a049;
27 }
```

```
1 /* TodoItem.module.css */
2 .item {
3   display: flex;
4   align-items: center;
5   padding: 10px;
6   margin-bottom: 10px;
7   border-radius: 4px;
8   background-color: #f9f9f9;
9   transition: background-color 0.2s;
10 }
11
12 .item:hover {
13   background-color: #f0f0f0;
14 }
15
16 .completed {
17   opacity: 0.6;
18   text-decoration: line-through;
19 }
20
21 .checkbox {
22   margin-right: 10px;
23 }
24
25 .text {
26   flex-grow: 1;
27 }
28
29 .deleteBtn {
30   background-color: #ff4d4d;
31   color: white;
32   border: none;
33   border-radius: 4px;
34   padding: 5px 10px;
35   cursor: pointer;
36   transition: background-color 0.2s;
37 }
38
39 .deleteBtn:hover {
40   background-color: #ff3333;
41 }
```

# Esercizio 1: Costruzione di Todo List Interattiva

## Compiti Individuali

1. Aggiungere una funzionalità di filtro per visualizzare "Tutti/Attivi/Completati"
2. Implementare la persistenza dei todo nel localStorage
3. Aggiungere una funzione di modifica testo per i todo esistenti
4. Creare uno stato per tenere traccia delle statistiche (totale, completati, da fare)





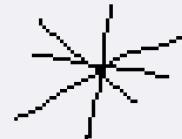
# Modulo 2

## Hooks

# Modulo 2: Hooks in Profondità

Workshop Pratico: Applicazione per Previsioni Meteo

- Teoria: `useEffect`, `useRef`, `useMemo`, `useCallback`
- Esercizio guidato:
  - Configurazione di chiamate API con `useEffect`
  - Memorizzazione dei risultati con `useMemo`
  - Ottimizzazione delle funzioni con `useCallback`
  - Gestione input di ricerca con `useRef`
  - Implementazione di una cache per le ricerche recenti
  - Debugging di problemi comuni con gli hooks



— [www.Duccio.ME](http://www.Duccio.ME) —

# useEffect

## Ciclo di vita dei componenti con gli Hooks

- **Concetto:** `useEffect` sostituisce i metodi del ciclo di vita nei componenti a classe
- **Equivalenze:**
  - `componentDidMount` → `useEffect(() => {}, [])`
  - `componentDidUpdate` → `useEffect(() => {})`



# useEffect

## Pattern di useEffect

```
1 useEffect(() => {
2   // 1. Codice da eseguire dopo il rendering
3   console.log('Componente renderizzato');
4
5   // 3. Funzione di cleanup (opzionale)
6   return () => {
7     console.log('Pulizia prima del prossimo effect o unmount');
8   };
9 }, /* 2. Array delle dipendenze */);
```



# Casi d'uso comuni di useEffect

## Fetching dei dati



```
1 function UserProfile({ userId }) {
2   const [user, setUser] = useState(null);
3   const [loading, setLoading] = useState(true);
4
5   useEffect(() => {
6     setLoading(true);
7     fetch(`https://api.example.com/users/${userId}`)
8       .then(response => response.json())
9       .then(data => {
10         setUser(data);
11         setLoading(false);
12     });
13   }, [userId]); // Si esegue solo quando userId cambia
14
15   if (loading) return <p>Caricamento...</p>;
16   return <div>{user.name}</div>;
17 }
```

# Casi d'uso comuni di useEffect

## Sottoscrizioni a eventi

```
 1 function WindowSizeTracker() {
 2   const [windowSize, setWindowSize] = useState({
 3     width: window.innerWidth,
 4     height: window.innerHeight
 5   });
 6
 7   useEffect(() => {
 8     function handleResize() {
 9       setWindowSize({
10         width: window.innerWidth,
11         height: window.innerHeight
12       });
13     }
14
15     window.addEventListener('resize', handleResize);
16
17     // Pulizia: rimuove l'event listener quando il componente viene smontato
18     return () => window.removeEventListener('resize', handleResize);
19   }, []); // Array vuoto = esegui solo al mount
20
21   return (
22     <div>
23       Larghezza: {windowSize.width}px, Altezza: {windowSize.height}px
24     </div>
25   );
26 }
```

# Casi d'uso comuni di useEffect

## Modifiche al DOM

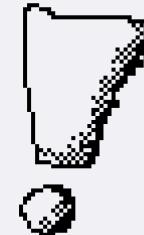
```
 1 function AutoFocusInput() {  
 2   const inputRef = useRef(null);  
 3  
 4   useEffect(() => {  
 5     // Focus automatico sull'input quando il componente viene montato  
 6     inputRef.current.focus();  
 7   }, []);  
 8  
 9   return <input ref={inputRef} type="text" />;  
10 }
```



# Altro su useEffect

## Pulizia con la funzione di return

- Necessaria per evitare memory leak
- Esempi di pulizia:
  - Cancellare sottoscrizioni
  - Cancellare timers
  - Rimuovere event listeners



```
1 useEffect(() => {
2   const timer = setTimeout(() => {
3     console.log('Questo messaggio appare dopo 2 secondi');
4   }, 2000);
5
6 // Pulizia: cancella il timer se il componente viene smontato prima
7 return () => clearTimeout(timer);
8 }, []);
```



# Altro su useEffect

## Attenzione alle dipendenze mancanti

- React ESLint plugin avvisa delle dipendenze mancanti
- Ignorare l'avviso può causare bug difficili da trovare
- Soluzione: includere tutte le variabili usate nell'effect

```
1 // ✗ Problematico: count cambia ma l'effect non si aggiorna
2 useEffect(() => {
3   document.title = `Hai cliccato ${count} volte`;
4 }, []); // Dipendenza mancante: count
5
6 // ✅ Corretto: l'effect si aggiorna quando count cambia
7 useEffect(() => {
8   document.title = `Hai cliccato ${count} volte`;
9 }, [count]);
```



# useRef

Cos'è un "ref" e quando usarlo

- Contenitore mutabile che persiste per tutta la vita del componente
- Non causa re-render quando cambia il suo valore
- Utile per valori che devono persistere tra i render

```
1 function Timer() {
2   const countRef = useRef(0);
3
4   function handleClick() {
5     countRef.current = countRef.current + 1;
6     console.log(`Hai cliccato ${countRef.current} volte`);
7     // Non causa re-render!
8   }
9
10  return <button onClick={handleClick}>ClICCami</button>;
11 }
```



# useRef

## Differenza tra `.current` e `state`

- `useState`:
  - Causa re-render quando il valore cambia
  - Ideale per dati che influenzano il rendering
- `useRef`:
  - Non causa re-render quando `.current` cambia
  - Ideale per valori "invisibili" al render

```
1 function Counter() {
2   // Causa re-render quando cambia
3   const [visibleCount, setVisibleCount] = useState(0);
4
5   // Non causa re-render quando cambia
6   const hiddenCountRef = useRef(0);
7
8   function handleClick() {
9     setVisibleCount(visibleCount + 1); // Aggiorna UI
10    hiddenCountRef.current += 1;      // Non aggiorna UI
11    console.log(`Hidden count: ${hiddenCountRef.current}`);
12  }
13
14  return (
15    <div>
16      <p>Conteggio visibile: {visibleCount}</p>
17      <button onClick={handleClick}>Incrementa</button>
18    </div>
19  );
20 }
```

# useRef: casi d'uso

Accesso al DOM

```
1 function VideoPlayer() {
2   const videoRef = useRef(null);
3
4   function handlePlay() {
5     videoRef.current.play();
6   }
7
8   function handlePause() {
9     videoRef.current.pause();
10  }
11
12  return (
13    <div>
14      <video ref={videoRef} src="video.mp4" />
15      <button onClick={handlePlay}>Play</button>
16      <button onClick={handlePause}>Pause</button>
17    </div>
18  );
19 }
```

# useRef: casi d'uso

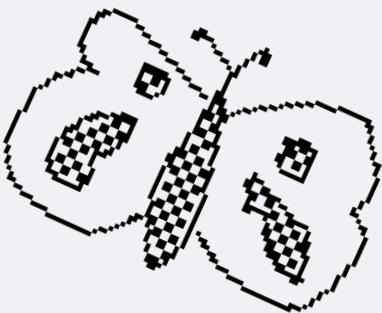
Valori persistenti che non causano re-render

```
● ○ ●  
1 function RenderCounter() {  
2   const [, forceRender] = useState({});  
3   const renderCount = useRef(0);  
4  
5   useEffect(() => {  
6     renderCount.current += 1;  
7   });  
8  
9   return (  
10    <div>  
11      <p>Questo componente ha renderizzato  
12        {renderCount.current} volte</p>  
13        <button onClick={() => forceRender({})}>Forza  
14          re-render</button>  
15    );  
16 }
```



# useRef: casi d'uso

Memorizzazione di timeouts/intervals



```
1 function Debounce() {
2   const [inputValue, setInputValue] = useState("");
3   const [debouncedValue, setDebouncedValue] = useState("");
4   const timerRef = useRef(null);
5
6   function handleChange(e) {
7     setInputValue(e.target.value);
8
9     // Cancella il timer precedente se esiste
10    if (timerRef.current) clearTimeout(timerRef.current);
11
12    // Imposta un nuovo timer
13    timerRef.current = setTimeout(() => {
14      setDebouncedValue(e.target.value);
15    }, 500);
16  }
17
18 // Pulisci il timer quando il componente viene smontato
19 useEffect(() => {
20   return () => {
21     if (timerRef.current) clearTimeout(timerRef.current);
22   };
23 }, []);
24
25 return (
26   <div>
27     <input
28       type="text"
29       value={inputValue}
30       onChange={handleChange}
31       placeholder="Scrivi qualcosa..."'
32     />
33     <p>Valore con debounce: {debouncedValue}</p>
34   </div>
35 );
36 }
```

# useMemo e useCallback

## Concetto di memorizzazione in React

- **Memorizzazione:** tecnica per memorizzare i risultati di operazioni costose
- Evita calcoli ripetuti quando gli input non cambiano
- Ottimizzazione delle performance, non una feature essenziale



# useMemo e useCallback

memorizzazione di valori calcolati

```
1 function ExpensiveCalculation({ numbers }) {
2   // Senza useMemo - ricalcola ad ogni render
3   // const total = numbers.reduce((acc, num) => acc + num, 0);
4
5   // Con useMemo - ricalcola solo quando 'numbers' cambia
6   const total = useMemo(() => {
7     console.log("Calcolo costoso in esecuzione...");
8     return numbers.reduce((acc, num) => acc + num, 0);
9   }, [numbers]);
10
11  return <div>Totale: {total}</div>;
12 }
```



# useMemo e useCallback

useCallback: memorizzazione di funzioni

```
function ParentComponent() {
  const [count, setCount] = useState(0);
  const [otherState, setOtherState] = useState(0);
  ...
  // Senza useCallback - crea una nuova funzione ad ogni render
  // const handleClick = () => {
  //   console.log("Clicked!", count);
  // };
  ...
  // Con useCallback - crea una nuova funzione solo quando 'count' cambia
  const handleClick = useCallback(() => {
    console.log("Clicked!", count);
  }, [count]);
  ...
  return (
    <div>
      <ChildComponent onClick={handleClick} />
      <button onClick={() => setOtherState(otherState + 1)}>
        Aggiorna altro stato ({otherState})
      </button>
    </div>
  );
}
...
// React.memo previene re-render se le props non cambiano
const ChildComponent = React.memo(function ChildComponent({ onClick }) {
  console.log("Child component rendered");
  return <button onClick={onClick}>Clicca</button>;
});
```



# useMemo e useCallback

Quando è realmente necessario utilizzarli

- **Usa useMemo quando:**
  - Hai calcoli costosi ( $O(n^2)$  o peggio)
  - Crei oggetti di grandi dimensioni
  - Hai un componente che renderizza spesso
- **Usa useCallback quando:**
  - Passi funzioni a componenti ottimizzati con `React.memo`
  - Funzioni utilizzate come dipendenze di altri hooks
  - Vuoi mantenere l'identità di una funzione tra i render



# useMemo e useCallback

Relazione con il rendering e le performance

```
...  
1 function SearchResults({ query, data }) {  
2   // Memorizza i risultati filtrati  
3   const filteredResults = useMemo(() => {  
4     console.log("Filtraggio dati...");  
5     return data.filter(item =>  
6       item.name.toLowerCase().includes(query.toLowerCase())  
7     );  
8   }, [data, query]);  
9  
10  // Memorizza una funzione per ordinare i risultati  
11  const sortResults = useCallback((a, b) => {  
12    return a.name.localeCompare(b.name);  
13  }, []);  
14  
15  return (  
16    <div>  
17      <p>Trovati {filteredResults.length} risultati per "{query}"</p>  
18      <ResultList  
19        results={filteredResults}  
20        sortFunction={sortResults}  
21      />  
22    </div>  
23  );  
24}  
25  
26 const ResultList = React.memo(function ResultList({ results, sortFunction }) {  
27   // Questo componente non si ri-renderizza se results e sortFunction  
28   // non cambiano, anche se il genitore si renderizza  
29   const sortedResults = [...results].sort(sortFunction);  
30  
31   return (  
32     <ul>  
33       {sortedResults.map(item => (  
34         <li key={item.id}>{item.name}</li>  
35       ))}  
36     </ul>  
37   );  
38 });
```



# useMemo e useCallback

## Trappole comuni e pattern efficaci

### Trappole

- **Over-optimization:** Utilizzare useMemo/useCallback ovunque
- **Dipendenze mancanti:** Non includere tutte le dipendenze
- **Creare dipendenze non necessarie:** Passare oggetti inline

```
1 // ✖ Problematico: options è un nuovo oggetto ad ogni render
2 function SearchComponent() {
3   const [query, setQuery] = useState("");
4
5   // Oggetto creato a ogni render
6   const options = {
7     caseSensitive: false,
8     matchWholeWord: true
9   };
10
11 // useEffect si esegue a ogni render poiché options è sempre nuovo
12 useEffect(() => {
13   performSearch(query, options);
14 }, [query, options]);
15
16 return <input value={query} onChange={e => setQuery(e.target.value)} />;
17 }
```

# useMemo e useCallback

## Pattern efficaci

```
1 // ✅ Corretto: options è memorizzato
2 function SearchComponent() {
3   const [query, setQuery] = useState("");
4   const [caseSensitive, setCaseSensitive] = useState(false);
5
6   // Memorizza l'oggetto options
7   const options = useMemo(() => ({
8     caseSensitive,
9     matchWholeWord: true
10 }), [caseSensitive]);
11
12 // useEffect si esegue solo quando query o caseSensitive cambiano
13 useEffect(() => {
14   performSearch(query, options);
15 }, [query, options]);
16
17 return (
18   <div>
19     <input value={query} onChange={e => setQuery(e.target.value)} />
20     <label>
21       <input
22         type="checkbox"
23         checked={caseSensitive}
24         onChange={e => setCaseSensitive(e.target.checked)}
25       />
26       Case sensitive
27     </label>
28   </div>
29 );
30 }
```

# Esercizio 2: Applicazione per Previsioni Meteo

## Setup del Progetto

- Creazione della struttura base
- Installazione di dipendenze (axios per le chiamate API)
- Spiegazione dell'API meteo che utilizzeremo (OpenWeatherMap)



```
1 weather-app/
2   public/
3     index.html
4     favicon.ico
5   src/
6     components/
7       WeatherApp.jsx
8       SearchBar.jsx
9       WeatherDisplay.jsx
10      WeatherCard.jsx
11      RecentSearches.jsx
12      ErrorMessage.jsx
13    hooks/
14      useWeatherApi.js
15      useLocalStorage.js
16    utils/
17      api.js
18      formatters.js
19    App.jsx
20    index.jsx
21    App.css
22  package.json
23  README.md
```

# Riepilogo della Lezione

## Punti chiave:

- Gli hooks devono essere chiamati solo al livello superiore di un componente o di altri hooks
- Le dipendenze degli hooks devono essere gestite correttamente per evitare problemi come loop infiniti o stale closures
- La memorizzazione deve essere applicata strategicamente, non in eccesso
- Usefulness di useRef per valori che non dovrebbero causare un ri-render
- Importanza dell'immutabilità nell'aggiornamento dello stato

## Applicazioni pratiche:

Abbiamo implementato un'applicazione meteo con:

- Chiamate API ottimizzate
- Sistema di cache per migliorare le performance
- Gestione intelligente delle ricerche recenti
- Calcoli derivati efficienti con useMemo
- Funzioni stabilizzate con useCallback



# Esercizi Pratici per Consolidare gli Hooks

## Esercizio 1: Cache avanzata

Estendi il sistema di cache implementando:

- Impostazione del tempo di scadenza della cache da parte dell'utente
- Pulsante per aggiornare manualmente i dati meteo
- Indicatore visuale dell'età dei dati (verde se recenti, giallo se più vecchi di 10 minuti, rosso se più vecchi di 25 minuti)

## Esercizio 2: Localizzazione con custom hook

Crea un custom hook `useGeoLocation` che:

- Chiede il permesso per accedere alla posizione dell'utente
- Ottiene le coordinate tramite l'API Geolocation del browser
- Le converte in una città utilizzando l'API di geocoding inverso
- Implementa la corretta gestione degli errori e dello stato di caricamento



# Esercizi Pratici per Consolidare gli Hooks

## Esercizio 3: Grafico delle temperature con useRef

Aggiungi un grafico che mostri l'andamento delle temperature durante la giornata utilizzando:

- Canvas HTML con useRef per accedere al contesto di disegno
- useEffect per disegnare il grafico quando i dati cambiano
- useMemo per calcolare i valori massimi e minimi delle scale

## Esercizio 4: Dark mode con Context e useReducer

Implementa un sistema di tema (dark/light mode) utilizzando:

- Context API per fornire il tema in tutta l'app
- useReducer per gestire le azioni di cambio tema
- useEffect per persistere la preferenza nel localStorage
- CSS variables per applicare i colori del tema corrente
- 





# Modulo 3

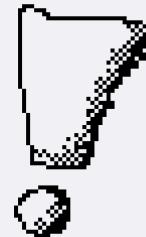
## lo Stato Globale

# Modulo 2: Hooks in Profondità

## Workshop Pratico: E-commerce Mini-Cart

**Teoria:**

1. Context API
2. useReducer
3. Pattern di gestione dello stato



# Context API

## Problema della prop drilling e soluzioni

- Prop drilling: quando si passano props attraverso componenti che non ne hanno bisogno
- // App → Componente1 → Componente2 → Componente3 (usa la prop)

Conseguenze: codice meno mantenibile, accoppiamento elevato, refactoring più complesso

### Possibili soluzioni:

- Composition pattern (render props, children)
- Higher-Order Components (HOC)
- Context API ← focus di oggi



# Cos'è il Context API e quando usarlo

**Definizione:** meccanismo per condividere dati tra componenti senza passarli manualmente attraverso le props

**Quando usarlo:**

- Per dati considerati "globali" (tema, utente autenticato, preferenze)
- Quando molti componenti hanno bisogno degli stessi dati
- Per evitare il prop drilling con dati che cambiano raramente

**Quando NON usare Context:**

- Per ogni comunicazione tra componenti (props restano lo strumento principale)
- Per stati che cambiano frequentemente (può causare re-render non necessari)
- Per logica complessa di gestione dello stato (potrebbe essere più appropriato Redux)



# Cos'è il Context API e quando usarlo

## `createContext`, `Provider` e `useContext`

- `createContext`: crea un oggetto Context
- `Provider`: componente che fornisce il valore ai componenti figli
- `useContext`: hook per accedere al valore del Context

```
1 // 1. Creazione del Context (ThemeContext.js)
2 import { createContext, useState } from 'react';
3
4 // Valore di default quando un componente consuma il Context al di fuori di un Provider
5 const ThemeContext = React.createContext('light');
6
7 // 2. Provider del Context
8 <ThemeContext.Provider value="dark">
9   <App />
10 </ThemeContext.Provider>
11
12 // 3. Consumo del Context in un componente
13 const theme = useContext(ThemeContext);
```



# Cos'è il Context API e quando usarlo

## Pattern di organizzazione dei context

### Separazione dei context:

```
1 // Più context per differenti tipologie di dati
2 const UserContext = createContext();
3 const ThemeContext = createContext();
```

### Context isolati:

```
1 // /contexts/ThemeContext.js
2 export const ThemeContext = createContext();
3 export const ThemeProvider = ({ children }) => {
4   const [theme, setTheme] = useState('light');
5   return (
6     <ThemeContext.Provider value={{ theme, setTheme }}>
7       {children}
8     </ThemeContext.Provider>
9   );
10};
```

### Custom hook per contesto:

```
1 // hook personalizzato per usare il context
2 export const useTheme = () => useContext(ThemeContext);
```



# Cos'è il Context API e quando usarlo

## Considerazioni sulle performance

- Context provoca re-render di tutti i componenti che lo consumano quando il valore cambia
- Per ottimizzare:
  - Dividere i context per dominio/tipologia di dati
  - Usare la memorizzazione (useMemo) per i valori del provider
  - Utilizzare React.memo sui componenti consumer

```
1 const MemoizedComponent = React.memo(MyComponent);
```



# Cos'è il Context API e quando usarlo

## Limiti del Context API

- Non ottimizzato per aggiornamenti frequenti (es. gestione stato globale con molte mutazioni)
- Non offre nativamente funzionalità di middleware o time-travel debugging
- Complessità aumenta con l'aumentare dei provider annidati
- Non sostituisce completamente una libreria di state management per casi complessi



# useReducer

Concetto di reducer: `(state, action) ⇒ newState`

- **Definizione:** funzione pura che accetta lo stato corrente e un'azione, restituisce un nuovo stato
- **Paradigma:** ispirato a Redux e alla programmazione funzionale

```
● ● ●  
1 function reducer(state, action) {  
2   // Restituisce il nuovo stato in base all'azione  
3   return newState;  
4 }
```



# useReducer

## Vantaggi rispetto a useState per logica complessa

- Prevedibilità: le modifiche di stato seguono sempre lo stesso pattern
- Debugging: più facile tracciare le modifiche di stato
- Separazione: la logica di aggiornamento è separata dai componenti
- Coerenza: garantisce aggiornamenti atomici per stati correlati
- Testing: funzioni pure facili da testare

## Gestione centralizzata delle modifiche di stato

- Uso base:

```
1 function reducer(state, action) {  
2   switch (action.type) {  
3     case 'increment':  
4       return { count: state.count + 1 };  
5     case 'decrement':  
6       return { count: state.count - 1 };  
7     default:  
8       throw new Error('Azione non supportata');  
9   }  
10 }
```



```
1 const initialState = { count: 0 };  
2 const [state, dispatch] = useReducer(reducer, initialState);  
3  
4 // Per modificare lo stato  
5 dispatch({ type: 'increment' });
```

# useReducer

Struttura delle azioni: tipo e payload

Convenzione standard:

```
{ type: 'ACTION_TYPE', payload: anyData }
```

Esempio:

```
1 jsxCopydispatch({
2   type: 'add_todo',
3   payload: { id: Date.now(), text: 'Nuovo todo', completed: false }
4 });
5
6 Action creators (funzioni che generano azioni):
7 jsxCopyconst addTodo = (text) => ({
8   type: 'add_todo',
9   payload: { id: Date.now(), text, completed: false }
10});
```



Action creators (funzioni che generano azioni):

```
1 const addTodo = (text) => ({
2   type: 'add_todo',
3   payload: { id: Date.now(), text, completed: false }
4 });
```

# useReducer

Pattern per l'immutabilità:

- **Spread operator** per oggetti e array

```
1 case 'update_todo':  
2   return {  
3     ...state,  
4     todos: state.todos.map(todo =>  
5       todo.id === action.payload.id  
6         ? { ...todo, ...action.payload }  
7         : todo  
8     )  
9   };
```



# useReducer

## Dispatcher e dispatch function

- **Dispatch function:** l'API per inviare azioni al reducer
- **Dispatcher pattern:** astrarre la chiamata a dispatch

```
 1 function TodoList() {
 2   const [state, dispatch] = useReducer(todoReducer, initialState);
 3
 4   const addNewTodo = (text) => {
 5     dispatch({ type: 'add_todo', payload: { text } });
 6   };
 7
 8   return (
 9     <div>
10       <button onClick={() => addNewTodo('Nuovo task')}>
11         Aggiungi
12       </button>
13     </div>
14   );
15 }
```

# Pattern di gestione dello stato

## Flux vs MVC

- **MVC (Model-View-Controller):**
  - Modello tradizionale con flusso bidirezionale
  - Potenziali problemi di sincronizzazione con stati complessi
- **Flux:**
  - Flusso unidirezionale dei dati: Action → Dispatcher → Store → View
  - Maggiore prevedibilità e manutenibilità
  - Redux e Context+useReducer seguono questo pattern

## Stato globale vs stato locale

- **Stato locale:**
  - Limitato a un componente o a un sottoalbero
  - Implementato con `useState` o `useReducer`
  - Più semplice da gestire e ottimizzare
- **Stato globale:**
  - Accessibile a tutta l'applicazione
  - Gestito con Context API, Redux, o altre librerie
  - Necessario per dati condivisi tra componenti distanti



# Pattern di gestione dello stato

Quando combinare Context e Reducer:

- Context + useReducer: pattern potente per lo stato globale

Vantaggi:

- Flusso dati unidirezionale
- Implementazione Flux senza librerie esterne
- Buon equilibrio tra performance e semplicità

```
1 const AppStateContext = createContext();
2
3 function AppProvider({ children }) {
4   const [state, dispatch] = useReducer(appReducer, initialState);
5
6   return (
7     <AppStateContext.Provider value={{ state, dispatch }}>
8       {children}
9     </AppStateContext.Provider>
10   );
11 }
```

# Pattern di gestione dello stato

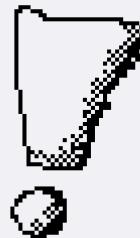
## Architettura a strati per applicazioni complesse

- **Domain-driven design:**
  1. Separare lo stato per dominio (auth, prodotti, carrello, ecc.)
  2. Context multipli con responsabilità specifiche
- **Strati consigliati:**
  1. **Presentation layer:** componenti UI
  2. **Application layer:** context, reducer, state
  3. **Domain layer:** logica di business, modelli
  4. **Infrastructure layer:** API, storage, servizi esterni
- **Colocazione dello stato:** posizionare lo stato il più vicino possibile a dove viene utilizzato



# Conclusione

- Context API e useReducer sono strumenti potenti integrati in React
- Per applicazioni semplici/medie: Context + useReducer è spesso sufficiente
- Per applicazioni complesse: considerare librerie specializzate
- Combinare approcci diversi per diverse parti dell'applicazione
- Privilegiare la colocatione dello stato quando possibile



# Conclusione

## Concetti Chiave

In questo modulo abbiamo:

1. **Esplorato il Context API** per gestire lo stato globale in modo efficiente
2. **Implementato il pattern Reducer** per gestire modifiche complesse allo stato
3. **Creato hook personalizzati** per semplificare l'accesso al Context
4. **Implementato la persistenza dello stato** con localStorage
5. **Sviluppato un'interfaccia completa** per un mini-carrello di e-commerce

## Vantaggi dell'architettura implementata

- **Separazione delle responsabilità:** Context gestisce l'accesso allo stato, Reducer gestisce le modifiche
- **Centralizzazione della logica:** Tutte le modifiche allo stato passano attraverso il Reducer
- **Riutilizzabilità:** Gli hook personalizzati rendono l'accesso ai dati semplice in qualsiasi componente
- **Manutenibilità:** Struttura chiara che facilita l'aggiunta di nuove funzionalità
- **Persistenza dei dati:** L'utente non perde i dati del carrello al refresh



## Esercizio 3: E-commerce Mini-Cart

- Context API e useReducer sono strumenti potenti integrati in React
- Per applicazioni semplici/medie: Context + useReducer è spesso sufficiente
- Per applicazioni complesse: considerare librerie specializzate
- Combinare approcci diversi per diverse parti dell'applicazione
- Privilegiare la collocazione dello stato quando possibile



— [www.Duccio.ME](http://www.Duccio.ME) —

# Esercizio 3: E-commerce Mini-Cart

## Esercizi Extra:

1. Aggiungere una lista di desideri:
  - o Creare un Context e Reducer per la wishlist
  - o Implementare funzionalità per aggiungere/rimuovere dalla wishlist
  - o Aggiungere un pulsante "Aggiungi alla wishlist" nelle card dei prodotti
2. Implementare notifiche toast:
  - o Creare un Context per le notifiche
  - o Mostrare toast quando un prodotto viene aggiunto/rimosso dal carrello
  - o Implementare un sistema di coda per le notifiche
3. Aggiungere un sistema di autenticazione:
  - o Implementare un Context per l'autenticazione
  - o Creare pagine di login/signup
  - o Collegare il carrello all'utente autenticato
4. Estendere il sistema di checkout:
  - o Aggiungere un form per i dati di spedizione
  - o Implementare la validazione dei dati
  - o Simulare il processo di pagamento
5. Implementare filtri per i prodotti:
  - o Creare un Context per i filtri
  - o Aggiungere filtri per categoria, prezzo, valutazione
  - o Implementare la logica di filtro nella visualizzazione dei prodotti

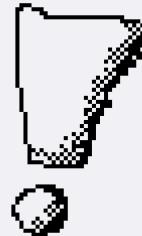


# Modulo 4

Routing e Navigazione

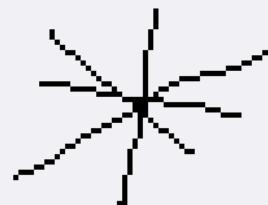
# Modulo 4: React Router v6

Workshop Pratico: Blog con Autenticazione



Teoria:

1. Introduzione a React Router v6
2. I componenti fondamentali: BrowserRouter, Routes, Route, Outlet, Link e NavLink, Navigate
3. Caratteristiche avanzate: Route nesting, Route parametriche, Protected routes



— [www.Duccio.ME](http://www.Duccio.ME) —

# Introduzione a React Router v6

## Cos'è React Router?

- Libreria standard de facto per il routing nelle applicazioni React
- Permette la navigazione tra diverse "pagine" senza ricaricare il browser
- Crea applicazioni Single Page Application (SPA) con URL distinti
- Mantiene UI e URL sincronizzati

## Cosa offre la versione 6?

- API completamente riprogettata rispetto alla v5
- Sintassi più semplice e dichiarativa
- Migliore gestione dei layout annidati
- Supporto nativo per i data loaders (v6.4+)
- Performance migliorate



# Introduzione a React Router v6

## Installazione

```
1 # npm  
2 npm install react-router-dom  
3  
4 # yarn  
5 yarn add react-router-dom
```



# Introduzione a React Router v6

## Filosofia principale di React Router v6

- **Routing dichiarativo:** definisci cosa deve essere visualizzato per ciascun percorso
- **Composizione:** facilità di combinare e annidare route
- **UI e URL sincronizzati:** l'interfaccia utente riflette sempre l'URL corrente
- **Hooks-based API:** utilizzo di React Hooks per operazioni di routing



# I componenti fondamentali

## BrowserRouter

- Componente wrapper che abilita il routing nell'applicazione
- Utilizza l'API History del browser per gestire la navigazione
- Va posizionato alla radice dell'applicazione

```
1 import { BrowserRouter } from 'react-router-dom';
2
3 function App() {
4   return (
5     <BrowserRouter>
6       {/* Il resto dell'applicazione */}
7     </BrowserRouter>
8   );
9 }
```



# I componenti fondamentali

## Routes

- Container per i componenti Route
- Confronta tutti i suoi elementi Route figli con l'URL corrente
- Renderizza solo la prima Route che corrisponde
- Sostituisce il vecchio `<Switch>` della v5

```
1 import { Routes, Route } from 'react-router-dom';
2
3 function App() {
4   return (
5     <BrowserRouter>
6       <Routes>
7         <Route path="/" element={<Home />} />
8         <Route path="/about" element={<About />} />
9         <Route path="/contact" element={<Contact />} />
10      </Routes>
11    </BrowserRouter>
12  );
13 }
```

# I componenti fondamentali

## Route

- Associa un percorso URL a un componente React
- Attributi principali:
  - `path`: il pattern dell'URL da abbinare
  - `element`: il componente React da renderizzare
  - `index`: indica una route predefinita per un percorso parent



```
1 <Route path="/blog" element={<BlogLayout />}>
2   {/* Route index renderizzata quando il path è esattamente '/blog' */}
3   <Route index element={<BlogOverview />} />
4   <Route path=":id" element={<BlogPost />} />
5 </Route>
```



# I componenti fondamentali

## Outlet

- Funziona come un "segnaposto" dove verranno renderizzati i componenti figli
- Essenziale per implementare layout annidati
- Simile al concetto di `{children}` ma specifico per il routing

```
1 // In BlogLayout.js
2 import { Outlet } from 'react-router-dom';
3
4 function BlogLayout() {
5   return (
6     <div>
7       <h1>Blog</h1>
8       <nav>
9         <ul>
10           <li><Link to="/blog/latest">Ultimi post</Link></li>
11           <li><Link to="/blog/popular">Post popolari</Link></li>
12         </ul>
13       </nav>
14
15     {/* Qui verrà renderizzato il componente figlio */}
16     <Outlet />
17   </div>
18 );
19 }
```

# I componenti fondamentali

## Link e NavLink

- **Link:** componente per navigare tra le pagine senza ricaricare il browser
- **NavLink:** estensione di Link che aggiunge stili quando il link è attivo

```
1 import { Link, NavLink } from 'react-router-dom';
2
3 function Navigation() {
4   return (
5     <nav>
6       /* Link base */
7       <Link to="/home">Home</Link>
8
9       /* NavLink con stile per lo stato attivo */
10      <NavLink
11        to="/dashboard"
12        style={({ isActive }) => ({
13          fontWeight: isActive ? 'bold' : 'normal',
14          color: isActive ? 'red' : 'black',
15        })}
16      >
17        Dashboard
18      </NavLink>
19
20      /* NavLink con classe per lo stato attivo */
21      <NavLink
22        to="/profile"
23        className={({ isActive }) =>
24          isActive ? 'active-link' : 'normal-link'
25        }
26      >
27        Profilo
28      </NavLink>
29    </nav>
30  );
31}
```



# I componenti fondamentali

## Navigate

- Componente per reindirizzamenti dichiarativi
- Utile per reindirizzamenti basati su condizioni

```
1 import { Navigate } from 'react-router-dom';
2
3 function RedirectPage() {
4   const isLoggedIn = checkAuthStatus();
5
6   // Reindirizza se l'utente non è autenticato
7   if (!isLoggedIn) {
8     return <Navigate to="/login" replace />;
9   }
10
11  return <ProtectedContent />;
12 }
```



# Caratteristiche avanzate

## Route nesting (Route annidate)

- Consente di creare layout gerarchici
- Le route figlie ereditano il percorso del genitore
- Perfetto per layout condivisi e UI complesse

```
1 <Routes>
2   <Route path="/" element={<Layout />}>
3     <Route index element={<Home />} />
4     <Route path="dashboard" element={<Dashboard />}>
5       <Route index element={<DashboardOverview />} />
6       <Route path="stats" element={<DashboardStats />} />
7       <Route path="settings" element={<DashboardSettings />} />
8     </Route>
9     <Route path="profile" element={<Profile />} />
10    </Route>
11 </Routes>
```

# Caratteristiche avanzate

## Route parametriche

- Cattura parti dinamiche dell'URL
- Utilizza la sintassi `:nome` nel pattern del path
- Accedi ai parametri con l'hook `useParams`

```
1 // Definizione della route
2 <Route path="/products/:productId" element={<ProductDetail />} />
3
4 // Componente ProductDetail
5 import { useParams } from 'react-router-dom';
6
7 function ProductDetail() {
8   // Estrae il parametro dall'URL
9   const { productId } = useParams();
10
11  return (
12    <div>
13      <h1>Dettaglio Prodotto</h1>
14      <p>Stai visualizzando il prodotto ID: {productId}</p>
15    </div>
16  );
17 }
```

# Caratteristiche avanzate

## Protected routes

- Route accessibili solo a utenti autenticati
- Pattern comune per aree riservate dell'applicazione

```
1 // Componente ProtectedRoute
2 function ProtectedRoute({ children }) {
3   const isAuthenticated = useAuth(); // Custom hook per verificare l'autenticazione
4
5   if (!isAuthenticated) {
6     // Reindirizza al login se non autenticato
7     return <Navigate to="/login" replace />;
8   }
9
10  return children;
11 }
12
13 // Utilizzo nelle routes
14 <Routes>
15   <Route path="/" element={<Layout />}>
16     <Route index element={<Home />} />
17     <Route path="login" element={<Login />} />
18
19   {/* Area protetta */}
20   <Route
21     path="admin"
22     element={
23       <ProtectedRoute>
24         <AdminLayout />
25       </ProtectedRoute>
26     }
27   >
28     <Route index element={<AdminDashboard />} />
29     <Route path="users" element={<UserManagement />} />
30   </Route>
31 </Route>
32 </Routes>
```



# React Router Hooks

## useParams

- Accede ai parametri dinamici dell'URL
- Restituisce un oggetto con tutti i parametri della route corrente

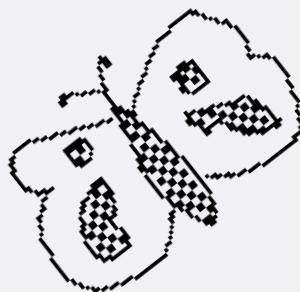
```
1 import { useParams } from 'react-router-dom';
2
3 function BlogPost() {
4   const { postId, categoryId } = useParams();
5
6   return (
7     <div>
8       <h1>Blog Post: {postId}</h1>
9       <p>Categoria: {categoryId}</p>
10    </div>
11  );
12}
13
14 // Usato con una route simile a:
15 // ...
16 <Route path="/blog/:categoryId/:postId" element={<BlogPost />} />
```



# React Router Hooks

## useNavigate

- Permette la navigazione programmatica
- Sostituisce il pattern `history.push()` delle versioni precedenti



```
1 import { useNavigate } from 'react-router-dom';
2
3 function LoginForm() {
4   const navigate = useNavigate();
5
6   const handleSubmit = async (e) => {
7     e.preventDefault();
8
9     // Logica di login...
10    const success = await loginUser(formData);
11
12    if (success) {
13      // Reindirizza dopo il login
14      navigate('/dashboard');
15
16      // Con opzioni
17      // navigate('/dashboard', { replace: true });
18      // Sostituisce l'entry nello stack di navigazione
19    }
20  };
21
22  return (
23    <form onSubmit={handleSubmit}>
24      {/* Form inputs */}
25    </form>
26  );
27 }
```

# React Router Hooks

## useLocation

- Accede alle informazioni sulla posizione corrente
- Utile per accedere a query params e stato di navigazione

```
1 import { useLocation } from 'react-router-dom';
2
3 function SearchResults() {
4   const location = useLocation();
5
6   // Estrae i parametri di query
7   const queryParams = new URLSearchParams(location.search);
8   const query = queryParams.get('q');
9
10  // Accede allo stato passato durante la navigazione
11  const { from } = location.state || { from: 'unknown' };
12
13  return (
14    <div>
15      <h1>Risultati di ricerca per: {query}</h1>
16      <p>Ricerca iniziata da: {from}</p>
17    </div>
18  );
19 }
20
21 // Navigazione con query params e stato
22 // ...
23 navigate('/search?q=react', { state: { from: 'homepage' } });


```

— <http://www.Duccio.ME> —

# React Router Hooks

## Considerazioni sul Routing Lazy

- Permette di caricare i componenti solo quando necessario (code splitting)
  - Migliora le prestazioni delle applicazioni complesse

```
1 import { lazy, Suspense } from 'react';
2 import { Route, Routes } from 'react-router-dom';
3
4 // Importazione lazy dei componenti
5 const Home = lazy(() => import('./pages/Home'));
6 const Dashboard = lazy(() => import('./pages/Dashboard'));
7 const Settings = lazy(() => import('./pages/Settings'));
8
9 function App() {
10   return (
11     <BrowserRouter>
12       <Suspense fallback={<div>Loading...</div>}>
13         <Routes>
14           <Route path="/" element={<Home />} />
15           <Route path="/dashboard" element={<Dashboard />} />
16           <Route path="/settings" element={<Settings />} />
17         </Routes>
18       </Suspense>
19     </BrowserRouter>
20   );
21 }
```

# Conclusione

## Best Practices

- Organizza le route in modo gerarchico e intuitivo
- Utilizza route annidate per layout condivisi
- Implementa la protezione delle route sensibili
- Utilizza la lazy loading per migliorare le performance
- Mantieni una struttura chiara tra URL e UI

## Evoluzione di React Router

- React Router continua ad evolversi con nuove funzionalità
- Le versioni più recenti (6.4+) includono data loaders e azioni
- Sempre più integrazione con le funzionalità moderne di React

## Risorse utili

- Documentazione ufficiale: [reactrouter.com](https://reactrouter.com)



# Esercizio 4: Blog con Autenticazione

## Fase 1: Setup del progetto e struttura iniziale

- Creeremo un'applicazione blog con:
  - Area pubblica (home, lista post, dettaglio post)
  - Area admin protetta (dashboard, gestione post)
  - Autenticazione
- Struttura cartelle del progetto:

```
1 /src
2   /components      # Componenti riutilizzabili
3   /context         # Context per l'autenticazione
4   /layouts          # Layout condivisi
5   /pages            # Pagine dell'applicazione
6     /admin          # Pagine area admin
7   /utils            # Utility varie
```

# Esercizio 4: Blog con Autenticazione

Fase 2: Implementazione del routing base

1. Installiamo React Router:

```
> npm install react-router-dom
```

2. Creiamo la struttura delle route in `App.js` con due layout diversi:

- `MainLayout`: per l'area pubblica
- `AdminLayout`: per l'area protetta

3. Implementiamo route annidate per organizzare meglio il codice e condividere layout



# Esercizio 4: Blog con Autenticazione

## Fase 3: Creazione del Context per l'autenticazione

1. Implementiamo `AuthContext.js` per gestire:
  - o Login/logout
  - o Stato di autenticazione
  - o Persistenza in localStorage
2. Creiamo il componente `RequireAuth` per proteggere le route admin

## Fase 4: Implementazione dei layout condivisi

1. Creiamo `MainLayout.js` e `AdminLayout.js` con:
  - o Header con navigazione
  - o Footer (solo layout principale)
  - o Spazio per i componenti figli (Outlet)



# Esercizio 4: Blog con Autenticazione

## Fase 5: Pagine di autenticazione e area pubblica

1. Implementiamo la pagina di login con:
  - o Form per username/password
  - o Redirect alla pagina originariamente richiesta
  - o Gestione errori
2. Creiamo le pagine pubbliche:
  - o Home
  - o Lista post
  - o Dettaglio post



## Fase 6: Implementazione dell'area admin protetta

1. Creiamo le pagine admin:
  - o Dashboard
  - o Lista di gestione post
  - o Modifica post



— [www.Duccio.ME](http://www.Duccio.ME) —

# Esercizio 4: Blog con Autenticazione

## Fase 7: Navigazione programmatica e dinamica

1. Implementiamo esempi di navigazione programmatica con `useNavigate`:
  - o Bottoni "Torna indietro"
  - o Redirect dopo operazioni (es. dopo salvataggio post)
2. Utilizziamo i parametri URL per le pagine dinamiche:
  - o Dettaglio post (`/blog/:postId`)
  - o Modifica post (`/admin/posts/:postId/edit`)

## Fase 8: Implementazione dei breadcrumbs dinamici

1. Creiamo un componente `Breadcrumbs` che:
  - o Analizza l'URL corrente con `useLocation`
  - o Genera link gerarchici in base alla struttura dell'URL
  - o Gestisce i parametri dinamici in modo user-friendly



# Esercizio 4: Blog con Autenticazione

## Esercizi EXTRA

1. **Esercizio base:** Aggiungere una nuova pagina pubblica "Chi siamo" con relativo link nella navigazione.
2. **Esercizio intermedio:** Implementare una funzionalità di ricerca che utilizzi i query parameters nell'URL (es. `/blog?search=react`).
3. **Esercizio avanzato:** Aggiungere un sistema di ruoli (admin/editor) e limitare alcune funzionalità solo agli admin.

## Considerazioni finali e best practices

- Strutturare le route in modo gerarchico e logico
- Utilizzare layout condivisi per evitare duplicazione di codice
- Implementare una buona gestione degli errori (route non trovate, accessi non autorizzati)

• Considerare sempre l'esperienza utente nella navigazione



— WWW.DUCKTALE.ME —