

Umberto Emanuele

Redux – Lezione 1

Gennaio 2023


2



Installazione

Redux è disponibile come pacchetto su NPM per l'uso con un bundle di moduli o in un'applicazione Node:

```
npm install --save redux
```

 **copia**

<https://redux.js.org/>

Cos'è Redux ?

Redux è un potente strumento per gestire gli **stati** all'interno di un'applicazione.

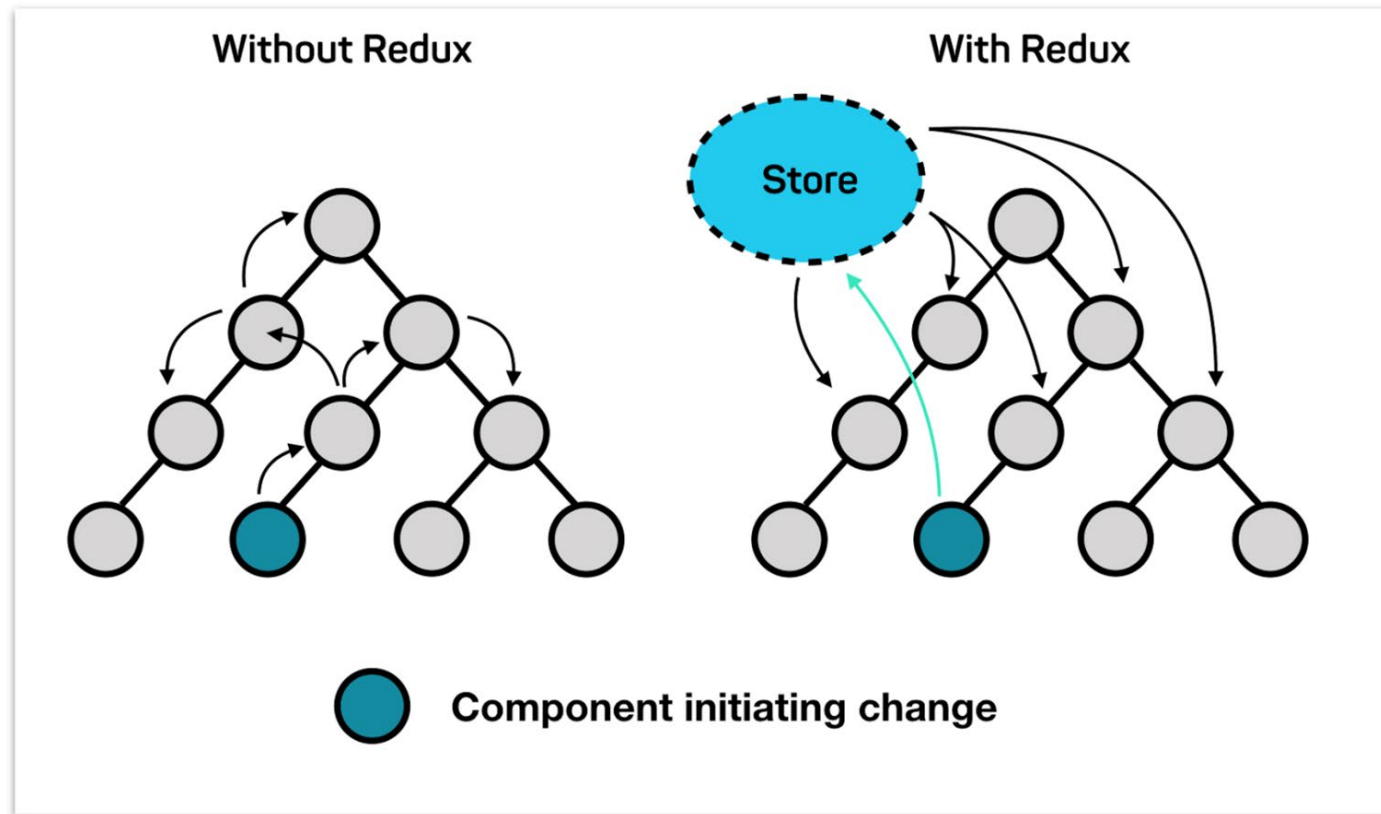
Puoi usare **Redux** insieme a **React** o con qualsiasi altra libreria. È piccolo (2kB, comprese le dipendenze), ma ha un ampio ecosistema di componenti aggiuntivi disponibili.

<https://redux.js.org/introduction/getting-started#examples>

Fino ad adesso, per condividere uno stato tra più componenti, l'unica soluzione era elevare lo stato fino a raggiungere il componente padre di tutti quelli che si desiderava “**collegare**”.

Questo può portare a uno scambio di **props** a volte non necessario, ed il risultato è codice inutilmente complicato.

Ora lo stato dell'applicativo può essere unificato e condiviso in modo nativo attraverso componenti differenti, permettendo un accesso diretto quando e dove necessario (anche solamente per sotto-sezioni dello stato)



Redux può essere descritto in tre principi fondamentali:

Singolo Sorgente di verità

E' possibile accedere ai dati da qualsiasi punto.

Lo stato dell'intera applicazione viene archiviato in una struttura ad oggetti all'interno di un singolo archivio (store).

Lo stato diventa di sola lettura

L'unico modo per cambiare lo stato è emettere un'azione

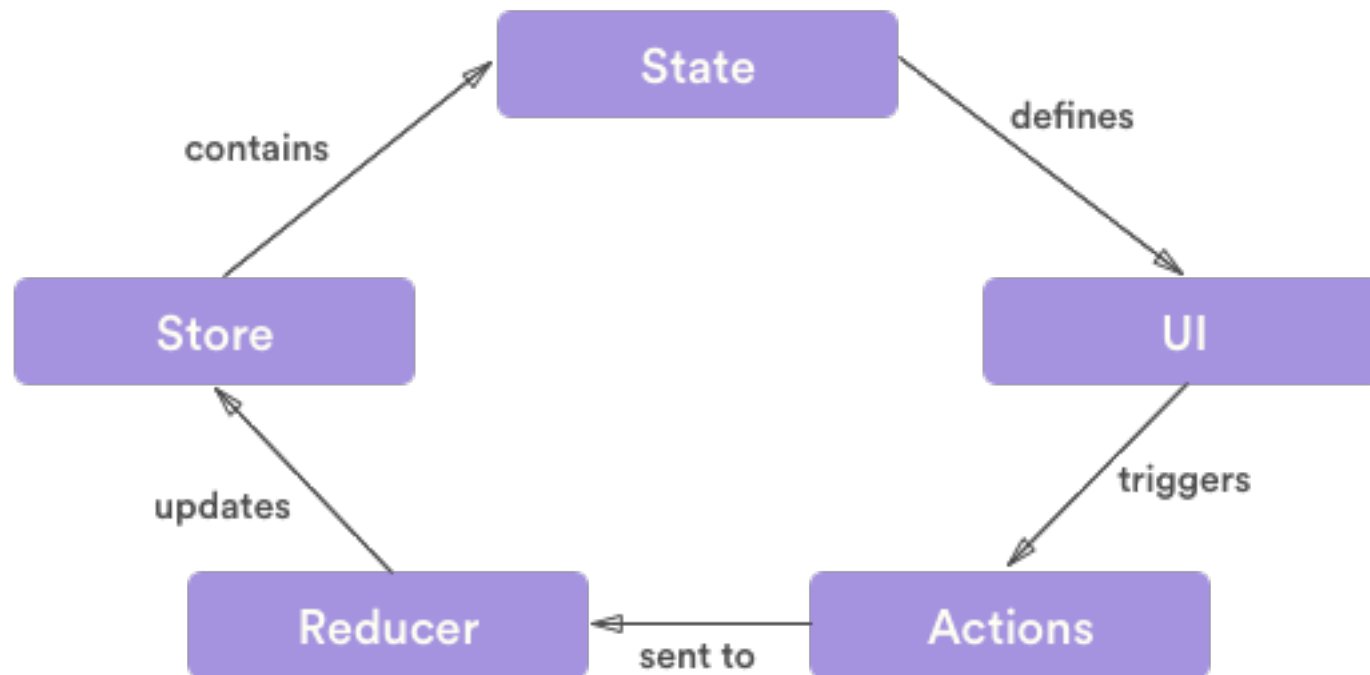
Le modifiche vengono apportate tramite reducers

Redux Store vs Stato dei Componenti

Se inseriamo **Redux** per gestire meglio i dati nostra applicazione è possibile continuare ad utilizzare e gestire lo **stato dei componenti**. Siamo noi a decidere quali stati gestire con Redux.

Usiamo lo **stato React** in tutti quegli stati locali al componente. Es. lo stato che gestisce l'input di un Form. Utilizzare lo store Redux farebbe aumentare la complessità della nostra applicazione.

Usiamo **Redux** per stati condivisi dall'intera applicazione e nei casi in cui dobbiamo passare dei dati in complessi componenti annidati.



Si basa sul concetto di avere un unico **stato**, rappresentato da un oggetto JSON e conservato in uno **store**. Questo stato può mutare solo in seguito ad **azioni**, ma non direttamente, la modifica avviene tramite l'invocazione di una funzione pura denominata **reducer**.

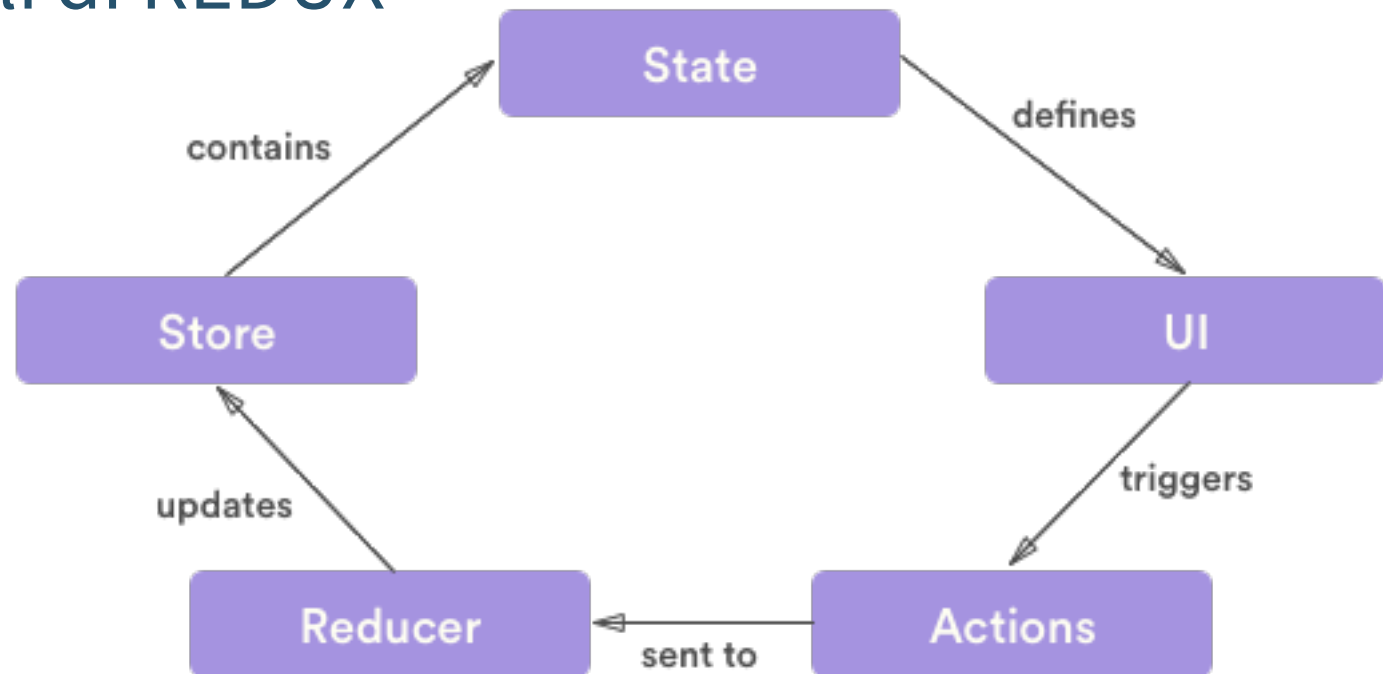
Cosa sono le Pure Function

- Restituiscono un solo valore, sempre lo stesso se gli argomenti passati sono gli stessi.
- Dipende solo dagli argomenti passati
- Non produce effetti collaterali es. `Math.random()`, chiamate HTTP...

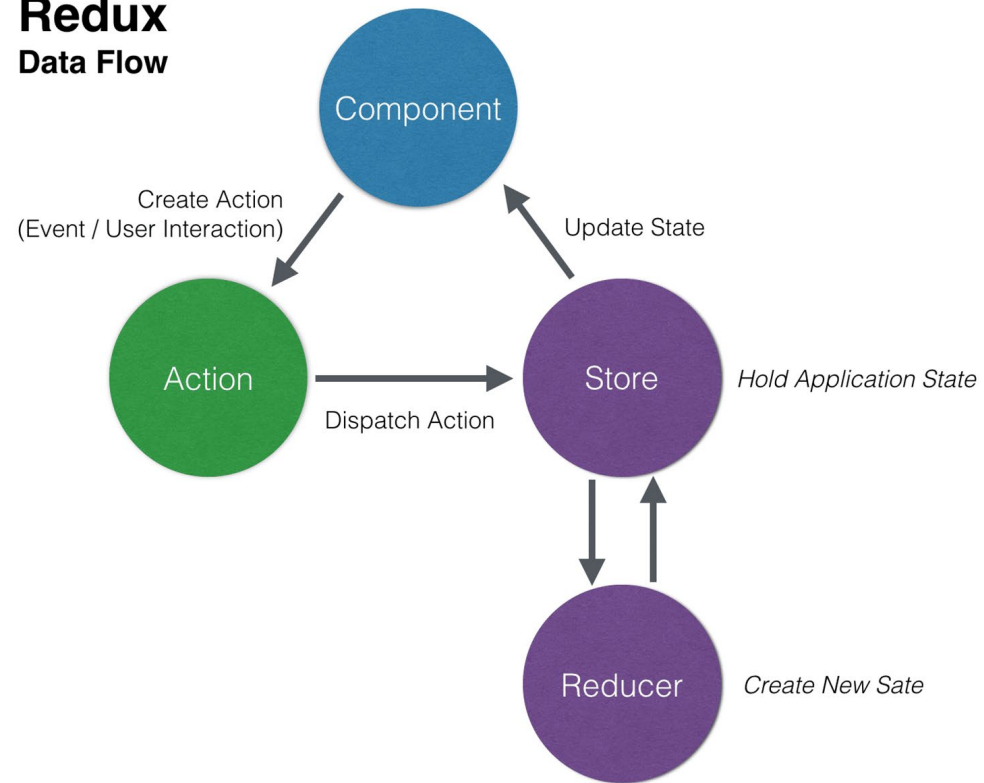
```
// Pure Function  
const add = (x, y) => x + y;  
add(2, 4); // 6  
  
  
// No Pure Function  
let x = 2;  
const add = (y) => {  
  x += y;  
};  
add(4); // x === 6 (the first time)
```

I Tre attori principali di REDUX

- Actions
- Reducers
- Store



Redux Data Flow



Questo “**store**” centralizzato sarà l’unica fonte di verità nell’applicazione, e sarà READ-ONLY e IMMUTABILE (non può essere modificato)

Ogni volta che sarà necessario applicare delle modifiche, un intero nuovo stato verrà generato, che prenderà il posto di quello precedente.

Inoltre, ogni iterazione successiva sarà semplicemente il risultato di un’operazione matematica, che renderà il nuovo stato prevedibile.

Ognuno di questi cambiamenti viene tracciato da Redux, e la loro natura prevedibile permette l’utilizzo di strumenti di debug molto utili come il logging automatico e funzioni di “time-travelling”

Ma come funziona?

Ogni volta che sarà necessario applicare una modifica allo stato condiviso (d'ora in poi Redux Store) sarà necessario fare il “**dispatch**” di un’**action**

Ad ogni “dispatch” di un’action, un **reducer** si aziona automaticamente, e genera il nuovo stato dell’applicativo partendo dallo stato precedente e dall’azione appena emessa

Questo nuovo stato sostituisce interamente quello precedente, e ogni componente collegato ad esso subirà un re-render

Cos'è una action?

È un semplice oggetto JavaScript con una proprietà obbligatoria chiamata **type**

Il resto delle proprietà deve essere la minima quantità di informazione possibile per fare in modo che il reducer sia in grado di calcolare il nuovo stato desiderato

Un action ha bisogno di essere spedita, “**dispatchata**” per fare in modo che il reducer venga azionato

Cos'è un reducer?

È una funzione pura (fornita dello stesso input, calcolerà sempre lo stesso output)
Intercetta ogni azione “**dispatchata**”, e insieme al vecchio stato è in grado di generare quello nuovo

Un **reducer** non muta MAI i suoi parametri (ad es. il vecchio stato)

Un **reducer** non esegue MAI side-effects (come chiamate API)

Non è detto che in un'app vi sia un solo reducer, se ne possono avere diversi (per rendere l'app più leggibile e mantenibile sul lungo periodo)

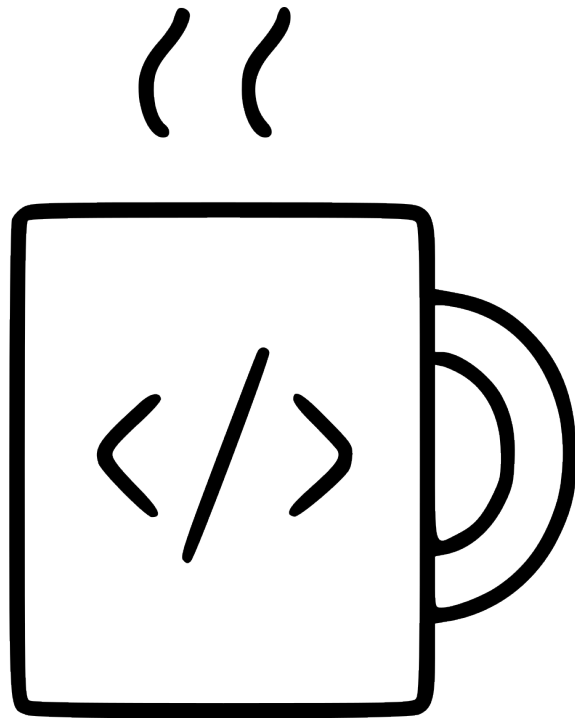
Store

```
type Store = {  
  dispatch: Dispatch  
  getState: () => State  
  subscribe: (listener: () => void) => () => void  
  replaceReducer: (reducer: Reducer) => void  
}
```

[Copy](#)

- Mantiene lo stato dell'applicazione;
- Consente l'accesso allo stato tramite `getState()` ;
- Consente di aggiornare lo stato tramite `dispatch(action)` ;
- Registra gli ascoltatori tramite `subscribe(listener)` ;
- Gestisce l'annullamento della registrazione degli ascoltatori tramite la funzione restituita da `subscribe(listener)` .

Uno **store** è un oggetto che contiene l'albero degli stati dell'applicazione. Ci dovrebbe essere un solo store in ogni app Redux



PAUSA

Ci vediamo alle ore 11.15

Store creator

```
type StoreCreator = (reducer: Reducer, preloadedState: ?State) => Store
```

src/store.js

```
import { createStore } from 'redux'  
import rootReducer from './reducer'  
  
const store = createStore(rootReducer)  
  
export default store
```

```
const store = createStore(rootReducer, preloadedState)
```

Ad uno **Store Creator** dobbiamo passare un Reducer e nel caso uno stato iniziale.


```
import {createStore} from 'redux';
```

```
const store = createStore((state = {}, action) => {  
  return state;  
})
```

Se lo si desidera, è possibile specificare lo stato iniziale come secondo argomento **createStore()**.

- Mantiene lo stato dell'applicazione;
- Consente l'accesso allo stato tramite `getState()` ;
- Consente di aggiornare lo stato tramite `dispatch(action)` ;
- Registra gli ascoltatori tramite `subscribe(listener)` ;
- Gestisce l'annullamento della registrazione degli ascoltatori tramite la funzione restituita da `subscribe(listener)` .

Per iniziare, dopo aver installato la dipendenza di Redux nel progetto, importare **createStore** di redux ed impostare il metodo **createStore** che prende in ingresso una funzione.

```
import React, { useState, useEffect } from 'react';
import { createStore } from 'redux';
import './App.css';

const myTodo = ['prova todo', 'mio todo'];

const storeReducer = (state = [], actions) => [...state];

function App() {

  const [todos, setTodos] = useState([]);
  //const store = createStore((state = [], actions) => [...state], myTodo);
  const store = createStore(storeReducer, todos);

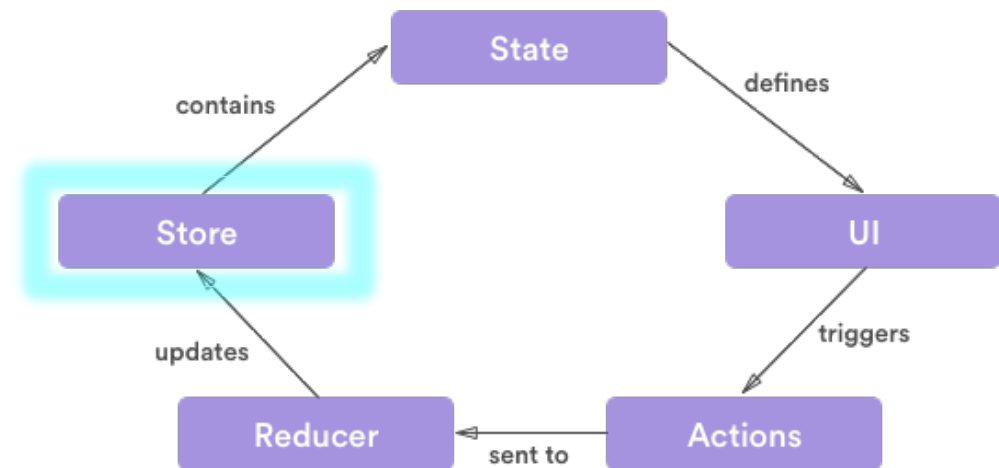
  useEffect(() => {
    setTodos(myTodo);
    console.log(store.getState());
  }, [])

  return (
    <div className="App">
      <header className="App-header">
        <h1>Todo LIST - React Redux</h1>
      </header>
      {store.getState().map((todo, index) => <p key={index}>{todo}</p>)}
    </div>
  );
}

export default App;
```

Creiamo una **ToDoList** utilizzando Redux.

Creiamo uno Store



```
function App() {  
  const [todos, setTodos] = useState([]);  
  //const store = createStore((state = [], actions) => [...state], myTodo);  
  const store = createStore(storeReducer, todos);  
  const inputTodo = React.createRef();
```

```
  return (  
    <div className="App">  
      <header className="App-header">  
        <h1>Todo LIST - React Redux</h1>  
      </header>  
      <input ref={inputTodo} />  
      <button onClick={ () => alert(inputTodo.current.value) } > ADD Todo </button>  
      {store.getState().map((todo, index) => <p key={index}>{todo}</p>)}  
    </div>  
  );
```

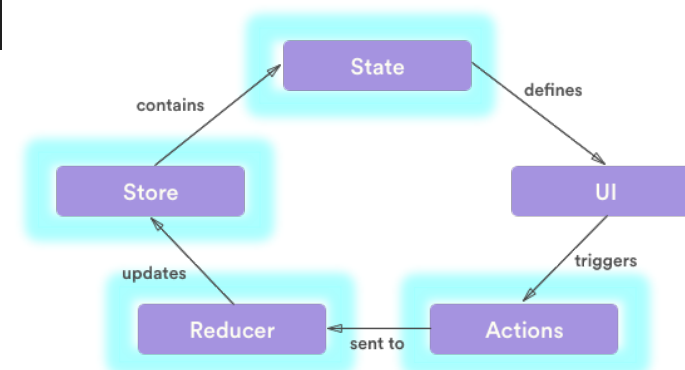
Aggiungiamo un **button** per l'inserimento di nuovi dati.

Utilizziamo un componente non controllato **input** creato con **React.createRef()**

```
function App() {  
  const [todos, setTodos] = useState([]);  
  //const store = createStore((state = [], actions) => {  
  const store = createStore(storeReducer, todos);  
  const inputTodo = React.createRef();  
  
  function storeReducer(state = [], actions) {  
    switch(actions.type) {  
      case 'ADD_TODO':  
        //alert(actions.todo);  
        return [actions.todo, ...todos];  
      default:  
        return [...todos];  
    }  
  }  
  
  store.subscribe(() => {  
    console.log('Store: ' + store.getState());  
    setTodos([...store.getState()]);  
  })  
  
  useEffect(() => {  
    setTodos(myTodo);  
    console.log(store.getState());  
  }, [])  
  
  return (  
    <div className="App">  
      <header className="App-header">  
        <h1>Todo LIST - React Redux</h1>  
      </header>  
      <input ref={inputTodo} />  
      <button onClick={ () => {  
        //alert(inputTodo.current.value)  
        store.dispatch({type: 'ADD_TODO', todo: inputTodo.current.value})  
      } } > ADD Todo </button>  
      {store.getState().map((todo, index) => <p key={index}>{todo}</p>)}  
    </div>  
  );  
}
```

- Mantiene lo stato dell'applicazione;
- Consente l'accesso allo stato tramite `getState()`;
- Consente di aggiornare lo stato tramite `dispatch(action)`;
- Registra gli ascoltatori tramite `subscribe(listener)`;
- Gestisce l'annullamento della registrazione degli ascoltatori tramite la funzione restituita da `unsubscribe()`.

Con il metodo **dispatch** aggiorniamo lo store tramite una funzione **reducer** che a sua volta attivando un ascoltatore **subscribe(listener)** aggiorna lo stato.

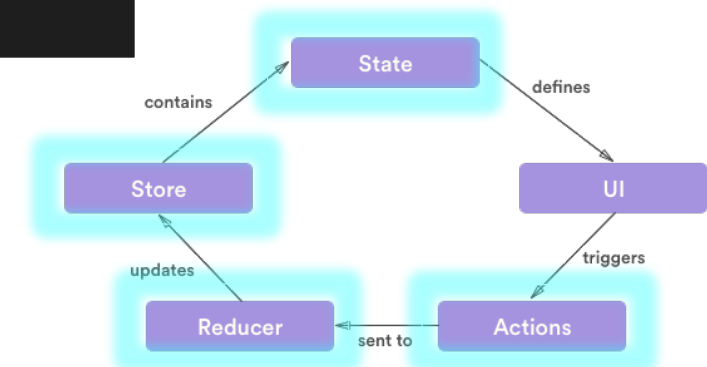


```
function storeReducer(state = [], actions) {
  switch(actions.type) {
    case 'ADD_TODO':
      //alert(actions.todo);
      return [actions.todo, ...todos]
    case 'REMOVE_TODO':
      //alert(actions.index);
      return [...todos.slice(0, actions.index), ...todos.slice(actions.index+1)]
    default:
      return [...todos];
  }
}

return (
  <div className="App">
    <header className="App-header">
      <h1>Todo LIST - React Redux</h1>
    </header>
    <input ref={inputTodo} />
    <button onClick={ () => {
      //alert(inputTodo.current.value)
      store.dispatch({type: 'ADD_TODO', todo: inputTodo.current.value})
    } } > ADD Todo </button>
    {store.getState().map((todo, index) =>
      <p key={index}>{todo}
        <button onClick={ () => store.dispatch({type: 'REMOVE_TODO', index: index})}>Remove</button>
      </p>)}
    </div>
  );
```

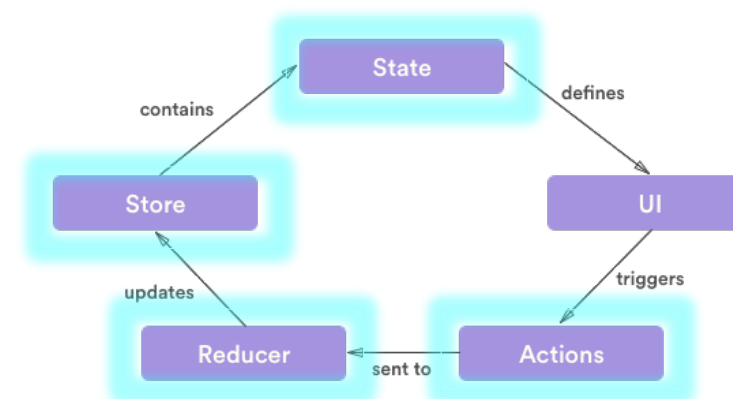
```
store.subscribe(() => {
  console.log('Store: ' + store.getState());
  setTodos([...store.getState()]);
})
```

Gestisco allo stesso modo la cancellazione di voci della todolist



- Mantiene lo stato dell'applicazione;
- Consente l'accesso allo stato tramite `getState()` ;
- Consente di aggiornare lo stato tramite `dispatch(action)` ;
- Registra gli ascoltatori tramite `subscribe(listener)` ;
- Gestisce l'annullamento della registrazione degli ascoltatori tramite la funzione restituita da `unsubscribe(listener)` .

Abbiamo visto il funzionamento di Redux, possiamo utilizzare Redux con Javascript, Angular, React senza nessun problema. Per utilizzare Redux in modo nativo, React ci mette a disposizione un modulo chiamato **React-Redux**. Per fare tutta la parte dei **Reducers**, le **Actions** e il **Dispatch** in maniera Automatica.





shaping the skills of tomorrow

challengenetwork.it

