14/07/2025 Bruzzone

REACT

Sfrutta i componenti

Componente = blocco di codice che rappresenta una parte dell’interfaccia utente.

Consente di suddividere l’interfaccia utente in parti più piccole rendendo il codice più organizzato e facile da gestire e mantenere . Può essere utilizzato in diverse parti dell’applicazione e anche in progetti diversi riducendo la duplicazione del codice .

Può ricevere dati dall’esterno tramite le props simili agli argomenti di una funzione.

Può anche gestire il proprio stato interno che è un insieme di dati che possono cambiare nel tempo e influenzare il odo in cui il componente viene renderizzato. Ogni volta che cambia una props o lo stato il componente viene renderizzato.

Quindi contiene uno stato (visibile solo dal componente) e le props visibili anche dal componente x che gliele passa. Tutto ciò che riceve è una prop, anche una funzione volendo.

Invece lo state non lo può leggere nessuno.

Esempio.

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

right: "-200px"

};

this.onOpenMenu = this.onOpenMenu.bind(this); //usare bind per i metodi

}

onOpenMenu() {

this.setState({ right: "0px" });

}

render() {

return (

<div style={{ right: this.state.right }}>

<button onClick={this.onOpenMenu}>Apri Menu</button>

</div>

);

}

}

Come una funzione che prende in input delle prop e restituisce html. Si può usare come un tag.

React tratta i componenti che iniziano con la lettera minuscola come normali tags DOM. quindi chiamare sempre i componenti con la prima lettera MAIUSCOLA.

<div> è un tag html. <Ciao> è un componente.

DOM = document object model. È la struttura di html.

Benefici di react

* Virtual DOM = copia virtuale del DOM. serve a mantenere una copia e aggiornare solo le parti che cambiano. Serve per ridurre il numero di render, e migliorare l’efficienza.
* Utilizzo di componenti.
* Possibilità di utilizzare molte librerie.

Render() e un metodo obbligatorio di ogni componente. Torna (return) html.

Differenze tra estensione JS e JSX

JS = javascript puro

JSX = estensione sintassi javascript usata per file che contengono anche jsx, sintassi simile a html usata da react per descrivere l’interaccia utente.

const elemento = <div> ciao </div>

si puo utilizzare jsx all’interno di if e cicli for, assegnarlo a variabili, utilizzarlo come argomento di una funzione e restituirlo come risultato di una funzione.

const oggi = {

giorno: “17”,

mese: “07”,

anno: “2025”

}

function formattaData(data) {

return data.giorno + “/” + data.mese + “/” + data.anno

}

function dammiIlBuongiorno(data) {

if(data) {

return <div>Buon {formattaData(data)}</div>

}

return <div>Buona giornata </div>

}

Come convertire una funzione in una classe.

function Clock(props) {

return(

<div>

<h1> ciao </h1>

<h2 > sono le props.data … </h2>

</div>)

}

---

Class Clock extends React.Component {

render() {

return(<div>

<h1> ciao </h1>

<h2 > sono le this.props.data … </h2>

</div>

);

}

}

Qui la data era una props del componente

Definisco costruttore per avere lo state.

Class Clock extends React.Component {

constructor(props);

super(props);

render() {

return(<div>

<h1> ciao </h1>

<h2 > sono le this.state.data … </h2>

</div>

);

}

**COS'È REACT?**

**React** è una **libreria JavaScript** (non un framework) sviluppata da Facebook, usata per creare **interfacce utente (UI)**, soprattutto per **applicazioni web a pagina singola (SPA)**.

React ti aiuta a costruire un'interfaccia **dividendo tutto in “componenti” riutilizzabili**.

**COMPONENTI IN REACT**

**Cos’è un componente?**

Un **componente** è come un **mattoncino LEGO**: costruisci un’interfaccia componendo tanti piccoli pezzi indipendenti.

Esempio semplice:

function Welcome() {

return <h1>Ciao, utente!</h1>;

}

Questo è un **componente funzionale** chiamato Welcome, che restituisce del codice **JSX** (un mix tra HTML e JavaScript).

**Tipi di componenti:**

1. **Funzionali (function component)**
2. **Classi (class component)** vecchio stile, quasi in disuso. Sono delle vere e proprie classi. La sintassi per definirli è Class NomeComponente e i 10 cicli di vita del componente. divisi in 3 (mounting updating unmounting).

**JSX - JavaScript + XML**

React usa **JSX**, una sintassi simile all’HTML ma scritta **dentro JavaScript**.

Esempio:

const nome = "Sofia";

const elemento = <h1>Ciao, {nome}</h1>; // JSX

{} in JSX ti permettono di **inserire JavaScript dentro l’HTML**.

**PROPS (proprietà)**

Le **props** servono per **passare dati ai componenti** (come parametri di una funzione).

function Welcome(props) {

return <h1>Ciao, {props.nome}</h1>;

}

Uso:

<Welcome nome="Sofia" />

**STATE (stato)**

Ogni componente può **memorizzare dati interni** nel suo **stato**, che può cambiare nel tempo.

**Ma come si gestisce lo stato?**

Con gli **Hook**.

**HOOKS – “ganci” per funzioni speciali**

Particolarita degli hooks. SI possono usare solo nel functional component. Nei class Component no.

**1. useState – per creare e modificare lo stato. Torna un array. Il primo elemento e lo stato e l’altro è il metodo per cambiare lo stato.**

import { useState } from "react";

function Contatore() {

const [numero, setNumero] = useState(0);

return (

<div>

<p>Hai cliccato {numero} volte</p>

<button onClick={() => setNumero(numero + 1)}>Clicca</button>

</div>

);

}

setNumero è una funzione che torna dall’Hook. useState() è l’hook e infatti va indicato prima. Non posso scriverlo dentro una funzione.

* useState(0) inizializza lo stato a 0
* numero è il valore attuale
* setNumero è la funzione per cambiarlo

**2. useEffect – per eseguire effetti collaterali (es: chiamate API, modifiche al DOM, timer)**

import { useEffect } from "react";

useEffect(() => {

console.log("Componente montato");

}, []); // eseguito una sola volta, quando il componente viene creato

L’array [] serve a indicare **quando** eseguire l'effetto.  
Se ci metti delle variabili, React eseguirà l'effetto ogni volta che cambiano.

Va a sostituire 3 metodi del lifecycle: il didmount, il didupdate e il willunmount.

I Custom Hooks sono funzioni che richiamano altri Hooks e si chiamano con use\*\*\*\*\*.

**ESEMPIO COMPLETO DI COMPONENTE FUNZIONALE**

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

function Timer() {

const [secondi, setSecondi] = useState(0);

useEffect(() => {

const intervallo = setInterval(() => {

setSecondi(prev => prev + 1);

}, 1000);

return () => clearInterval(intervallo); // pulizia

}, []); //nelle quadre ci sono di solito delle variabili che determinano la condizone pe cui venga chiamato lo useeffect, se non metto niente lo fa solo la prima volta.

return <p>Tempo trascorso: {secondi} secondi</p>;

}

| **Concetto** | **Cosa fa** |
| --- | --- |
| **Componenti** | Mattoni riutilizzabili che creano l’interfaccia |
| **JSX** | HTML scritto dentro JavaScript |
| **Props** | Parametri che passiamo ai componenti |
| **useState** | Per gestire lo stato interno di un componente |
| **useEffect** | Per gestire effetti collaterali (timer, fetch, ecc.) |

**SETSTATE()**

La funzione setState (o setQualcosa) **serve per aggiornare lo stato di un componente**.  
Ogni volta che aggiorni lo stato, **React “ri-renderizza” il componente**: cioè, aggiorna il contenuto mostrato a schermo.

**NELLE CLASSI: this.setState()**

import React, { Component } from 'react';

class Contatore extends Component {

constructor(props) {

super(props);

this.state = {

conteggio: 0

};

}

incrementa = () => {

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

};

render() {

return (

<div>

<p>Conteggio: {this.state.conteggio}</p>

<button onClick={this.incrementa}>Aumenta</button>

</div>

);

}

}

**NELLE FUNZIONI CON useState**

import React, { useState } from 'react';

function Contatore() {

const [conteggio, setConteggio] = useState(0);

return (

<div>

<p>Conteggio: {conteggio}</p>

<button onClick={() => setConteggio(conteggio + 1)}>Aumenta</button>

</div>

);

}

**Cosa succede qui:**

* useState(0) inizializza lo stato a 0
* conteggio è il valore attuale
* setConteggio è la funzione per **modificare il valore**
* Quando clicchi il bottone, setConteggio aggiorna il valore e React **ri-renderizza** la UI.

**IMPORTANTE**

* La funzione setX **non modifica subito** la variabile X (è asincrona).
* setConteggio(conteggio + 1);
* console.log(conteggio); // potrebbe NON stampare il nuovo valore subito

**Ciclo di vita di un componente.**

I componenti vengono:

* **Mounted** (creazione) 4 metodi
* **Update** (aggiornamento) 5 metodi
* **Unmounted** (Rimozione dal DOM) 1 metodo

Ciascuna fase ha i propri metodi che sono responsabili di un particolare periodo all’interno del ciclo di vita del componente.

**Creazione** detto anche render iniziale, avviene una sola volta. Formata da 4 metodi:

1. Constructor() serve ad inizializzare il componente. e per il binding dei metodi.

sintassi per definire lo state è essenziale super(props);

this.state = {

…

}

this.nomeFunzione = this.nomeFunzione(this) //per i metodi

1. static getDerivedStateFromProps()
2. Render() inserisce all’interno del dom l’html che io vado ad inserire.

Unico metodo obbligatorio della class component.

Torna solo codice html. Non puo tornare altro (NO console.log / NO chiamate API ☹)

1. componentDidMount() serve per togliere il componente dalla pagina.

**Aggiornamento**

1. static getDerivedStateFromProps()
2. shouldComponentUpdate() torna true o false
3. render()
4. getSnapShotBeforeUpdate()
5. componentDidUpdate() serve per fare eventuali chiamate API.

**Distruzione**

1. componentWillUnMount()

per creare un progetto/app si puo fare npm init

npm create react app -> crea progetto con gia con il necessario, ma pesante a livello di dipendenze e codice

npm start -> crea progettp

vite

Functional components

Sono delle vere e proprie funzioni al contrario dei class components. Sono questi che usano gli Hooks.

Sparisce il this.

Capitolo 10. Introduzione a typescript.

**REDUX**

Redux è una **libreria JavaScript** (non parte di React, ma spesso usata insieme) che serve per **gestire lo stato globale** di un’applicazione. Redux è come un contenitore di dati che tutti possono leggere e modificare. Sintassi difficile, ha dei listener da gestire.

Redux come un **contenitore centrale di dati (Store)**:

* tutti i componenti possono **leggere** i dati da lì,
* tutti i componenti possono **modificare** i dati lì dentro,
* ma le modifiche devono seguire regole precise.

Serve per:

* evitare di passare props da un componente all’altro in catena (prop drilling),
* avere lo **stato sincronizzato** in tutta l’app,
* rendere più semplice il debug e la manutenzione. Se si vuole cambiare un dettaglio piccolo. Quindi di decide di mettere alcuni dati (tipo i file html) nel redux per far si che possano modificarlo i tecnici senza rivolgersi allo sviluppatore.

**Principi fondamentali di Redux**

1. **Unico Store**. Tutta l’applicazione ha **un solo stato globale**, salvato nello Store (un grande oggetto JavaScript).

Es: { utente: { nome: "Sofia" }, carrello: [ ... ] }

1. **Lo Stato è immutabile**  
   Non si modifica lo stato direttamente.  
   Invece, si crea una **copia modificata** dello stato e Redux lo aggiorna.
2. **Le modifiche avvengono tramite Azioni (Actions)**  
   Un’azione è un **oggetto** che descrive cosa deve succedere.

Es: { type: "AGGIUNGI\_ARTICOLO", payload: { id: 1, nome: "Pizza" } }

1. **I Reducer dicono come aggiornare lo stato**  
   Un reducer è una **funzione pura** che riceve lo stato attuale e un’azione, e restituisce il nuovo stato.

function carrelloReducer(state = [], action) {

switch (action.type) {

case "AGGIUNGI\_ARTICOLO":

return [...state, action.payload];

default:

return state;

}

}

1. **I Componenti leggono e inviano azioni**
   * Leggono i dati dallo Store con useSelector.
   * Modificano i dati inviando azioni con useDispatch.

**Flusso dei dati in Redux**

1. **UI** → l’utente clicca un bottone
2. **Dispatch di un’azione** → dispatch({ type: "AGGIUNGI\_ARTICOLO", payload: {...} })
3. **Reducer** → decide come cambiare lo stato
4. **Nuovo stato nello Store**
5. **UI aggiornata automaticamente**

Perche typescript è utilizzato? Ad andre non piace

Esempio:

**\***

function MyComp() {

**\***

let pippo = {

nome: “pippo”,

cognome:”baudo”,

sesso:”M”

}

const [utente, setUtente] = UseState(pippo) **\***

function cambia() {

setUtente(null);

} //lo posso fare

return() {

<h1> anagrafica </h1>

<h2> nome: </h2>

<p> {utente.nome} </p> //ma a questo punto esplode

}

}

Allora typescrit permette di fare i tipi che aiutano in fase di build, Javascript me l’avrebbe evitato perche non avrebbe buildato.

type Persona = {

nome: string;

cognome: string;

sesso: string;

}

type Anagrafica {

nome: string;

cognome: string;

sesso: string;

indirizzo: string;

cap: string;

}

E poi faccio nel codice di prima:

const [utente, setUtente] = UseState<Persona>(pippo) **\***

con le interfacce ho il vantaggio dell’extends

interface Persona {

nome: string;

cognome: string;

sesso: string;

data?: string;

}

Interface Anagrafica extends Persona {

indirizzo: string;

cap: string;

}