**Corso React (base)**

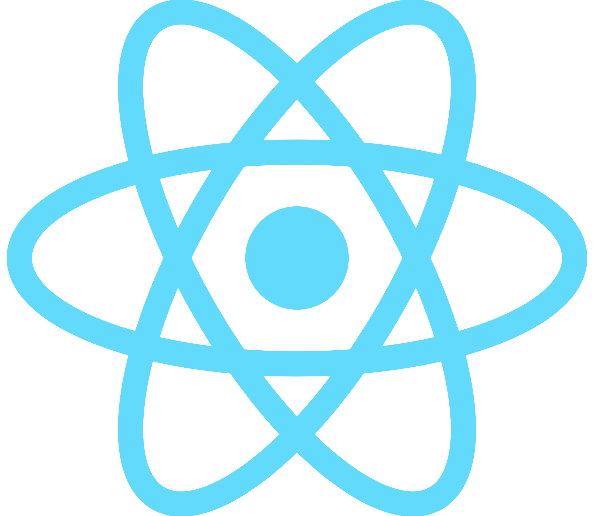


Figura 1 react\_logo.png

Link al corso: <https://www.youtube.com/watch?v=TCYnHcngPLc&list=PLP5MAKLy8lP9Ekc_hVSggV7ZkUzgW7IrH>

**Capitolo 1 – Introduzione**

Immagine che contiene nuvola, dipinto, paesaggio, arte

Descrizione generata automaticamente

Figura 2 "So it **begins**" - Theoden Hammerwand

**Cosa è?**

React è una **libreria** che serve a costruire applicazioni.

Essendo una libreria, e non un framework, può avere delle diramazioni che lo strutturino maggiormente, come Next.Js e Remix.Js.

**Capitolo 2 – Installazione**

Immagine che contiene Modello in scala, Set per costruzioni, Mattoncino giocattolo, Set di gioco

Descrizione generata automaticamente

Figura 3 "I **built** this sh\*t brick by brick!" - Franklin Saint Lee

1. Installazione Node.Js come environment
2. Installazione VSCode
3. Create React App **\***
4. React Vite

4a. Usiamo il commando **npm create vite@latest corso-react -- --template react** (<https://vitejs.dev/guide/>)

4b. Entriamo nella cartella (**cd corso-react**)

4c. Installiamo i pacchetti (**npm install**)

4d. Avviamo il progetto in development mode (**npm run dev**)

**\*** NOTA: Questo metodo di inizializzaione di un progetto in react non è deprecato, ma non è più consigliato per la creazione di nuove UI in React.

**Capitolo 3 – Struttura del Progetto**

Immagine che contiene testo, schermata, cielo

Descrizione generata automaticamente

Figura 4 "This is...**construct**" - Matrix

**Struttura**:

* **node\_modules**: Pacchetti con le librerie node del progetto
* **public**: cartella con tutte le risorse pubbliche del sito
* **src**: codice effettivo de progetto
* **eslintrc.cjs**: strumento che forza determinate consuetudini di scrittura codice in fase di sviluppo
* **gitignore**: indici dei files da non caricare in remoto
* **index.html**: la pagina principale che fa da entry point
* **package.json**: file di configurazione che raccogli dipendenze (necessarie per il progetto) e devDipendenze (necessarie solo allo sviluppatore), script, ecc…
* **package-lock.json**: file di configurazione che raccoglie TUTTI gli indici delle dipendenze, mentre il package.json raccoglie solo le dipendenze principali usate da lo sviluppatore
* **reademe.md**: file descrittivo
* **vite.config.json**: file di configuazione di vite, il motore di build del progetto

**Capitolo 4 – Rudimenti di Sintassi**

Immagine che contiene testo, libreria, Pubblicazione, biblioteca

Descrizione generata automaticamente

Figura 5 The **Verb**

Struttura di un applicativo React:

* **Entry point**: **index.html**, questo ha un elemento con **id=root** che contiene tutto il codice

che viene iniettato dallo **script main.jsx**

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

* **Inizializzazione contenuto**: **main.jsx**, modulo che inizializza l’applicazione importando le **main libraries**

Immagine che contiene testo, schermata, diagramma, linea

Descrizione generata automaticamente

La **struttura** di base di un **componente** consiste in una **funzione** che effettua il **return** di un blocco di **jsx**.



Immagine che contiene testo, schermata, software

Descrizione generata automaticamente

**Capitolo 5 – Introduzione ai componenti**

Immagine che contiene schermata

Descrizione generata automaticamente

Figura 6 We are **pieces** of the same cake

I componenti sono **funzioni** javascript che generano blocchi funzionali che generano **“html”**

Un componente viene generato a mano, la sua struttura di base è costituita da una funzione che contiene delle istruzioni in jsx (un js glorified) che restituiranno l’html voluto.

Tutti i componenti verrano incapsulati all’interno del componente patriarca denominato App.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 7 esempio di componente Navbar…

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 8 ...inserito all'interno del componente primigenio

**Capitolo 6 – JSX**

Immagine che contiene libro, schermata, Gioco per PC, interno

Descrizione generata automaticamente

Figura 9 **Js** with steroids!

**Regole base della sintassi JSX**:

1. Le parentesi servono quando ci sono più righe di codice
2. Ogni componente deve restituire un unico elemento
3. In considerazione della regola 2, ogni componente può essere attorniato non da un singolo div, ma dal **fragment**

Immagine che contiene schermata, Carattere, Elementi grafici, design

Descrizione generata automaticamente

1. Ogni tag element deve essere chiuso

Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamente

1. Molti attributi diventano in **camelCase…**
2. … un’eccezione è **className**, essendo class una parola chiave di js
3. Una regola base è l’interpolazione di **valori** (primitive, oggetti o addirittura intere funzioni)

Immagine che contiene testo, Carattere, schermata, design

Descrizione generata automaticamente

1. Con le **doppie graffe** è possibile inserire anche **oggetti**, esempio per lo stile



**Capitolo 7 – Stile CSS**

Immagine che contiene dipinto, arte, Arte moderna, portafotografie

Descrizione generata automaticamente

Figura 10 “I'm pretty sure there's a lot more to life than being **really, really, ridiculously good looking**. And I plan on finding out what that is.” - Zoolander

**Importante**: ogni file css, per quanto correlato ad uno specifico componente, se questo è importato, sovrascriverà lo stile degli altri componenti, poiché sono **globali**.

Una libreria che evita questo problema e localizza lo stile allo specifico componente è [**styled-components**](https://styled-components.com/).

**Stilizzazione**:

* È possibile stilizzare **inline** un elemento html, oppure usare l’interpolazione a doppia graffa per importare un oggetto di stile.

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Figura 11 Nota: lo stile dovrà essere trattato come un oggetto

* È possibile usare una **classe**, come nel classico html, usando l’attributo **className**



* Essendo un oggetto lo stile può contenere tutta una serie di operazioni affinché questo sia reattivo

Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente

* È possibile inserire classi dinamicamente

Immagine che contiene testo, schermata, Carattere, numero

Descrizione generata automaticamente

Figura 12 classi



Figura 13 jsx

**Capitolo 8 – Integrazione Tailwind CSS**

Immagine che contiene mammifero, cartone animato, cielo, dipinto

Descrizione generata automaticamente

Figura 14 "I'm Bootstrap, but **stronger**."

* Installazione di tailwind css, autoprefixer e postcss cli

**npm install -D tailwindcss postcss autoprefixer**

* Inizializzaione config di tailwind

npx tailwindcss init -p

* Gestione tailwind.config.js

/\*\* @type {import('tailwindcss').Config} \*/

export default {

  content: ["./index.html", "./src/\*\*/\*.{js,ts,jsx,tsx}"],

  theme: {

    extend: {},

  },

  plugins: [],

};

* Import delle directives di tailwind css

@tailwind base;

@tailwind components;

@tailwind utilities;

* Avvio del servizio

**npm run dev**

* Esempio di uso in un h3



**Capitolo 9 – Props (proprietà dei componenti)**

Immagine che contiene disegno, arte, pianta, aria aperta

Descrizione generata automaticamente

Figura 15 “What distinguishes communism is not the abolition of props in general, but the abolition of bourgeois **props**.” - K. Marx

**Cosa sono?**

Le **props** sono **attributi** specifici di React che possono essere usati sui componenti.

Questi permettono il passaggio dei dati da un componente ad un altro.

Avendo un componente **Card**

Immagine che contiene testo, schermata, software, Software multimediale

Descrizione generata automaticamente

Questo presenta un blocco html con un’immagine, un titolo e un paragrafo, affinchè questi siano dinamici e che i dati vengano inseriti dal componente padre, bisogna passare le props come argument della funzione componente, così da popolare l’html.

Immagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamenteIl **componente** richiamato dentro il **padre** conterrà i valori da passare dentro le singole **card**.

Essendo le props un oggetto è possibile destrutturalo per inviarlo, così da rendere più efficiente il codice

Da

Card(props)

a

 Card({ title, img, description })

**Children**: è possibile invertire la situazione utilizzando la keyword **children**, questa permetteràdi passare il codice posto dentro il componente innestato direttamente dentro il componente figlio!

Componente **Padre** (App.jsx):

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Componente **Figlio** (Card.jsx):

Immagine che contiene schermata, testo, Carattere, numero

Descrizione generata automaticamente

**Capitolo 10 – Rendering condizionale**

Immagine che contiene Policromia, schermata, Elementi grafici, arte

Descrizione generata automaticamente

Figura 16 The only **response** to “what i do of my life?!?”

Il rendering condizionale permette di visualizzare, in base al soddisfacimento di condizioni, una parte di codice.

Es.

Vi sono delle card, una delle props “**isVisited**” è un booleano che indica se, la città nella card, sia stata o meno visitata

Componente Padre:

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

Componente Figlio:



Naturalmente è possibile definire a più livelli il rendering

Risultato:

Immagine che contiene testo, cielo, schermata, panorama

Descrizione generata automaticamente

Naturalmente gli approcci possibili…





…sono praticamente infiniti!

**AND**: operatore utilizzato per la medesima cosa, in base ad una condizione renderizza un contenuto



**Capitolo 11 – Rendering di liste**

Immagine che contiene testo, libro, forniture per ufficio, cancelleria

Descrizione generata automaticamente

Figura 17 The **list** of things I don't give a f\*\*k about

Il rendering di liste è uno dei processi più basilari quanto importanti all’interno di una applicazione scritta usando React.

Il concetto è semplice: uso del metodo **.map()** per il print di un componente figlio che riceve una serie di proprietà da un array di oggetti.

Immagine che contiene testo, schermata, Carattere, software

Descrizione generata automaticamente

La funzione effettua il **return** della card con le **props** innestate che verrano visualizzate nel componente figlio.

**Key**: key è una keyword che indica un **valore** univoco identificante il componente

Nota: fondamentale è l’uso di **filter** nel qual caso in cui si dovesse **filtrare la lista** per le città **visitate**

Immagine che contiene testo, Carattere, schermata, Elementi grafici

Descrizione generata automaticamente

**Capitolo 12 – Eventi**

Immagine che contiene pixel, cartone animato, clipart

Descrizione generata automaticamente

Figura 18 C’mon, **click** this cute mouse!

React fornisce la possibilità di gestire degli eventi, quali il click, keyup, ecc… in modo tale far scaturire delle reazioni di funzioni.

La sintassi è simile a quella solita di JS, con la differenza che riscrive in **camelCase** l’evento.

**onclick 🡪 onClick**

La funzione può essere intercalata tramite una **arrow function** o richiamata se definita **esternamente** al return.

Caso 1:

Immagine che contiene testo, Carattere, schermata, numero

Descrizione generata automaticamente

Caso 2:

Immagine che contiene testo, Carattere, schermata, Elementi grafici

Descrizione generata automaticamente



***Nota****:*

*Le funzioni passate ai gestori eventi devono essere passate, non chiamate. Per esempio:*

*Immagine che contiene testo, schermata, Carattere, linea

Descrizione generata automaticamente*

*La differenza è sottile. Nel primo esempio, la handleClickfunzione viene passata come onClickgestore di eventi. Questo dice a React di ricordarlo e di chiamare la tua funzione solo quando l'utente fa clic sul pulsante.*

*Nel secondo esempio, () alla fine handleClick()attiva la funzione immediatamente durante* [*il rendering*](https://react.dev/learn/render-and-commit) *, senza alcun clic. Questo perché JavaScript è all'interno di* [*JSX {e}*](https://react.dev/learn/javascript-in-jsx-with-curly-braces) *viene eseguito immediatamente.*

*Quando scrivi il codice in linea, la stessa trappola si presenta in modo diverso:*

*Immagine che contiene testo, schermata, Carattere, Blu elettrico

Descrizione generata automaticamente*

*Il passaggio di codice in linea come questo non si attiva al clic, ma si attiva ogni volta che il componente esegue il rendering:*

*// This alert fires when the component renders, not when clicked!*

*<button onClick={alert('You clicked me!')}>*

*Se vuoi definire il tuo gestore eventi in linea, avvolgilo in una funzione anonima in questo modo:*

*<button onClick={() => alert('You clicked me!')}>*

*Invece di eseguire il codice all'interno con ogni rendering, questo crea una funzione da chiamare in seguito.*

*In entrambi i casi, ciò che vuoi passare è una funzione:*

*<button onClick={handleClick}>passa la handleClickfunzione.*

*<button onClick={() => alert('...')}>passa la () => alert('...')funzione.*

**Capitolo 13 – Gestione dello stato**

Immagine che contiene schermata, Videogioco di strategia

Descrizione generata automaticamente

Figure 1 It's part of the act the fifty **states**…

“I componenti spesso necessitano di modificare ciò che appare sullo schermo come risultato di un'interazione. Digitando in un form si dovrebbe aggiornare il campo di input, facendo clic su "Avanti" su un carosello di immagini si dovrebbe cambiare l'immagine visualizzata, facendo clic su "acquista" si dovrebbe inserire un prodotto nel carrello. I componenti devono “ricordare” le cose: il valore di input corrente, l'immagine corrente, il carrello della spesa. In React, questo tipo di memoria specifica del componente è chiamato **state**.”

La **gestione di un state** viene effettuata da uno dei tanti **hook\*** di React.

Lo useState, destrutturarato, darà il valore e la funzione.

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

lo **useState** definisce il valore iniziale (pari a 0 in questo caso), ovvero il valore di **count**, questo valore verrà gestito dalla funzione **setCount**.

Il valore dello stato precedente e viene sovrascritto da quello successivo

[0 🡪 1] | Lo stato iniziale non esiste più!

      <button

        className="bg-slate-700 text-white p-2 rounded-md"

        onClick={() => {

          setCount((count) => count + 1);

        }}>

        count è {count}

      </button>

Questo permetterà la visualizzazione del count con il valore aggiornato, poiché react “reagisce” al cambio di stato, ri-renderizzando il componente.

**\*Hook**: è un elemento che permette di effettuare varie operazioni

Usando del js normale non ci sarà il re-render del componente, portando alla mancata visualizzazione del nuovo valore.

**Es.**

  let conteggio = 0;

      <button

        className="bg-red-700 text-white p-2 rounded-md"

        onClick={() => {

          conteggio++;

        }}>

        conteggio è {conteggio}

      </button>

Immagine che contiene testo, schermata, Carattere, Elementi grafici

Descrizione generata automaticamente



**Es.2**

  const [items, setItems] = useState([1, 2, 3]);

  function aggiungiItem() {

    let nuovoItem = 4;

    setItems([...items, nuovoItem]);

    console.log(items);

  }

        <button

          className="bg-red-700 text-white p-2 rounded-md"

          onClick={aggiungiItem}>

          Items sono {items}

        </button>

Output

Immagine che contiene schermata, Carattere, numero, testo

Descrizione generata automaticamente

Avendo un array di int e volendolo gestire tramite state, aggiungendo un int supplementare, è possibile utilizzare il setItems usando come primo argument uno **spread operator**, così da richiamare tutti i valori dell’array di items.

Trasporta tutto, e modifica qualcosa



**Lift the State up!**

**Capitolo 14 – From Child to Father**

Immagine che contiene cartone animato, giocattolo, Animazione

Descrizione generata automaticamente

Figure 2 like father like **son**

Una delle azioni fondamentali quando si gestiscono più componenti è il passaggio di stato da un componente figlio ad uno genitore che lo avvolge.

Flusso:

1. Abbiamo la definizione di uno state, avente come stato iniziale 4 città (cities) e una funzione (addCities) che serve ad aggiungerne una.

Immagine che contiene testo, Carattere, schermata, Elementi grafici

Descrizione generata automaticamente

1. Questa funzione dello useState viene wrappata all’interno di una funzione d’uso (addCity), che potrà accettare come arg un oggetto relativo una nuova città (city)

Immagine che contiene testo, Carattere, schermata, Elementi grafici

Descrizione generata automaticamente

1. Questa funzione (addCity) verrà inviata come prop al componente figlio (CardForm)



1. CardForm userà una sua funzione interna (handleClick) dentro il quale richiamare il riferimento della funzione iniettata (addCity) che userà un arg interno a questo child

Immagine che contiene testo, schermata, Carattere, schermo

Descrizione generata automaticamente

1. La funzione del componente padre verrà richiamata, aggiungendo una città!

Immagine che contiene testo, schermata, paesaggio, natura

Descrizione generata automaticamente

**Capitolo 15 – Form**

Immagine che contiene testo, schermata, tabellone

Descrizione generata automaticamente

**Capitolo 16 – React Developer Tool**

[…]

**Capitolo 17 – useEffect**

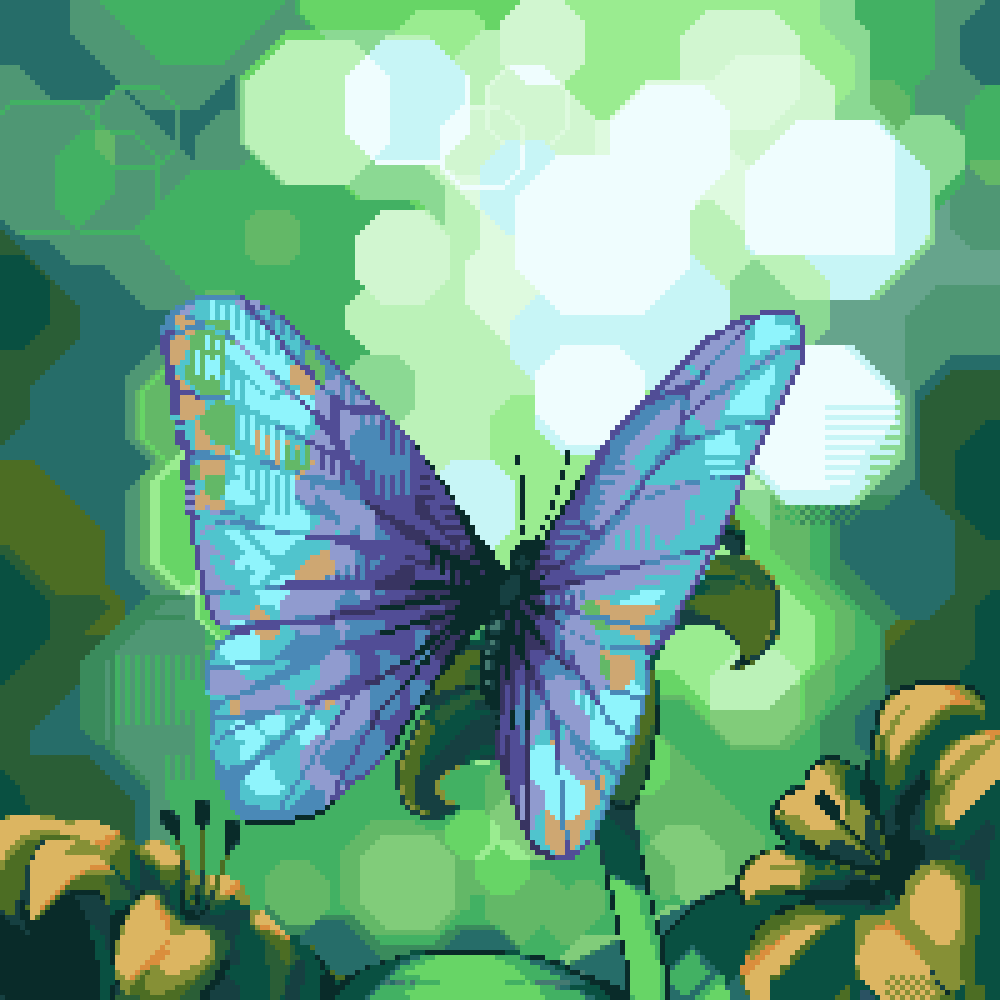


Figure 3 Butterfly **useEffect**

Lo **useEffect**, insieme allo useState, è uno degli hook più importanti di React che ti consente di sincronizzare un componente con un sistema esterno, gestendo quelli che sono tutti i side effect che esulano dal mero rendering a schermo.

Lo useEffect permette di eseguire effetti collaterali nelle componenti funzionali. Questo hook viene eseguito ogni volta che uno o più valori dipendenti cambiano. È simile a combinare i lifecycle methods componentDidMount, componentDidUpdate, e componentWillUnmount in una sola funzione. Può essere utilizzato per effettuare chiamate API, manipolare il DOM, gestire sottoscrizioni, e altro ancora.

***Nota****: In Angular esiste un hook denominato* ***ngOnChanges****, un metodo di lifecycle hook che viene chiamato ogni volta che i dati di input di un componente cambiano. Questo metodo fornisce un oggetto che contiene i cambiamenti dei dati in modo da poter reagire di conseguenza. Viene spesso utilizzato per effettuare azioni come aggiornare lo stato interno del componente o eseguire operazioni di rendering condizionale.*

*Entrambi sono utilizzati per gestire i cambiamenti nei dati, la loro implementazione e il loro utilizzo sono diversi, poiché React e Angular hanno approcci diversi alla gestione dello stato e dei lifecycle delle componenti.*

**Immagine che contiene testo, schermata, Carattere, diagramma

Descrizione generata automaticamenteStruttura dello useEffect:**

Lo useEffect effettua delle operazioni in base alla **variazione di stato** delle dipendenze inserite, se queste non sono presenti, lo useEffect verrà triggerato ad ogni ciclo di vita del componente.

In questo caso, alla variazione del count, verrà modificato il title del document.

***Nota****: Se volessimo che lo useEffect venga avviato unicamente al primo rendering del componente, bisognerà specificare inserendo come dipendenze un array vuoto ()*

**Cleanup:**

*Il* ***Cleanup*** *è una funzione che viene eseguita quando un componente React viene smontato o "ripulito" dal DOM.*

*Quando si utilizza useEffect per effettuare effetti collaterali come l'aggiornamento dello stato, l'interazione con API esterne o la sottoscrizione a eventi, è possibile che si desideri "ripulire" dopo se stessi quando il componente viene rimosso dal DOM o quando il componente cambia.*

*Ad esempio, se ti sottoscrivi a un evento globale all'interno di useEffect, dovresti assicurarti di cancellare tale sottoscrizione quando il componente viene rimosso per evitare memory leak o comportamenti indesiderati.*

*Per fare questo, all'interno della funzione di callback di useEffect, puoi ritornare una funzione che eseguirà il cleanup quando necessario. Questa funzione di cleanup verrà eseguita quando il componente viene smontato.*

Es.

import React, { useEffect } from 'react';

function MyComponent() {

  useEffect(() => {

    // Effetto collaterale: sottoscrizione a un evento

    window.addEventListener('resize', handleResize);

    // Funzione di cleanup

    return () => {

      window.removeEventListener('resize', handleResize);

    };

  }, []); // L'array vuoto [] assicura che l'effetto venga eseguito solo una volta durante il montaggio e smontaggio del componente

  const handleResize = () => {

    // Gestisce il ridimensionamento della finestra

  };

  return (

    <div>

      {/\* Contenuto del componente \*/}

    </div>

  );

}

export default MyComponent;

**Capitolo 18 – Chiamate HTTP**

Immagine che contiene cartone animato, interno

Descrizione generata automaticamente

Figura 19 So **call** me maybe...

In React, le metodologie che permettono di effettuare chiamate http possono essere molteplici, ma il principale è **fetch()**, metodo build-in di JavaScript.

Puoi utilizzare il metodo fetch() per recuperare risorse da un URL, come dati JSON, file di testo o risorse binarie come immagini. Questa funzione restituisce una Promise che risolve con un oggetto Response rappresentante la risposta alla richiesta HTTP.

Ecco un esempio di base di come puoi utilizzare fetch():

fetch("https://jsonplaceholder.typicode.com/posts", {

  method: "GET",

})

.then((response) => {

  return response.json(); // Conversione della risposta in formato json

})

.then((data) => {

  console.log(data);

  setData(data);

})

.catch((error) => {

  console.log(error);

});

In questo esempio, fetch() viene utilizzato per effettuare una richiesta GET all'URL specificato. Successivamente, vengono gestiti i due blocchi .then(): il primo verifica se la risposta è stata ricevuta con successo e la converte in formato JSON, mentre il secondo utilizza i dati ricevuti. Infine, eventuali errori vengono gestiti nel blocco .catch().

Un **esempio** di chiamata http effettuata in fase di rendering del componente è quella sfruttante lo useEffect, con gestite le relative dipendenze e state:

  const [data, setData] = useState([]);

  useEffect(() => {

    fetch("https://jsonplaceholder.typicode.com/posts", {

      method: "GET",

    })

      .then((response) => {

        return response.json();

      })

      .then((data) => {

        console.log(data);

        setData(data);

      })

      .catch((error) => {

        console.log(error);

      });

  }, []);

**Capitolo 19 – useReducer**

Immagine che contiene testo, cartone animato, schermata

Descrizione generata automaticamente

Lo **useReducer** è una evoluzione dello useState,

**Capitolo 20 – Context API**

[…]

**Capitolo 21 – Redux**

[…]

**Capitolo 22 – Routing**

Immagine che contiene edificio, aria aperta, finestra, Urbanistica

Descrizione generata automaticamente

Figura 20 It winds from Chicago to la, more than two thousand miles all the way. Get your kicks on **route** sixty-six... -Nat K. Cole

**Il routing è l’instradamento degli utenti su quella che è in realtà è una applicazione single page, renderizzando tutto in una singola pagina index.**

La libreria principalmente utilizzata per gestire il routing è **React-Router**.

1. **Installazione** della libreria
2. Import di essa nel file **main.jsx**

import { createBrowserRouter, RouterProvider } from "react-router-dom";

1. Definizione della struttura primaria del **routing**

Istanziamo l’array di oggetti router, il quale definirà il telaio del routing dell’applicazione

const router = createBrowserRouter([

  {

…

Root della spa

  {

    path: "/",

    element: <App></App>,

  },

…

Singola pagina

  {

    path: "/contacts",

    element: <Contacts></Contacts>,

  },

…

Pagina da cui instradare sotto-pagine definite dinamicamente ed **esplicitamente**

  {

    path: "/cards",

    element: <Cards></Cards>,

  },

  {

    path: "/cards/:cardID",

    element: <Card></Card>,

  },

…

Pagina da cui instradare sotto-pagine definite dinamicamente ed **implicitamente** da cui definire dei children

  {

    path: "/cards-children",

    element: <CardsChildren></CardsChildren>,

    children: [

      {

        path: "/cards-children/:cardID",

        element: <CardChildren></CardChildren>,

      },

    ],

  },

…

Da qui bisognerà istanziare il tutto tramite il **RouterProvider**

ReactDOM.createRoot(document.getElementById("root")).render(

  <React.StrictMode>

    <RouterProvider router={router} />

  </React.StrictMode>

);