Corso React

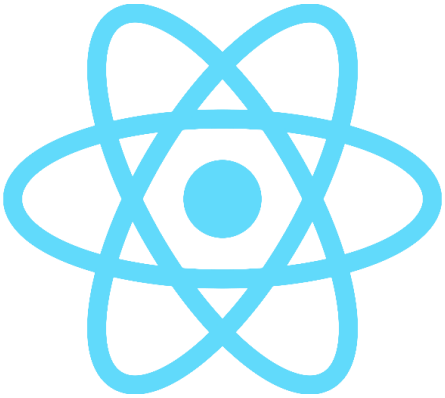


Figura 1 react_logo.png

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

Capitolo 1 – Introduzione

*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

*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

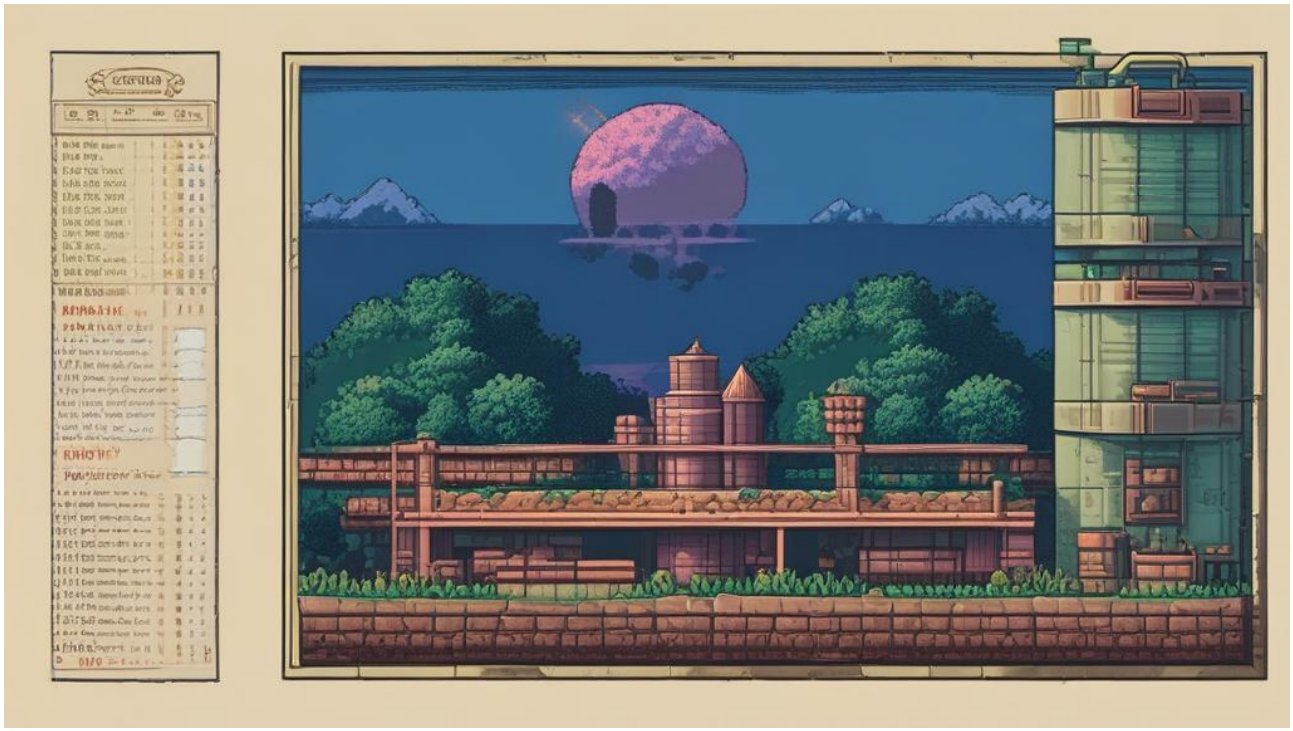


*Figura 4 "This is...**construct**" - Matrix*

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

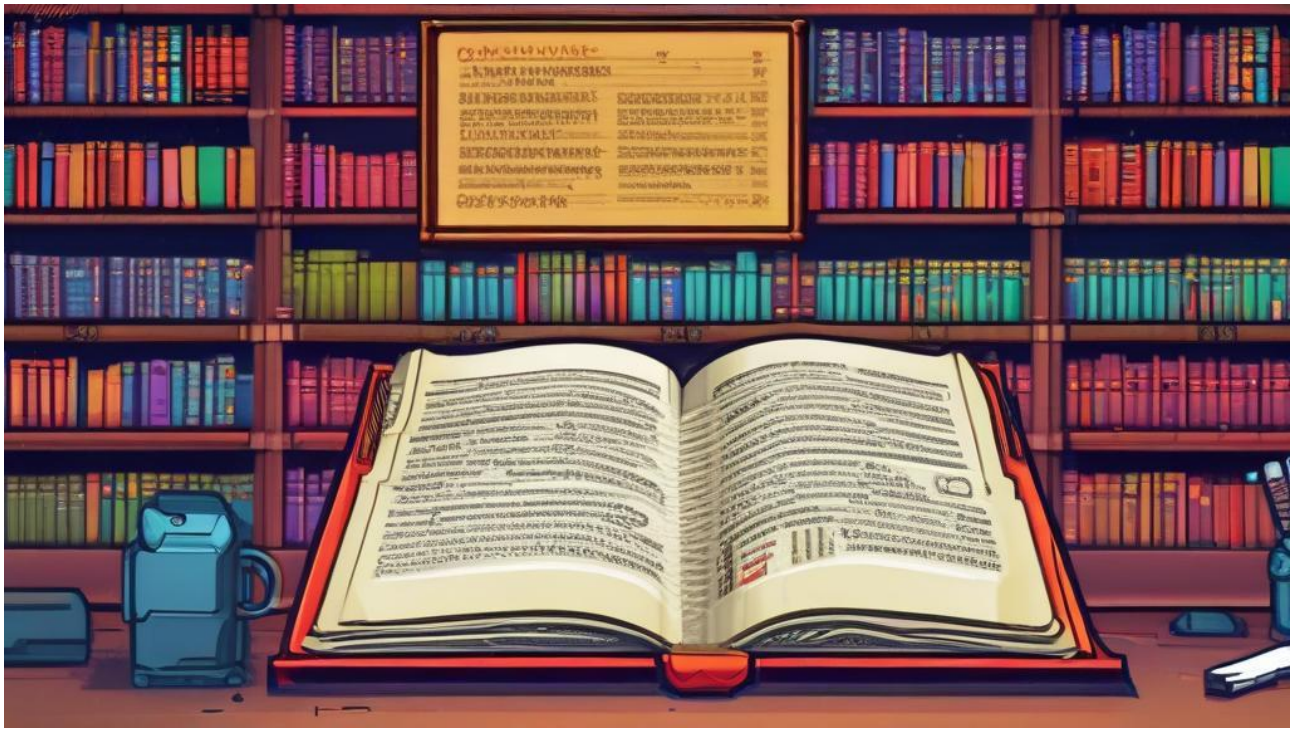


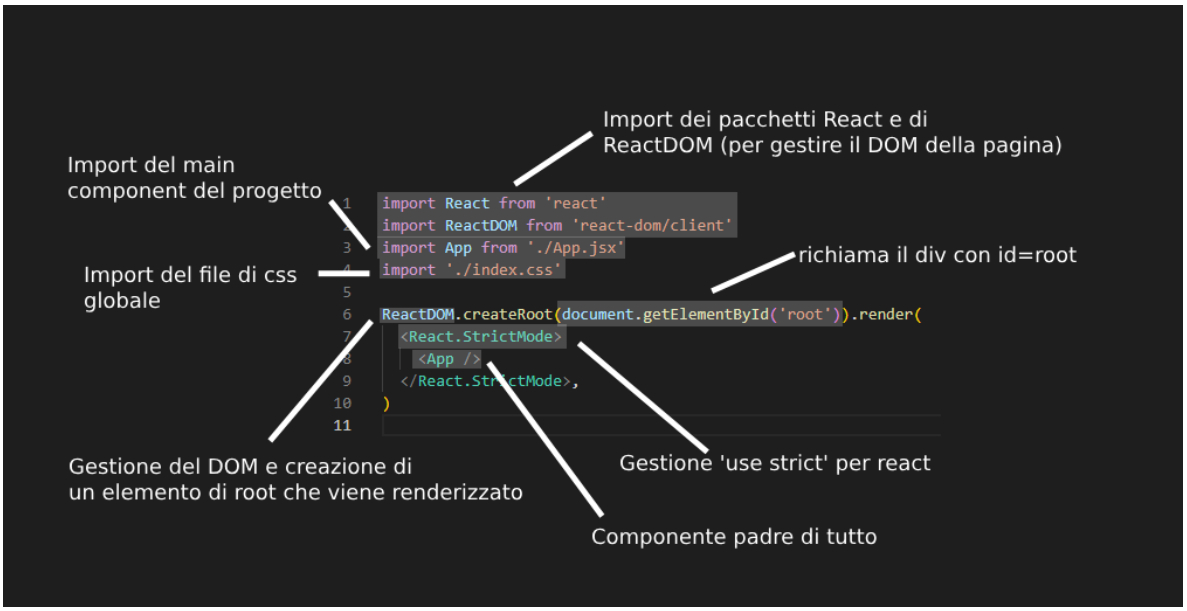
Figura 5 "Atra gülai un ilian tauthr ono un atra ono waíse skölir frá rauthr" (t. "May luck and happiness follow you and may you be a shield from misfortune" – Ancient language)

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**

```
9   <body>
10    <div id="root"></div>
11    <script type="module" src="/src/main.jsx"></script>
12  </body>
```

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



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

App.css
App.jsx

```jsx
import { useState } from 'react'
import reactLogo from './assets/react.svg'
import viteLogo from '/vite.svg'
import './App.css'

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

  return (
    <>
      <div>
        <a href="https://vitejs.dev" target="_blank">
          <img src={viteLogo} className="logo" alt="Vite logo" />
        </a>
        <a href="https://react.dev" target="_blank">
          <img src={reactLogo} className="logo react" alt="React logo" />
        </a>
      </div>
      <h1>Vite + React</h1>
      <div className="card">
        <button onClick={() => setCount((count) => count + 1)}>
          count is {count}
        </button>
        <p>
          Edit <code>src/App.jsx</code> and save to test HMR
        </p>
      </div>
      <p className="read-the-docs">
        Click on the Vite and React logos to learn more
      </p>
    </>
  )
}

export default App
```

Capitolo 5 – Introduzione ai componenti

*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.

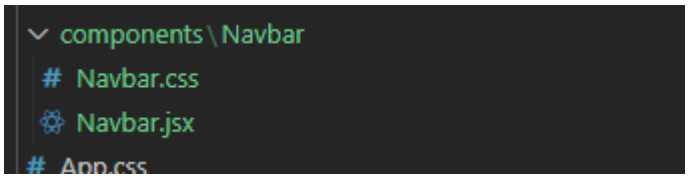


Figura 7 esempio di componente Navbar…

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

  return (
    <>
      <Navbar></Navbar>
      <div>
        <a href="https://vitejs.dev" target="_blank">
```

Figura 8 …inserito all'interno del componente primigenio

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**

```
return (
  <>···
  </>
);
```

4. Ogni tag element deve essere chiuso

```
<img /> Corretto

<img> Sbagliato
```

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

```
const x = 3000;
return (
  <>
    <img /> Corretto
    <h1>{x}</h1>
```

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

```
<h2 style={{ color: "blue" }}>Obaoba</h2>
```

*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.

Stilizzazione:

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

```
const styleImg = {
  color: "red",
  borderRadius: "30px",
};

return (
  <>
    <img alt="immagine" src={img} style={styleImg} /> Corretto
```

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

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

```
<h2 className="testo-red">Obaoba</h2>
```

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

```
const styleImg = {
  color: "red",
  borderRadius: x > 1000 ? "10px" : "30px",
};
```

- È possibile inserire classi dinamicamente

```
13    .box {
14      width: "30px";
15      height: "60px";
16    }
17
18    .blue {
19      background-color: blue;
20    }
21
22    .red {
23      background-color: red;
24    }
```

Figura 12 classi

```
<div className={`box ${x > 1000 ? "blue" : "red"}`}> x è {x}</div>
```

Figura 13 jsx

*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

```
<h3 className="text-3xl font-bold underline text-slate-800">dsdadas</h3>
```

Capitolo 9 – Props (proprietà dei componenti)

*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**

```jsx
import "./Card.css";

function Card(props) {
  const img = props.img;
  const title = props.title;
  const description = props.description;

  return (
    <>
      <div className="flex flex-col justify-center items-center m-2 bg-white text-slate-600 font-semibold w-1/3 rounded-md">
        <img
          alt="img"
          src={img}
          className=" w-full h-40 object-cover rounded-t-md"
        />
        <div className="p-2">
          <h2 className="text-2xl">{title}</h2>
          <p>{description}</p>
        </div>
      </div>
    </>
  );
}

export default Card;

```

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.

Il **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

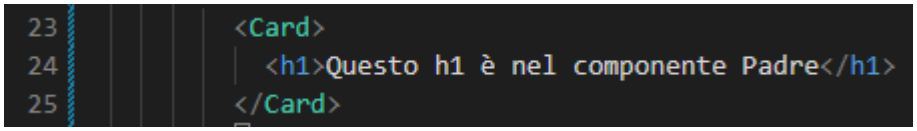
```
Card(props)
```

a

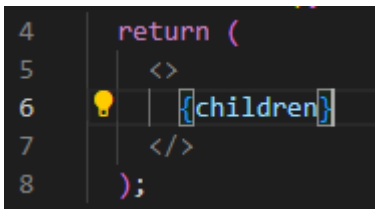
```
 Card({ title, img, description })
```

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

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



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



Capitolo 10 – Rendering condizionale

*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:

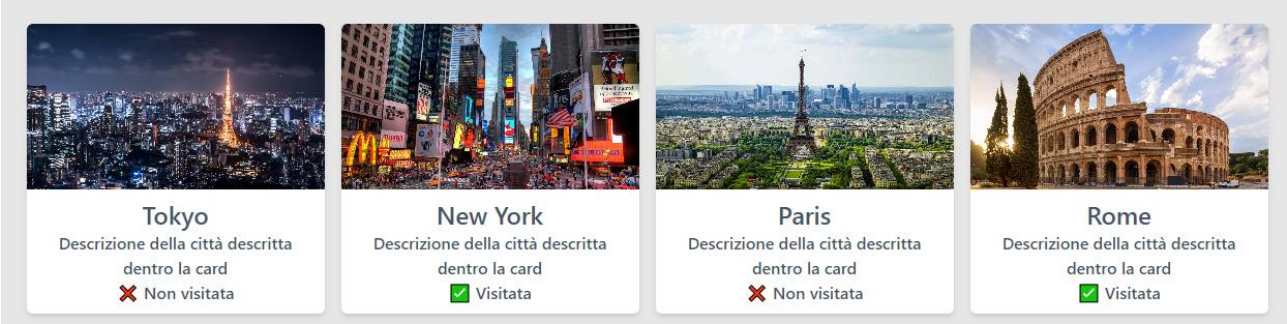
```
<Card
  title="Tokyo"
  description="Descrizione de
  img="https://www.gotokyo.or
  isVisited={false}></Card>
<Card
  title="New York"
  description="Descrizione de
  img="https://upload.wikimed
  isVisited={true}></Card>
<Card
```

Componente Figlio:

```
{isVisited ? <span>✅ Visitata</span> : <span>❌ Non visitata</span>}
```

Naturalmente è possibile definire a più livelli il rendering

Risultato:



Naturalmente gli approcci possibili…

```
const visitedLabel = isVisited ? "✅ Visitata" : "❌ Non visitata";
```

```
{visitedLabel}
```

…sono praticamente infiniti!

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

```
{isVisited && <span>✅ Visitata</span>}
{!isVisited && <span>❌ Non visitata</span>}
```

Capitolo 11 – Rendering di liste

*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.

```
19          {cities.map((city) => (
20            <Card
21              key={city.id}
22              title={city.title}
23              img={city.img}
24              description={city.description}
25              isVisited={city.isVisited}></Card>
26          ))}
```

La funzione effettua il **return** della card con le **props** innestate che verranno 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**

```
{cities
  .filter((city) => city.isVisited)
  .map((city) => (
    <Card
```

*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:

```
<button
  onClick={() => {
    console.log("aaa");
  }}>
  Clicca
</button>
```

Caso 2:

```
function handleClick() {
  console.log("bob");
}
return (
```

```
<button onClick={handleClick}>Clicca</button>
```

***Nota**:*

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

passing a function (correct)	calling a function (incorrect)
`<button onClick={handleClick}>`	`<button onClick={handleClick()}>`

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 , senza alcun clic. Questo perché JavaScript è all'interno di JSX {e} viene eseguito immediatamente.

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

passing a function (correct)	calling a function (incorrect)
`<button onClick={() => alert('...')}>`	`<button onClick={alert('...')}>`

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

*Figure 1 "It's part of the act the fifty **states**..." (Sufjan Stevens)*

"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>
```


```
1
2
3
4
5
>
```

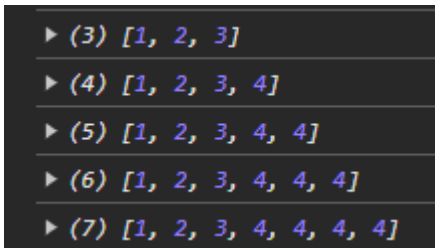
Es.2

```
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



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

```
...items,
```

Lift the State up!

Capitolo 14 – From Child to Father

Figure 2 "If you can fill the unforgiving minute With sixty seconds' worth of distance run, Yours is the Earth and everything that's in it, And—which is more—you'll be a Man, my son!" (from "if...", by J. R. Kipling)

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.

```
const [cities, addCities] = useState([
]);
```

2. 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)

```
const addCity = (city) => {
  addCities([...cities, city]);
};
```

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

```
{/* Form */}
<CardForm addCity={addCity}></CardForm>
```

4. 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

```
function CardForm({ addCity }) {
  const handleClick = () => {
    const newCity = {
      id: 5,
      isVisited: false,
      title: "Berna",
      description: "Descrizione della città descritta dentro la card",
      img: "https://www.bern.com/assets/images/7/altstadt2-884b8f17.jpg",
    };

    addCity(newCity);
  };
```

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

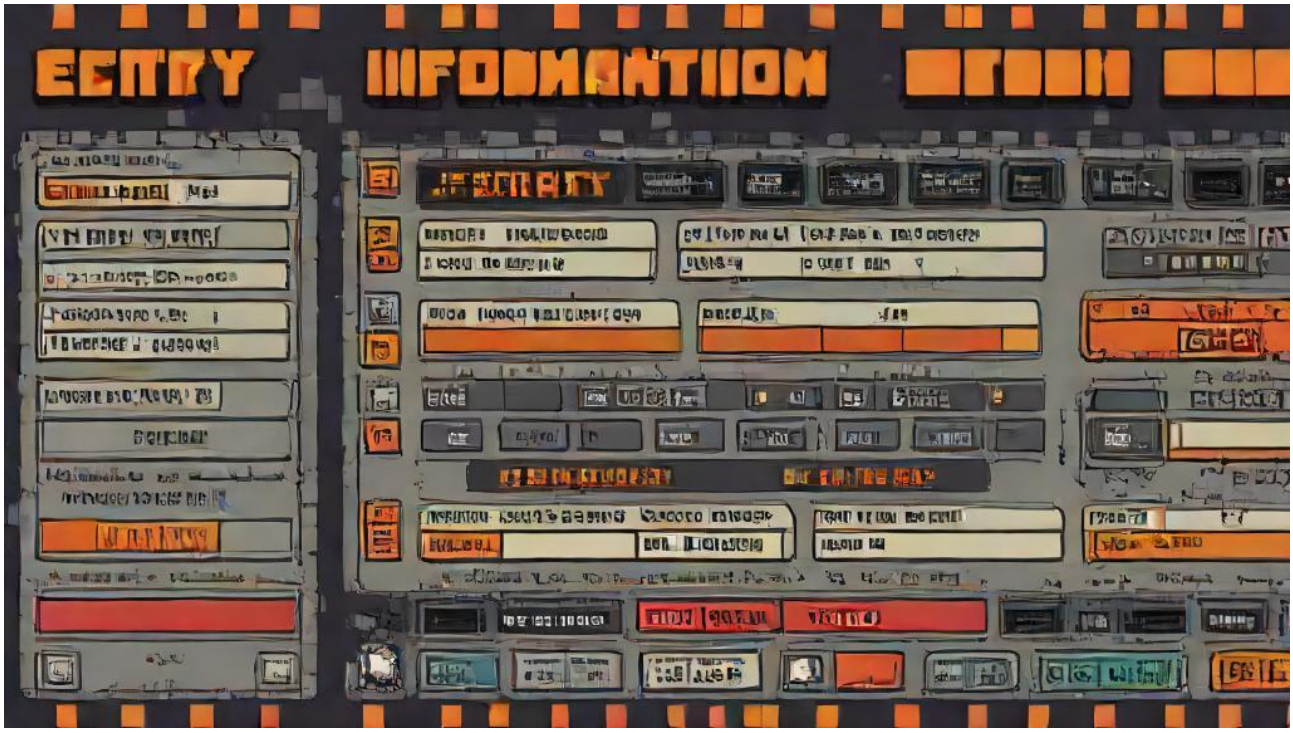


Figura 19 "...insert here the text..."

La gestione dei form è un elemento imprescindibile da un buono sviluppo in una applicazione.

Il flusso di creazione di un form è il seguente

1. Istanziamento di uno state che definisce l'oggetto form, dando una struttura ad oggetto a questo

```
const [formData, setFormData] = useState({
  id: 0,
  title: "",
  description: "",
  imgURL: "",
  isVisited: false,
});
```

2. Costruzione della view del form, identificando il name del campo, definendo il value come proprietà dell'oggetto formData, creado una funzione di gestione del valore ad ogni variazione di questo in tempo reale

```
<form
  className="flex flex-col justify-center items-center gap-2"
  onSubmit={handleSubmit}>
  <input ···
    placeholder="id"></input>
  <input
    type="text"
    name="title"
    className="rounded-md bg-slate-700 text-white p-2"
    placeholder="title"
    value={formData.title}
    onChange={handleInputChange}></input>
  <textarea ···
```

3. Acquisizione dell'evento, e destrutturazione dell'interezza del blocco, filtro per gestione del tipo di dato, text o checkbox e set dello state del campo in cambiamento inserendo il valore restituito in precedenza

```
17  const handleInputChange = (event) => {
18    // Acquisizione dell'evento, e destrutturazione dell'interezza del blocco
19    const { value, type, checked, name } = event.target;
20
21    // Filtro per gestione del tipo di dato, text o checkbox
22    const inputValue = type === "checked" ? checked : value;
23
24    // Set dello state del campo in cambiamento inserendo il valore restituito in pre
25    setFormData({
26      ...formData,
27      [name]: inputValue,
28    });
29  };
```

4. Impostazione dell'evento submit nel form

```
onSubmit={handleSubmit}>
```

5. Invio del submit, con set dello state delle città, evitando il refresh della pagina dopo il trigger di questo evento tramite il preventDefault()

```
31  const [cities, setCities] = useState(citiesImported);
32
33  const handleSubmit = (event) => {
34    event.preventDefault();
35
36    const city = {
37      id: Math.random(),
38      title: formData.title,
39      description: formData.description,
40      imgURL: formData.imgURL,
41      isVisited: formData.isVisited,
42    };
43
44    setCities([...cities, city]);
45  };
```

Capitolo 16 – React Developer Tool

[…]

Capitolo 17 – useEffect

*Figure 3 Butterfly **useEffect***

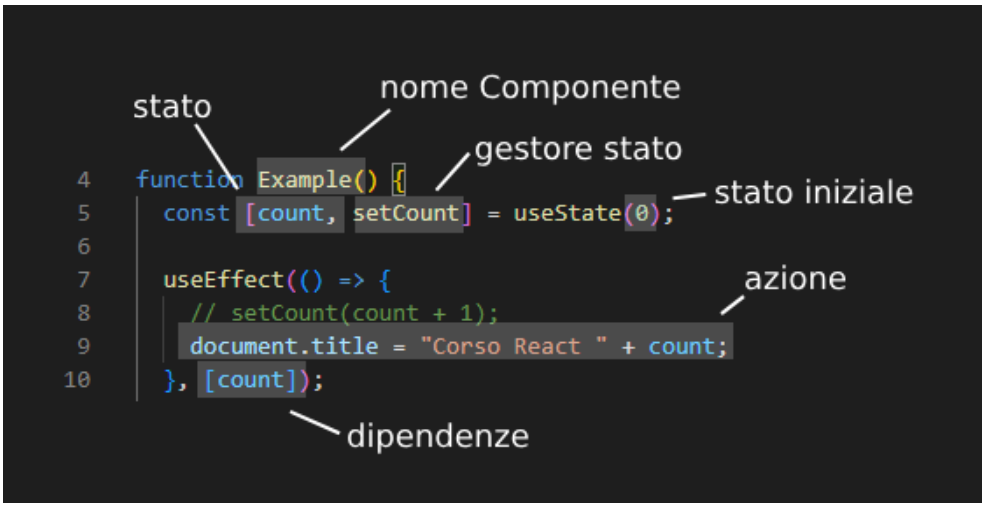
Lo **useEffect**, insieme allo useState, è uno degli hook più importanti di React che ti consente di sincronizzare un componente con un sistema esterno, gestendo quelli che sono tutti i side effect che esulano dal mero rendering a schermo.

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.

Struttura 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;
```

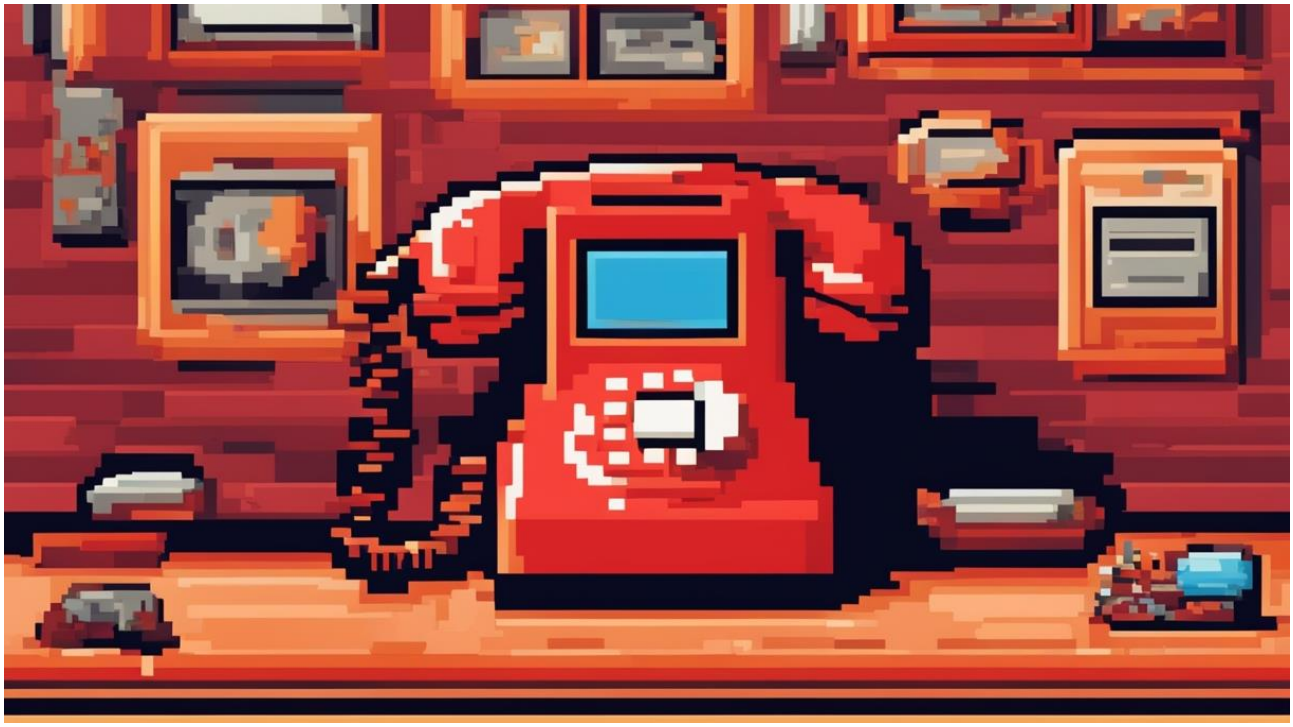


*Figura 20 So **call** me maybe...*

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

Figura 21 - Agent: "You are only human, Neo: "Hum...upgrades!

Lo **useReducer** è una evoluzione dello **useState**, permette una gestione più complessa dello state.

Oltre ad esso ci sono evoluzioni ulteriori che potenziano la gestione dello state, come ad esempio il **Context API** e **Redux**, quest'ultima una vera e propria libreria che permette una gestione globale dello state.

Ad es. creiamo un form da cui modificare un oggetto formState da manipolare tramire lo useReducer

```
10  // useReducer | dispatch ==> emanare un ordine
11  const [formState, dispatchFormState] = useReducer(formReducer, {
12    name: "",
13    email: "",
14  });
```

Come è possibile notare, oltre lo stato iniziale del form (che in questo Caso è un oggetto con prop. Name e email), abbiano una **funzione** (**formReducer**) con la quale manipoleremo lo state.

Le **best practices** consigliano come **nomenclatura: dispatchNameFunction** e **nameReducer**.

Dopo aver settato lo stato iniziale e lo useReducer, bisognerà costruire la view per la manipolazione del form.

```jsx
<label htmlFor="name">Name:</label>
<input
  type="text"
  id="name"
  name="name"
  value={formState.name}
  onChange={(event) =>
    handleFieldChange(event.target.name, event.target.value)
  }></input>
```

In essa setteremo i valori dei 2 campi di input, ponendoli come proprietà dell'interò oggetto form e la funzione **handleFieldChange**, la quale userà il **dispatchFormState** per gestire lo state.

```jsx
const handleFieldChange = (field, value) => {
  //                |              ACTION              |
  dispatchFormState({ type: "change_field", field, value });
};
```

La funzione userà il dispatch per richiamare la funzione di gestione (formReducer), la quale riceverà l'**ACTION** (ovvero il **tipo** di azione e i **parametri**).

Infine entrerà in gioco la funzione vera e propria di gestione, la quale, in base al tipo di azione, setterà lo state secondo i parametri passati:

```jsx
function formReducer(state, action) {
  switch (action.type) {
    case "change_field": {
      return { ...state, [action.field]: action.value };
    }
    case "reset_form": {
      return {
        name: "",
        email: "",
      };
    }
    default: {
      return state;
    }
  }
}
```

…dove lo **state** è quello passato inizialmente nell'useReducer…

```jsx
const [formState, dispatchFormState] = useReducer(formReducer, {
  name: "",
  email: "",
});
```

…mentre l'**action** l'oggetto passato nell'handleFieldChange!


useReducer Hook
[formState, dispatchFormState]
Initial State
{ name: "", email: "" }
formReducer
switch (action.type) {
case "change_field":
return { ...state, [action.field]: action.value };
case "reset_form":
return { name: "", email: "" };
default:
return state;
}
Dispatch Actions
dispatchFormState({ type: "change_field", field, value }
dispatchFormState({ type: "reset_form" })
Form Component
<form>
<input name="name" value={formState.name} ... />
<input name="email" value={formState.email} ... />
<button onClick={resetForm}>Reset Form</button>
<button onClick={sendForm}>Invia</button>
</form>
Event Handlers
const handleFieldChange = (field, value) => {
dispatchFormState({ type: "change_field", field, value})
};
const resetForm = () => {
dispatchFormState({ type: "reset_form" });
};
const sendForm = (e) => {
e.preventDefault();
console.log("formstate: ", formState);
};

Capitolo 20 – Context API

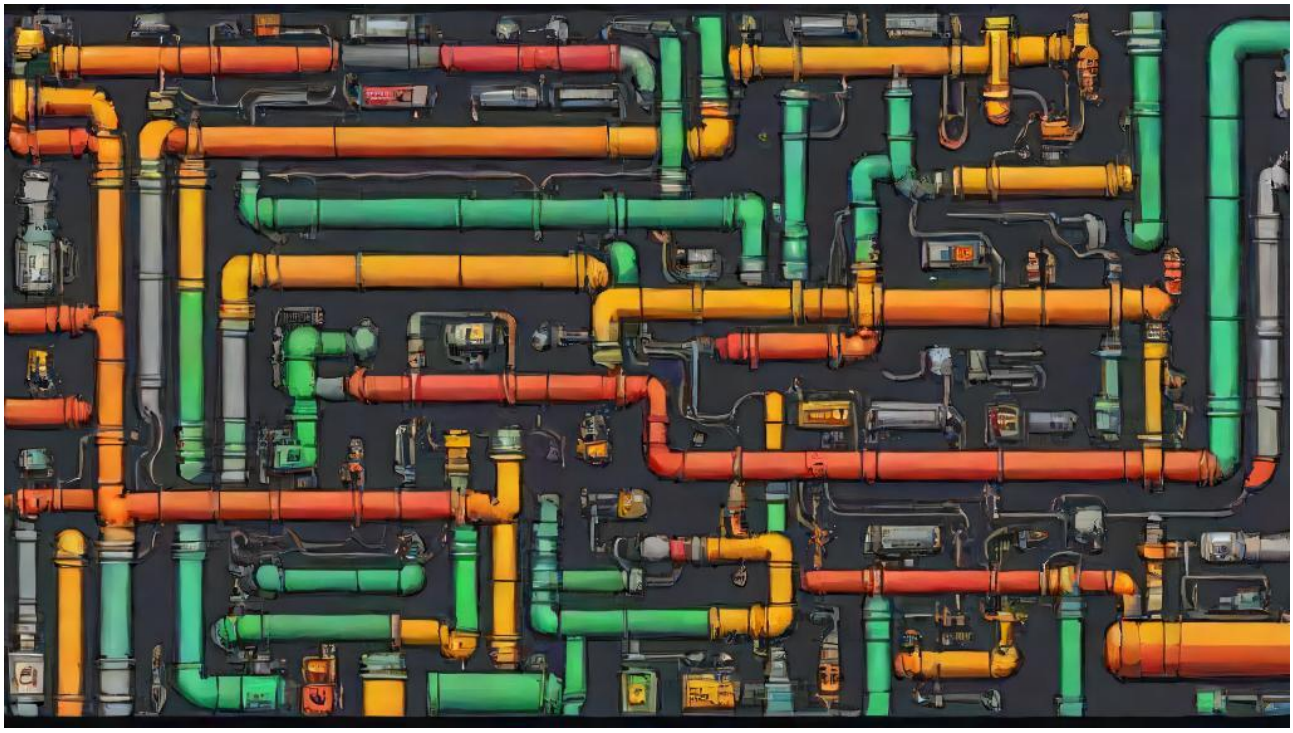
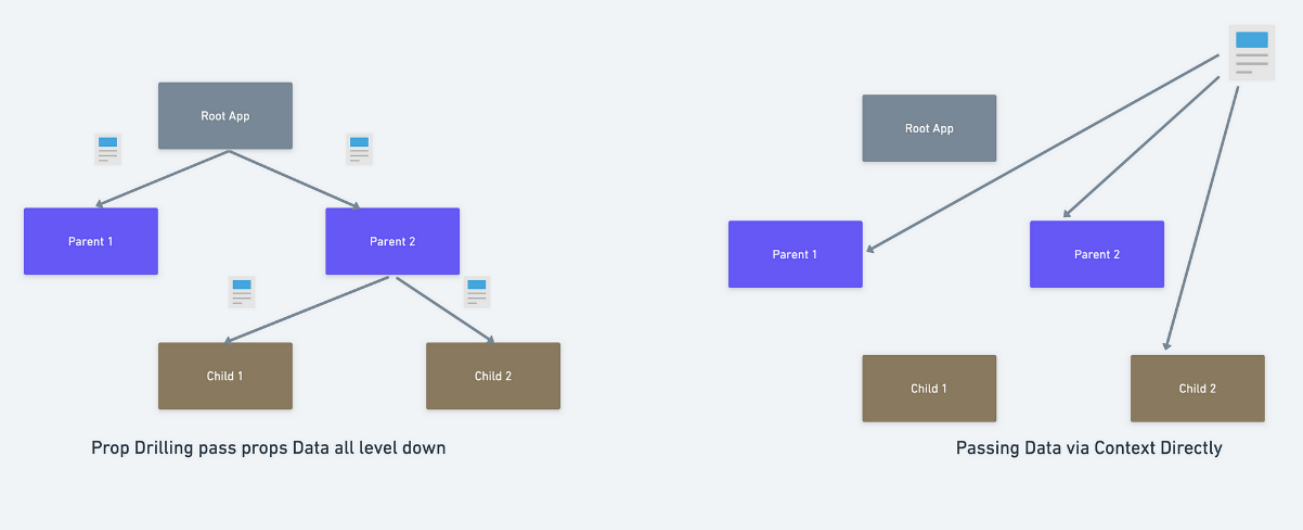


Figura 22 - Doc Brown: "Roads? Where We're Going, We Don't Need Roads"

Sistema che permette di gestire il **passaggio di stato** tra component evitando **prop drilling** o **emitter**.

Questo ricorda l'uso dei service in **Angular** per il passaggio di dati.



Il sistema comporta la creazione di **stores**, ovvero contesti dentro i quali è possibile **commerciare** dati tra componenti.

1. Innanzitutto, bisogna istanziare il **context** dentro cui passare il dato

```
1   import { createContext } from "react";
2
3   export const ProvaContext = createContext({});
```

2. Definire il **Provider** (fornitore) che definisce e instrada i valori da passare (in questo caso lo **state** count e setCount)

```
<ProvaContext.Provider value={{ count, setCount }}>
```

3. Importare il componente settare il context (**App.jsx**)

```
import { ProvaContext } from "./stores/EsempioDiContext";
```

4. Così da poterlo usare nei componenti dentro il provider (**Son.jsx**)

```
return (
  <ProvaContext.Provider value={{ count, setCount }}>
    {/* INTRO */}
    <div className="bg-white rounded-md m-6 p-6">···
    </div>
    <img
      className="w-1/2 h-1/3 object-cover rounded-xl m-auto"···
      src={orn}
    />

    {/* COMPONENTE CHE USA LO USECONTEXT PER OTTENERE LO STATE */}
    <Son></Son>
```

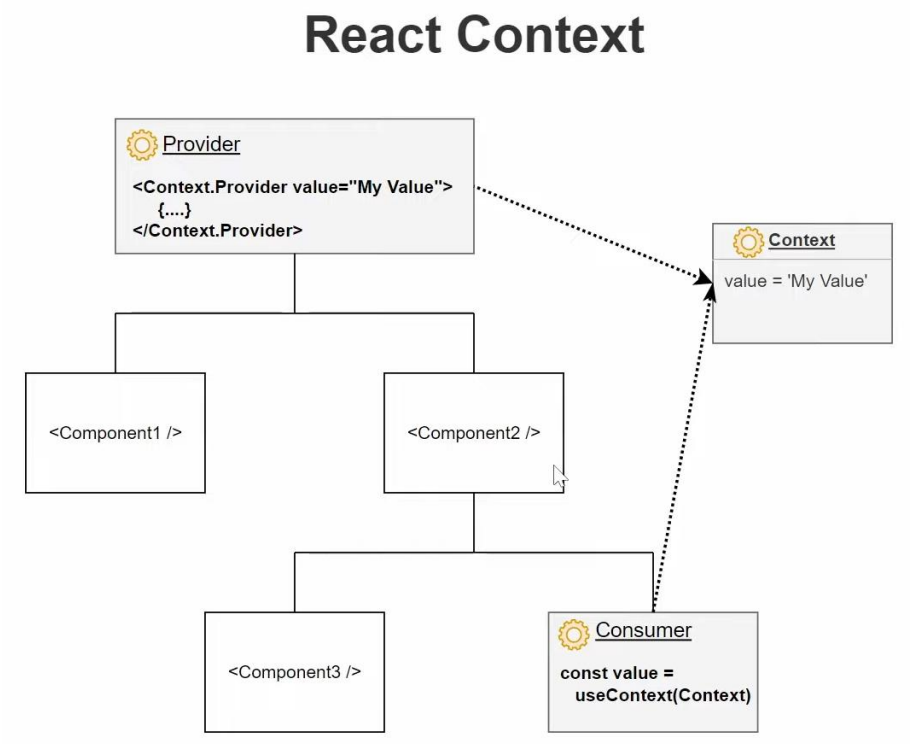
5. Dentro Son useremo lo state e il setState del context inizializzato in App.jsx (nota, è possibile. Oltre ad essre più corretto, inizializzare dentro il ProvaContext i nostri state, e poi inviarli nei componenti inoculati dentro il nostro provider)

```
import { useContext } from "react";
import { ProvaContext } from "../../stores/EsempioDiContext";
import "./Son.css";

function Son() {
  const { count, setCount } = useContext(ProvaContext);
  return (
    <>
      <h1 className="bg-blue-600 p-2 rounded-md m-6 text-
        Sono il son
      </h1>
      <button
        onClick={() => {
          setCount(count + 1);
        }}>
        Aumenta count da context: {count}
      </button>
    </>
  );
}

export default Son;
```

6.



Il **Provider** mette a disposizione un **context** a cui i **componenti interessati** possono attingere per ottenere lo state.

[…]

*Figura 23 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";
```

3. 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>
);
```

```
import { Link } from "react-router-dom";
function Navbar() {
  return (
    <>
      <ul className="flex flex-row justify-center items-center gap-2 m-
        <li className="text-white bg-blue-600 rounded-md p-2">
          <Link to={"/"}>Home</Link>
        </li>
        <li className="text-white bg-blue-600 rounded-md p-2">
          <Link to={"/cards"}>Cards</Link>
        </li>
        <li className="text-white bg-blue-600 rounded-md p-2">
          <Link to={"/about"}>About</Link>
        </li>
      </ul>
    </>
  );
}
```

Tramite l'uso del **Link** del **router-dom** sarà possibile di gestire il reindirizzamento (in realtà un **rendering condizionale**) presso la **page** corrispondente!

Per poter visualizzare un unico item è possibile usare lo useParams, che identifica e seleziona il parametro dell'url e filtra le città per visualizzare l'item corrispondente

```
import { useParams } from "react-router-dom";
```

```
const { idCard } = useParams();
if (idCard) {
  for (let i = 0; i < citiesImported.length; i++) {
    if (citiesImported[i].id.toString() === idCard.toString()) {
      title = citiesImported[i].title;
      img = citiesImported[i].img;
      description = citiesImported[i].description;
      isVisited = citiesImported[i].isVisited;
    }
  }
}
```

Per gestione **implicitamente** un routing, è possibile usare l'**Outlet**, un gestore di react-dom che permette di visualizzare internamente alla pagina un item filtrato

```
import { Outlet } from "react-router-dom";
```

…importandolo…

E **inserendolo** nella pagina, così da **renderizzare internamente** l'item

```
<Outlet />
</>
```

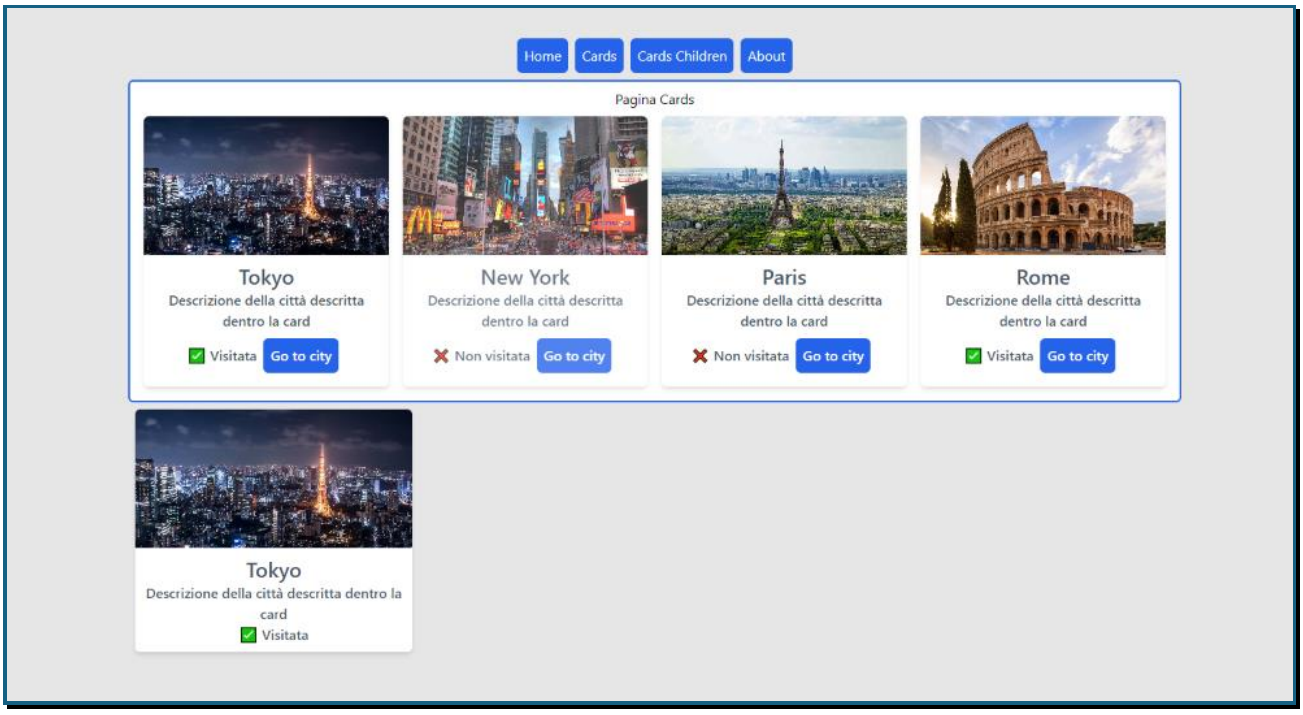


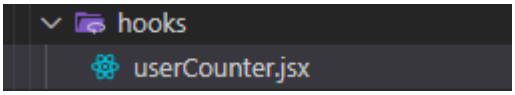
Figura 24 Zorg : If you want something done, do it yourself! (cit. The fifth element)

React permette di creare hooks personalizzati, ma perché?

Uno dei motivi sarebbe quello di definire, in specifici contesti, della **logica**, come ad esempio delle **chiamate http**, in un unico **servizio** personalizzato.

Esempio: Creiamo un hook che, all'accesso di una route, ci dice quanto tempo un utente è rimasto in una pagina

1. Costruiamo il nostro custom hook



2. Definiamo la sua logica

```
import { useEffect } from "react";
function useCounter() {
  useEffect(() => {
    let boba = 0;
    const interval = setInterval(() => {
      boba += 1;
      console.log("| ", boba);
    }, 1000);

    return () => {
      clearInterval(interval);
    };
  }, []);
}

export default useCounter;

```

Spiegazione: abbiamo costruito un hooks che utilizza uno useEffect che definisce un counter al rendering del componente About, il main component presenza nella route **'/about'**, questo attiverà un intervallo che, allo smontaggio del componente eseguirà un **cleanup** (rappresentato dal return che effettua il blocco dell'interval) dello **useEffect**.

3. Importiamolo nel componente in cui vogliamo utilizzarlo

```

import useCounter from "../../hooks/userCounter";

function About() {
  useCounter();
  return (
    <>
```

In fine: il custom hook ha una enorme potenza, perché permette di creare una funzione (con integrazione di hook built-in in react) da riutilizzare in tutti i componenti che ne necessitano.
