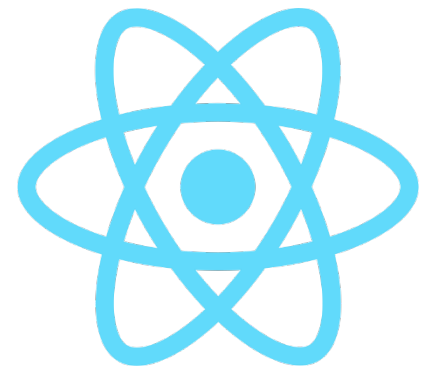


# React



# Contatore

Prima di andare avanti andiamo a creare un nuovo componente «contatore» per poi vedere alcune nuove funzionalità di React. Per prima cosa creiamo il componente Contatore e dichiariamo una variabile usando useState e passiamo come valore 0, ed all'interno del nostro componente stampiamo il valore della variabile e aggiungiamo 2 bottoni : aumenta e diminuisci.

# Bootstrap

Prima di continuare importiamo le librerie di bootstrap all'interno del nostro progetto. Il modo più semplice è scaricare le librerie da riga di comando :

```
npm install - -save bootstrap
```

Dopo di ch  all'interno del file index.js fare l'import con :

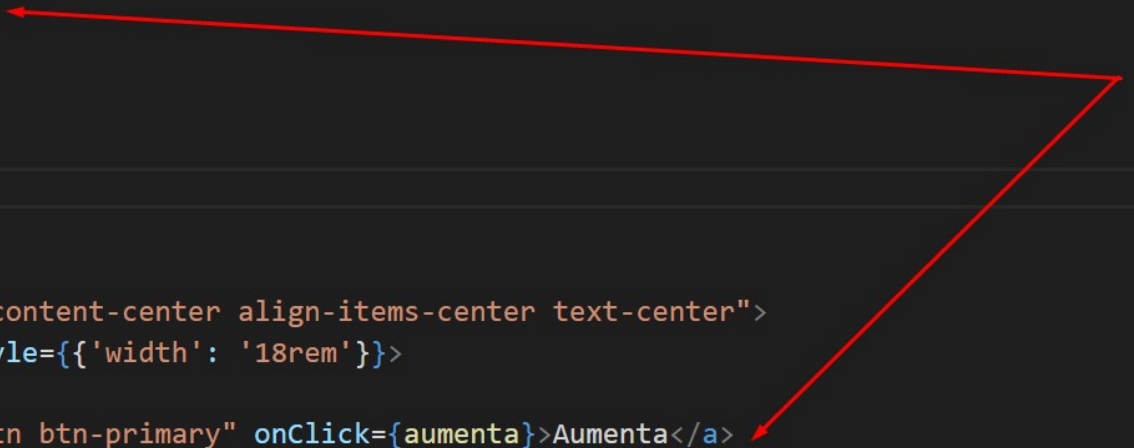
```
import 'bootstrap/dist/css/bootstrap.css';
```

```
const Contatore = () => {  
  const [contatore, setContatore] = useState(0);  
  
  return (  
    <div className="container">  
      <div className="row justify-content-center align-items-center text-center">  
        <div className="card" style={{'width': '18rem'}}>  
          <h1>{contatore}</h1>  
          <a href="#" class="btn btn-primary">Aumenta</a>  
          <a href="#" class="btn btn-secondary">Diminuisci</a>  
        </div>  
      </div>  
    </div>  
  )  
}
```

Possiamo ora sviluppare le nostre funzioni in due modi differenti ma che fanno la stessa cosa : il primo modo è mettendo la funzione `setContatore` direttamente nell'evento `onClick` tramite una arrow function.

Il secondo modo, quello classico che conosciamo, dove facciamo più passaggi ma il codice risulta più leggibile, cioè tramite una funzione separata. Ricordiamo che la funzione `setContatore` prende un parametro in **ingresso che rappresenta il valore attuale** del contatore :

```
const Contatore = () => {  
  
  const [contatore, setContatore] = useState(0);  
  
  const aumenta = () => {  
    setContatore(valoreAttuale => {  
      console.log(valoreAttuale);  
      return valoreAttuale + 1;  
    })  
  }  
  
  return (  
    <div className="container">  
      <div className="row justify-content-center align-items-center text-center">  
        <div className="card" style={{'width': '18rem'}}>  
          <h1>{contatore}</h1>  
          <a href="#" class="btn btn-primary" onClick={aumenta}>Aumenta</a>  
          <a href="#" class="btn btn-secondary" onClick={() => setContatore(contatore - 1)}>Diminuisci</a>  
        </div>  
      </div>  
    </div>  
  )  
}
```



# useState : return value vs functional

Quando chiamo la funzione useState abbiamo detto che mi torna il valore ed una funzione set. Questa funzione set abbiamo detto che serve per poter gestire ed aggiornare in modo autonomo il valore passato a useState. Ad esempio :

```
const [contatore1, setContatore1] = useState(0);  
let contatore2 = 0;
```

Ripetiamo che è la stessa cosa, nel primo caso sto usando la funzione `useState` nel secondo caso sto valorizzando direttamente la variabile.

Quindi qual è la differenza tra i due ? Che `useState` è una funzione che torna il suo valore ed in più torna una **funzione che gestisce lo stato** della variabile. Quindi se ad esempio cambio il valore della variabile tramite la sua funzione di ritorno, viene aggiornata automaticamente nel componente (e ricordiamo pure che `useState` può essere usata solo all'interno di un componente)



```
const [contatore1, setContatore1] = useState(0);
let contatore2 = 0;

const aumenta = () => {
  setContatore1(valoreAttuale => {
    console.log(valoreAttuale);
    return valoreAttuale + 1;
  })
}

const aumenta2 = () => {
  contatore2++;
  console.log(contatore2);
}

const diminuisci2 = () => {
  contatore2--;
  console.log(contatore2);
}

return (
  <div className="container">
    <div className="row justify-content-center align-items-center text-center">
      <div className="card" style={{'width': '18rem'}}>
        <h2>Contatore 1 : {contatore1}</h2>
        <h2>Contatore 2 : {contatore2}</h2>
        <a href="#" class="btn btn-primary" onClick={aumenta}>Aumenta Cont1</a>
        <a href="#" class="btn btn-secondary" onClick={()=>setContatore1(contatore1 - 1)}>Diminuisci Cont1</a>
        <a href="#" class="btn btn-primary" onClick={aumenta2}>Aumenta Cont2</a>
        <a href="#" class="btn btn-secondary" onClick={diminuisci2}>Diminuisci Cont2</a>
      </div>
    </div>
  </div>
)
```

Una volta vista la differenza tra valorizzare una variabile tramite `useState` e valorizzarla normalmente con il suo valore, vediamo quando una `useState` cambia valore tramite la sua funzione `set` che può farlo in 2 modi : tramite il valore o tramite il ritorno di una funzione.

Per fare un esempio andiamo a vedere la differenza sul bottone Aumenta e Diminuisce. Su aumenta cambiamo direttamente il valore, su diminuisce lo cambiamo tramite il ritorno di funzione, detto anche **functional return**.

```
const [contatore1, setContatore1] = useState(0);
```


```
const aumenta = () => {  
  setContatore1(contatore1+1);  
  console.log(contatore1);  
}
```

```
const diminuisci = () => {  
  setContatore1(valoreAttuale => {  
    console.log(valoreAttuale);  
    return valoreAttuale - 1;  
  })  
}
```

Se proviamo ora il nostro codice effettivamente funziona correttamente in tutte e due i modi. Qual è quindi la differenza tra usare la funzione set **passando direttamente il nuovo valore** oppure usare una **arrow function** che automaticamente prende in ingresso il valore attuale della variabile ?

La differenza è che il **functional return tiene traccia del valore** e lo aggiorna in maniera corretta. Per capirlo modifichiamo il nostro esempio ed aggiungiamo la funzione setTimeout cioè il cambio valore avviene dopo 2 secondi e non subito :

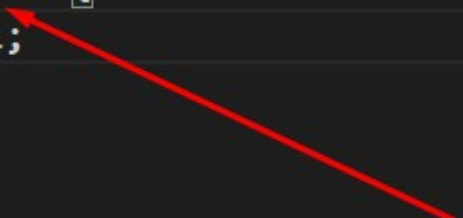
```
const Contatore = () => {  
  
  const [contatore1, setContatore1] = useState(0);  
  
  const aumenta = () => {  
    setTimeout(function() {  
      setContatore1(contatore1+1);  
      console.log(contatore1);  
    }, 2000)  
  }  
}
```



In questo caso ad ogni click cambia il valore, ma ogni 2 secondi. Quindi se clicco dopo 2 secondi non ci sono problemi. Ma cosa succede se clicco velocemente per 10 volte ? Succede che non prende tutti e 10 i click in quanto non rimane traccia di tutto quello che è successo mentre attendeva i 2 secondi.

Questo problema viene risolto e viene gestito automaticamente dal **functional return** di useState in questo modo :

```
const aumenta = () => {  
  setTimeout(function(){  
    setContatore1(valoreAttuale => {  
      return valoreAttuale + 1;  
    });  
    console.log(contatore1);  
  }, 2000)  
}
```



Per questo motivo normalmente tutti gli `useState` vengono gestiti con una functional return.

Possiamo usare le functional return anche per implementare dei controlli. Per esempio potremmo volere che il numero non aumenti oltre il 5. In questo caso potremmo fare come segue

```
const aumenta = () => {  
  setContatore((oldValue) => {  
    if (oldValue + 1 === 5) {  
      return oldValue;  
    }  
    return oldValue + 1;  
  });  
};
```

# Esercizio Appuntamenti

Creare un'app con una lista di appuntamenti.  
Gli appuntamenti vengono caricati da una lista di oggetti.

```
{  
  id: 4,  
  nome: "Rebecca",  
  stato: "Lorem ipsum"  
  img: «url jps»,  
}
```

L'app farà visualizzare e cancellare gli appuntamenti.



# useEffect

**useEffect** è un Hook che è responsabile di gestire tutte le azioni che avvengono al di fuori del componente.

Per fare un esempio andiamo a creare un nuovo componente ed importiamo lo `useEffect` così come abbiamo fatto per `useState` ed anche in questo caso è utilizzabile solo dentro un componente :

```
import React, { useEffect } from 'react'

const EsempioUseEffect = () => {

  useEffect(() => {
    console.log('ho chiamato una useEffect');
  })

  return (
    <div>
      <h1>useEffect</h1>
    </div>
  )
}

export default EsempioUseEffect
```

La **prima regola** che possiamo evincere da questo esempio è che `useEffect` viene chiamato DOPO che viene fatto il render del componente, praticamente viene chiamato ogni volta che usiamo il nostro componente.

Questo possiamo verificarlo subito andando a stampare dopo dello `useEffect` un altro log e vediamo che invece lo stampa prima :

```
import React, { useEffect } from 'react'

const EsempioUseEffect = () => {

  useEffect(() => {
    console.log('ho chiamato una useEffect');
  })

  console.log('sono al di fuori di useEffect');

  return (
    <div>
      <h1>useEffect</h1>
    </div>
  )
}

export default EsempioUseEffect
```

► XHR finished loading: GET "<http://localhost:3000/79f6b5e...hot-update.json>".

sono al di fuori di useEffect

ho chiamato una useEffect

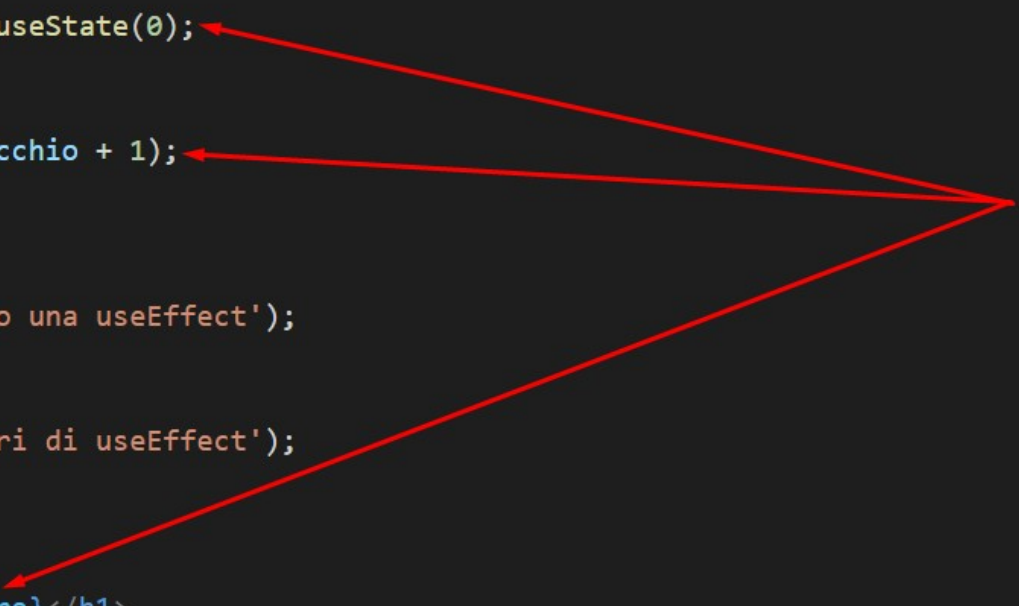
>

Per fare un esempio andiamo a cambiare il titolo della pagina utilizzando l'oggetto document di javascript (**document.title**).

Ricordiamoci che useEffect viene chiamato ogni volta che il nostro componente subisce un render, quindi se modifichiamo qualcosa al nostro componente verrà automaticamente eseguito anche la funzione useEffect (questo grazie alle useState) :

Riprendiamo l'esempio precedente dove con un click e l'utilizzo di useState vado ad incrementare un numero :

```
const EsempioUseEffect = () => {  
  
  const [valore, setValore] = useState(0);  
  
  const aumenta = () => {  
    setValore(vecchio => vecchio + 1);  
  };  
  
  useEffect(() => {  
    console.log('ho chiamato una useEffect');  
  });  
  
  console.log('sono al di fuori di useEffect');  
  
  return (  
    <div>  
      <h1>useEffect {valore}</h1>  
      <button onClick={aumenta}>Aumenta</button>  
    </div>  
  )  
}  
  
export default EsempioUseEffect
```



# useEffect 15

Aumenta

The screenshot shows the Chrome DevTools Console with the 'Console' tab selected. The left sidebar shows a list of messages: 38 messages, 38 user messages, 0 errors, 3 warnings, and 35 info. The main console area displays a log of messages from 'EsempioUseEffect.js'. The messages are grouped by similar messages in the console. The log shows a sequence of 'sono al di fuori di useEffect' and 'ho chiamato una useEffect' messages. A red arrow points from the text 'useEffect 15' to the 15th log entry, which is 'sono al di fuori di useEffect'.

Message	File
sono al di fuori di useEffect	EsempioUseEffect.js:15
ho chiamato una useEffect	EsempioUseEffect.js:12
sono al di fuori di useEffect	EsempioUseEffect.js:15
ho chiamato una useEffect	EsempioUseEffect.js:12
sono al di fuori di useEffect	EsempioUseEffect.js:15
ho chiamato una useEffect	EsempioUseEffect.js:12
sono al di fuori di useEffect	EsempioUseEffect.js:15
ho chiamato una useEffect	EsempioUseEffect.js:12
sono al di fuori di useEffect	EsempioUseEffect.js:15
ho chiamato una useEffect	EsempioUseEffect.js:12

All'interno degli Hooks possiamo inserire anche le condizioni ricordando che verranno eseguite solo dopo il render :

```
useEffect(() => {  
  console.log('ho chiamato una useEffect');  
  if(valore < 1){  
    document.title = "Nessun valore";  
  }else{  
    document.title = "C'è qualcosa..."  
  }  
});
```

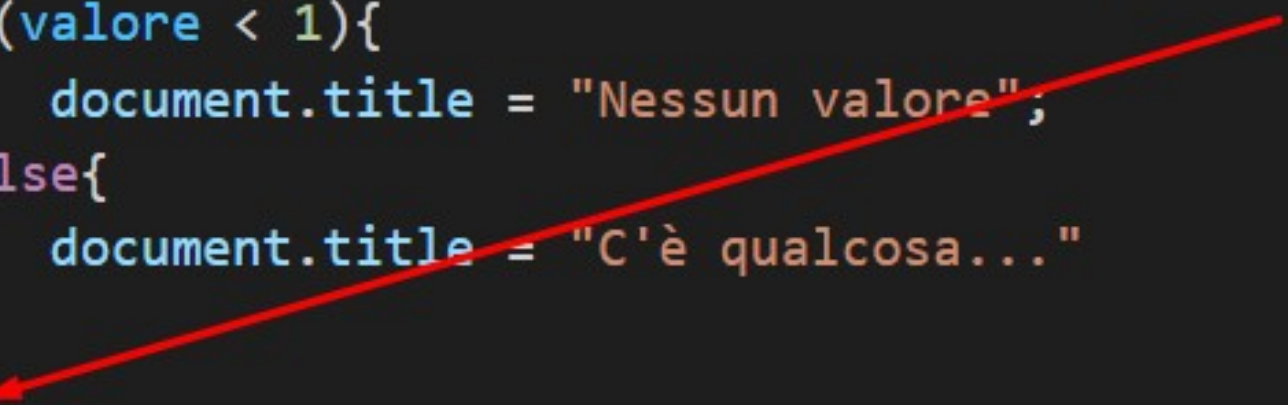


# useEffect secondo parametro

Di default il nostro useEffect verrà eseguito ad ogni render del nostro componente. Ma c'è un modo per decidere quando eseguire il nostro useEffect, e cioè attraverso il secondo parametro che opzionalmente accetta useEffect che è un array.

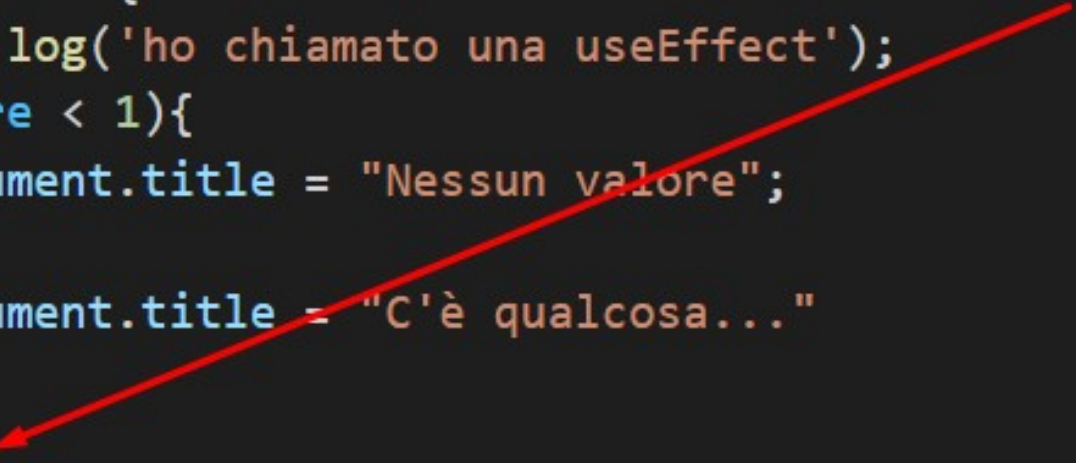
Se lascio **l'array vuoto** sto dicendo che il nostro useEffect deve essere eseguito una sola volta :

```
useEffect(() => {  
  console.log('ho chiamato una useEffect');  
  if(valore < 1){  
    document.title = "Nessun valore";  
  }else{  
    document.title = "C'è qualcosa..."  
  }  
},[]);
```



Oppure posso passare un parametro del componente, serve ad indicare che alla variazione di quel parametro useEffect verrà eseguito :

```
useEffect(() => {  
  console.log('ho chiamato una useEffect');  
  if(valore < 1){  
    document.title = "Nessun valore";  
  }else{  
    document.title = "C'è qualcosa..."  
  }  
},[valore]);
```



Niente impedisce, nel rispetto del buon senso, di avere e chiamare molteplici "use effect". Questo scenario non è affatto improbabile, anzi è molto comune. Potrei avere, ad esempio, un "useEffect" che si attiva ogni volta che si aggiorna il valore "street value", e un altro che necessita di essere chiamato sempre. Un ulteriore "use effect" potrebbe essere invocato da me ogni volta che lo ritengo necessario, al quale potrei passare una funzione.

Potrei avere una funzione "Arrow" che fa un semplice log, esclusivamente quando un certo valore non è impostato. È possibile avere un "use effect" che si attiva solo quando c'è una variazione di valore e un altro che si attiva indipendentemente. Entrambi potrebbero essere attivati ogni volta che si verifica un render, a seguito di una variazione di valore.

Se avessi questa configurazione, vedremmo nella console solo il secondo "use effect" che si attiva dopo il primo render, il quale attiva anche il primo "use effect". Dopo ciò, solo il secondo "use effect" verrà invocato. Questo concetto è fondamentale.

```
const [value, setValue] = React.useState(0);

const aumenta = () => {
  setValue((oldValue) => oldValue + 1);
};


const funzio = () => {
  console.log("Secondo use Effect");
};

useEffect(() => {
  console.log("Eccomi, sono use Effect");
  if (value < 1) {
    document.title = `Nessun Messaggio`;
  } else {
    document.title = `Nuovo Messaggi: ${value}`;
  }
}, []);

useEffect(funzio);
```

Un altro aspetto cruciale di "use effect" è la sua capacità di restituire una "**cleanup function**", che ci permette di eseguire operazioni di pulizia. Questa funzione viene definita ogni volta che si esegue "use effect" e viene eseguita prima del successivo render o della successiva invocazione di "use effect".

```
useEffect(() => {  
  console.log('ho chiamato una useEffect');  
  if(valore < 1){  
    document.title = "Nessun valore";  
  }else{  
    document.title = "C'è qualcosa..."  
  }  
  
  return( () => { console.log("Eseguo un po di pulizia") })  
},[valore]);
```



Se immaginiamo di impostare un event listener all'interno del nostro "use effect", che si attiva ogni volta che c'è un aggiornamento di uno specifico stato o elemento, è fondamentale rimuovere quel listener prima della prossima invocazione di "use effect". Altrimenti, se questo stato cambiasse frequentemente, potremmo ritrovarci con l'evento settato numerose volte, compromettendo le performance dell'applicazione. Con la "clean up function", possiamo garantire che l'event listener precedente venga rimosso prima di impostarne uno nuovo.

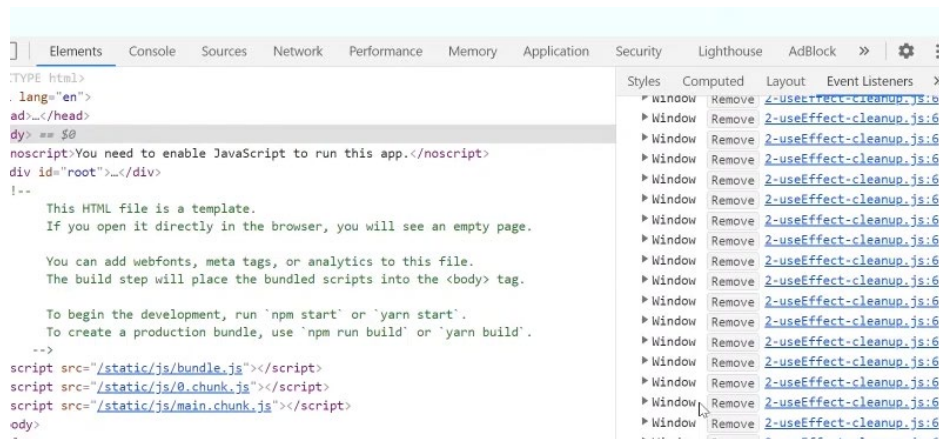
Vediamo l'esempio più in dettaglio di come usare il return dello

```
const CleanUp = () => {  
  
  const [size, setSize] = useState(window.innerWidth);  
  
  const dimensioneFinestra = () => {  
    setSize(window.innerWidth);  
  };  
  
  /*  
  * Ad ogni resize della pagina Aggiorna la state size e rimuove l'event Listener  
  */  
  useEffect(() => {  
    window.addEventListener("resize", dimensioneFinestra);  
  
  });  
  return (  
    <div  
      className='container w-75 col-6 offset-3 bg-white shadow p-4 mx-auto'  
      style={{ textAlign: "center" }}  
    >  
      <h1> {size} </h1>  
    </div>  
  );  
};
```



Nel componente, abbiamo una semplice visualizzazione di questa dimensione utilizzando Bootstrap. Ogni volta che la dimensione della finestra cambia, la nostra "clean up function" all'interno dell'hook `useEffect` ci permette di aggiornare lo stato e di rimuovere l'event listener precedentemente aggiunto, garantendo un comportamento ottimale e previene l'aggiunta di listener multipli.

Se osserviamo nella console, vedremo che ogni volta che ridimensioniamo la finestra, viene registrato l'evento. Tuttavia, grazie alla nostra "clean up function", viene sempre mantenuto un solo listener attivo, garantendo prestazioni ottimali e comportamento atteso.



```
useEffect(() => {  
  window.addEventListener("resize", dimensioneFinestra);  
  return () => {  
    window.removeEventListener("resize", dimensioneFinestra);  
  };  
});
```

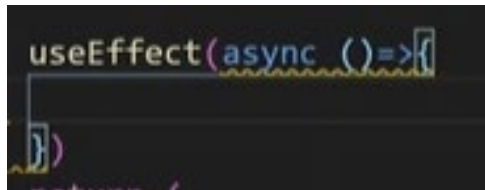
In situazioni più complesse, con molti componenti che possono attaccare event listener in diverse parti dell'applicazione, avere una "clean up function" è buona prassi. Fornisce un controllo completo su ciò che accade, su ciò che viene rimosso e su ciò che rimane attivo. Ci sono molte altre situazioni in cui questo metodo è fondamentale.

# useEffect - Data Fetching

Vediamo come usare useEffect per fetchare dati con fetch o axios.

Creiamo uno useState photos. Andreamo a prendere i dati da jsonplaceholder.

Per fare questo creiamo la funzione async getData e passeremo al nostro useEffect. Dobbiamo creare una funzione a parte perché non **possiamo rendere lo useEffect async**. Il codice dell'immagine quindi non può essere usato



```
useEffect(async () => {  
  // ...  
})
```

# useEffect - Data Fetching

Di seguito la chiama della funzione con lo useEffect

```
const FetchComponent = () => {  
  
  const [photos, setPhotos] = useState([]);  
  
  const getData = async () => {  
    //const response = await axios.get(url);  
    //setPhotos(response.data);  
    const photos= await fetch(url).then(ris=>ris.json())  
    setPhotos(photos.slice(0,20));  
  };  
  
  //Fetcha i dati solo al primo render  
  useEffect(() => {  
    getData();  
  }, []);  
  return (  

```

# Json Server

Vediamo ora come lavorare su dati presenti sul server in modo tale che la nostra applicazione interagisca con un backend.

Usiamo json server che simula un vero e proprio server. Se creiamo un tag nel json che si chiama post ci espone questi metodi

## Routes

Based on the previous `db.json` file, here are all the default routes. You can also add **other routes** using `--routes`.

### Plural routes

```
GET    /posts
GET    /posts/1
POST   /posts
PUT    /posts/1
PATCH /posts/1
DELETE /posts/1
```

### Singular routes

```
GET    /profile
POST   /profile
PUT    /profile
PATCH /profile
```

Si installa un package gli diamo la struttura che vogliamo avere dei nostri dati e da quel momento simula un db su cui possiamo eseguire post e get.

<https://www.npmjs.com/package/json-server>

Dalla console bash lanciamo ***npm i -g json-server*** e installiamolo globalmente. Nel frattempo trasformiamo la nostra struttura dati in clienteService in un json

```
const clienti=[...];  
JSON.stringify(clienti);
```

Dopo di che creiamo un db.json nella root della nostra applicazione, incolliamo il json e mettiamo in ascolto il server su questo file eseguendo la seguente istruzione

```
json-server --watch db.json
```

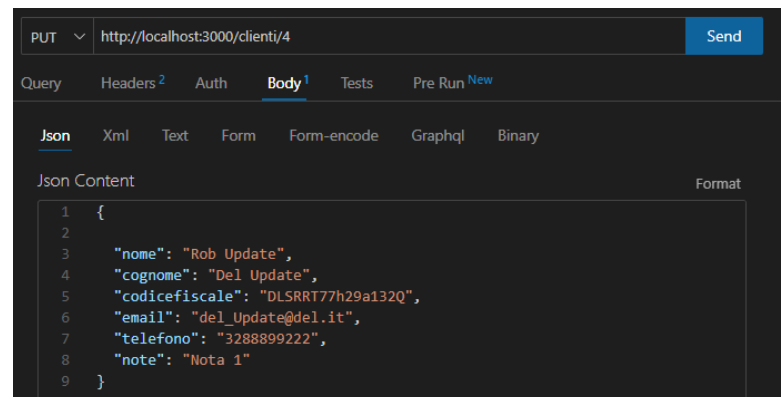
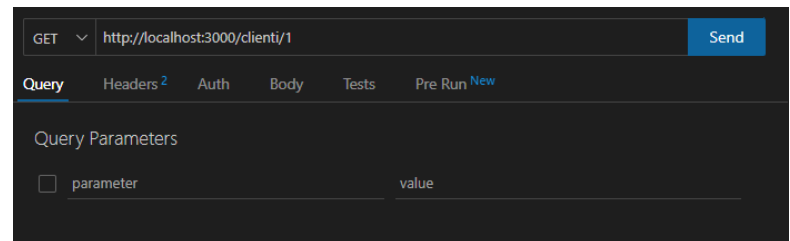
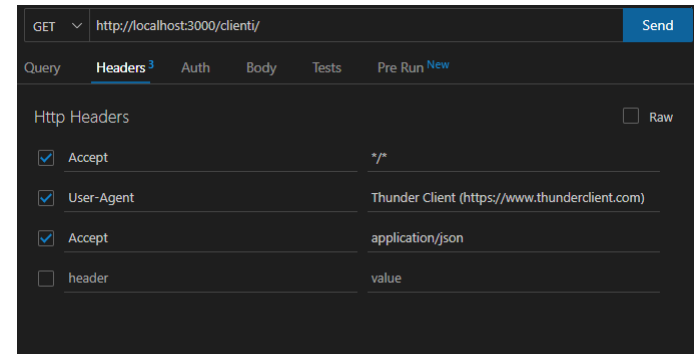
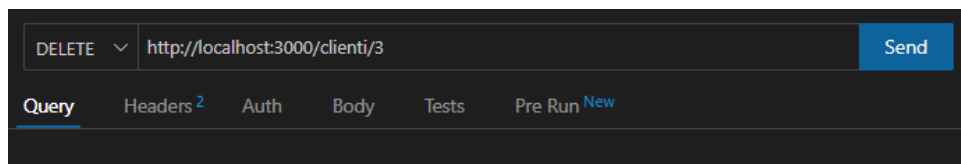
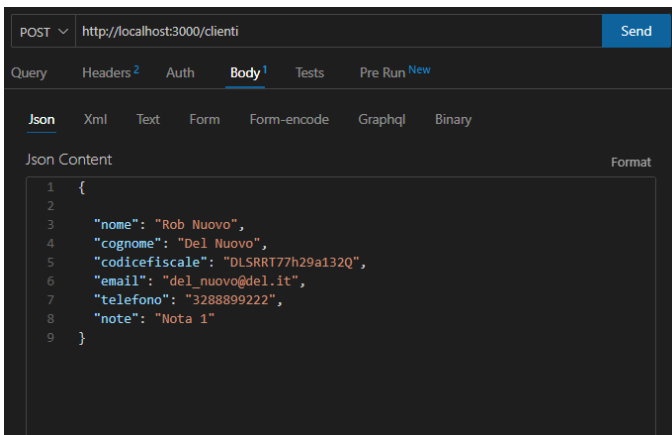
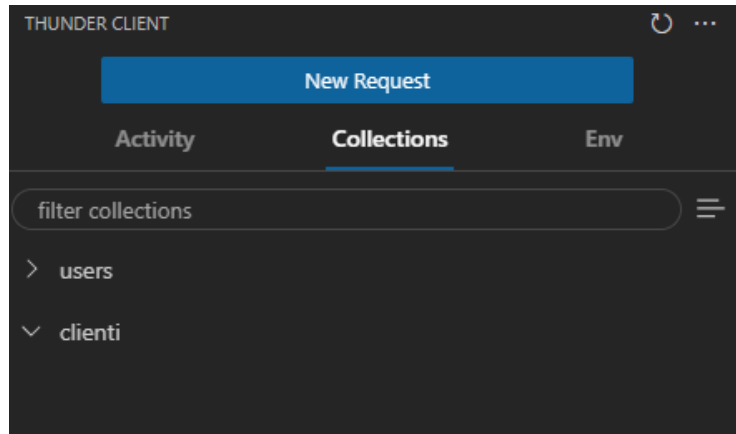
Installiamo una nuova estensione di VC thunder client per testare le API Rest e configuriamo le nostre chiamate rest

```
[{"id":1,"nome":"Rob1","cognome":"Del","codicefiscale":"DLSRRT77h29a132Q","email":"del1@del.it","telefono":"3288899222","note":"Nota 1"}, {"id":2,"nome":"Rob2","cognome":"Del","codicefiscale":"DLSRRT77h29a132Q","email":"del2@del.it","telefono":"3288899222","note":"Nota 2"}, {"id":3,"nome":"Rob3","cognome":"Del","codicefiscale":"DLSRRT77h29a132Q","email":"del3@del.it","telefono":"3288899222","note":"Nota 3"}, {"id":4,"nome":"Rob4","cognome":"Del","codicefiscale":"DLSRRT77h29a132Q","email":"del4@del.it","telefono":"3288899222","note":"Nota 4"}]
```

```
{
  "clienti": [
    {
      "id": 1,
      "nome": "Rob1",
      "cognome": "Del",
      "codicefiscale": "DLSRRT77h29a132Q",
      "email": "del1@del.it",
      "telefono": "3288899222",
      "note": "Nota 1"
    },
    {
      "id": 2,
      "nome": "Rob2",
      "cognome": "Del",
      "codicefiscale": "DLSRRT77h29a132Q",
      "email": "del2@del.it",
      "telefono": "3288899222",
      "note": "Nota 2"
    },
    {
      "id": 3,
```



Creiamo una nuova collection clienti e poi le request per creare il post, get, put e delete.



# Esercizio 1

Riprendere l'esercizio sulla biblioteca e modificarlo per usare lo `useState` e lo `useEffect`.

Usa `json-server` per creare un db di libri e le chiamate rest.

## Esercizio 2

Realizzare un componente che gestisce un cronometro. Impostare una variabile a zero e mostrarla a video. Aggiungere 3 bottoni : Start Stop e Reset.

