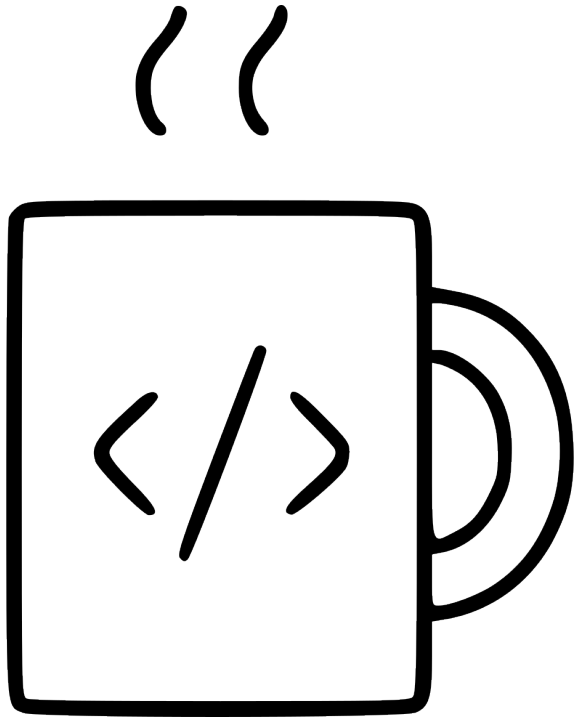
Umberto Emanuele

React – Giorno 3

Novembre 2023

State (stato) e Lifecycle (ciclo di vita) in un componente React

https://it.reactjs.org/docs/state-and-lifecycle.html



PAUSA

Ci vediamo alle ore 11.10

Convertire una Funzione in una Classe

Puoi convertire un componente funzione come `Clock` in una classe in cinque passaggi:

1. Crea una classe ES6, con lo stesso nome, che estende `React.Component`.
2. Aggiungi un singolo metodo vuoto chiamato `render()`.
3. Sposta il corpo della funzione all'interno del metodo `render()`.
4. Sostituisci `props` con `this.props` nel corpo del metodo `render()`.
5. Rimuovi la dichiarazione della funzione rimasta vuota.

Il metodo render viene invocato ogni volta che si verifica un aggiornamento

La gestione dello **Stato** di un Componente

Props: Dati di sola lettura che non cambiano

Stato di un componente: Lo **state** (o stato) è simile alle **props**, ma è privato e completamente controllato dal componente. Lo stato del componente può cambiare e il componente si aggiorna al cambiamento dello stato.

State e Lifecycle

2. Aggiungi un costruttore di classe che assegna il valore iniziale di `this.state`:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Ciao, mondo!</h1>
        <h2>Sono le {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Qual è lo stato in un'app di React?

Si può pensare a esso come un singolo oggetto JavaScript che rappresenta tutti i dati nella vostra app.

Lo stato può essere definito su qualsiasi componente, ma se si desidera condividere lo stato tra i componenti allora è meglio definirlo sul componente di primo livello.

Lo stato può quindi essere tramandato ai componenti figlio e averne accesso come richiesto.

State e Lifecycle

```jsx
import React from 'react';
import './App.css';
import Clock from './components/Clock';

function App() {
  return (
    <div className="App">
      <Clock timezone='0' country='Rome' />
    </div>
  );
}

export default App;
```

```jsx
class Clock extends Component {
    constructor(){
        super();
        this.state = {
            date: new Date()
        };
    }

    render(){

        const tempo = this.state.date.getTime() + this.props.timezone * 3600 * 1000;
        const data = new Date(tempo);

        return <h2>In {this.props.country} is {data.toLocaleTimeString()} </h2>

    }

}

export default Clock;
```

React

Come aggiornare uno stato: Ci sono 2 modi diversi

```
this.setState((prevState) => ({
  count:prevState.count + 1;
}))
```

Se lo stato da aggiornare dipende dallo stato precedente

Se non c'è nessuna dipendenza dallo stato precedente

```
this.setState({
  count: 7;
})
```

Non Modificare lo Stato Direttamente

Per esempio, questo codice non farebbe ri-renderizzare un componente:

```
// Wrong
this.state.comment = 'Hello';
```

Devi invece utilizzare `setState()`:

```
// Correct
this.setState({comment: 'Hello'});
```

State e Lifecycle

Poiché `this.props` e `this.state` potrebbero essere aggiornate in modo asincrono, non dovresti basarti sul loro valore per calcolare lo stato successivo.

Ad esempio, questo codice potrebbe non riuscire ad aggiornare correttamente il contatore:

```
// Sbagliato
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

```
// Giusto
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

State e Lifecycle

Gli Aggiornamenti di Stato Vengono Applicati Tramite Merge

Quando chiami `setState()`, React effettua il merge dell'oggetto che fornisci nello state corrente.

Ad esempio, il tuo state potrebbe contenere molte variabili indipendenti:

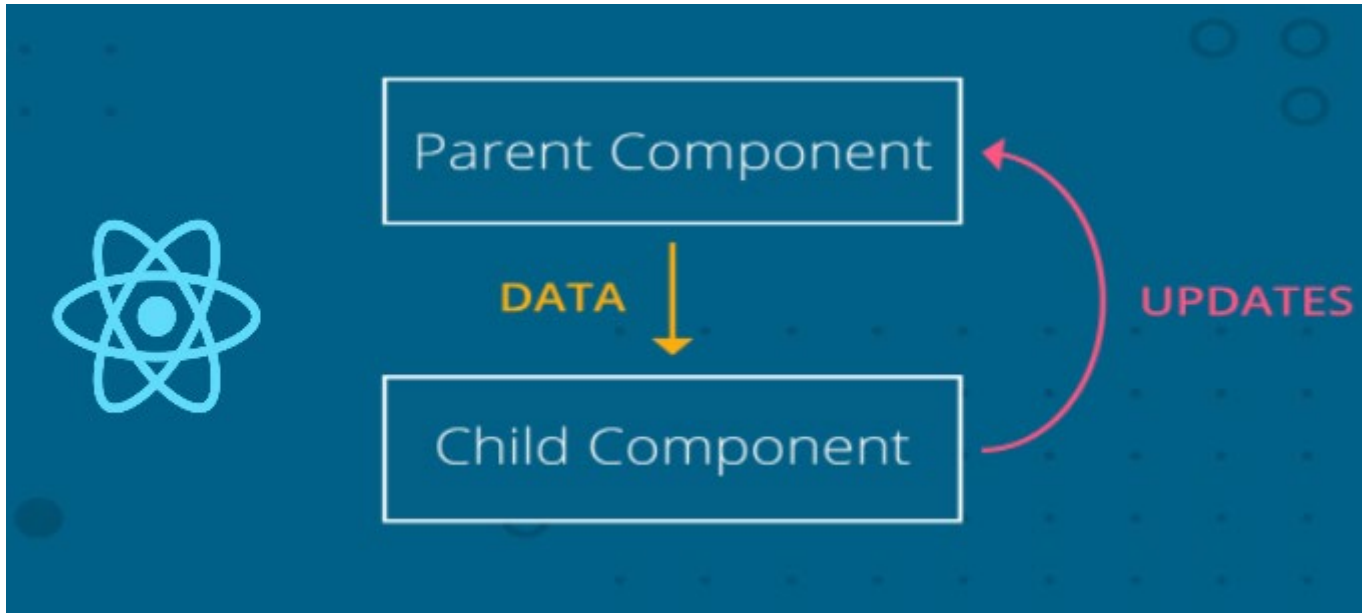
```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

A questo punto puoi aggiornarle indipendentemente con invocazioni separate del metodo `setState()`:

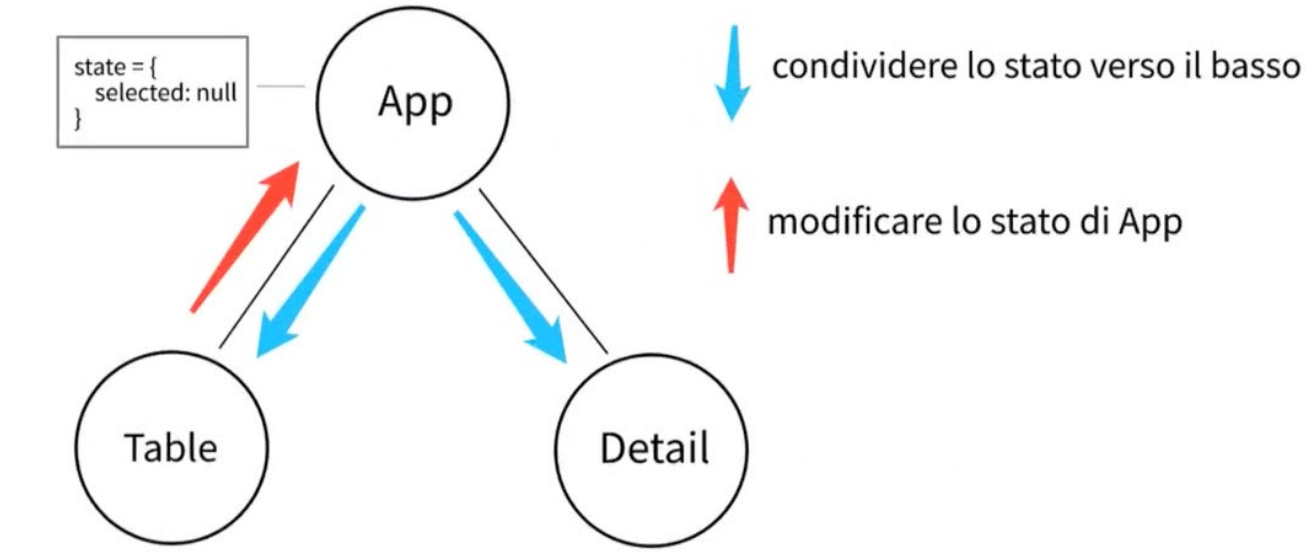
```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

React

I Dati Fluiscono Verso il Basso(definito flusso di dati “top-down” o “unidirezionale”)



State e Lifecycle



Elevare lo stato in App e condividere
'selected' con i children Table e Detail

I Dati Fluiscono Verso il Basso

(definito flusso di dati “top-down” o “unidirezionale”)

Né i componenti genitori né i componenti figli possono sapere se un certo componente è “**stateful**” o “**stateless**” (cioè se è dotato o meno di stato) e non dovrebbero preoccuparsi del fatto di essere definiti come funzione o come classe.

Questa è la ragione per cui lo stato è spesso definito locale o incapsulato. Esso non è accessibile a nessun componente a parte quello a cui appartiene.

Un componente potrebbe decidere di passare il suo stato ai componenti figli sotto forma di **props**.

Lifecycle (ciclo di vita) in un componente React

https://it.reactjs.org/docs/state-and-lifecycle.html

Lifecycle di un Componente

componentWillMount() viene eseguito immediatamente **prima** che il componente sia **inserito** nel DOM.

componentDidMount() viene eseguito immediatamente **dopo** che il componente venga **inserito** nel DOM.

componentWillUnMount() viene eseguito immediatamente **prima** che il componente venga **rimosso** dal DOM.

componentWillReceiveProps() viene eseguito ogni volta che il componente riceve nuove props.

Lifecycle di un Componente

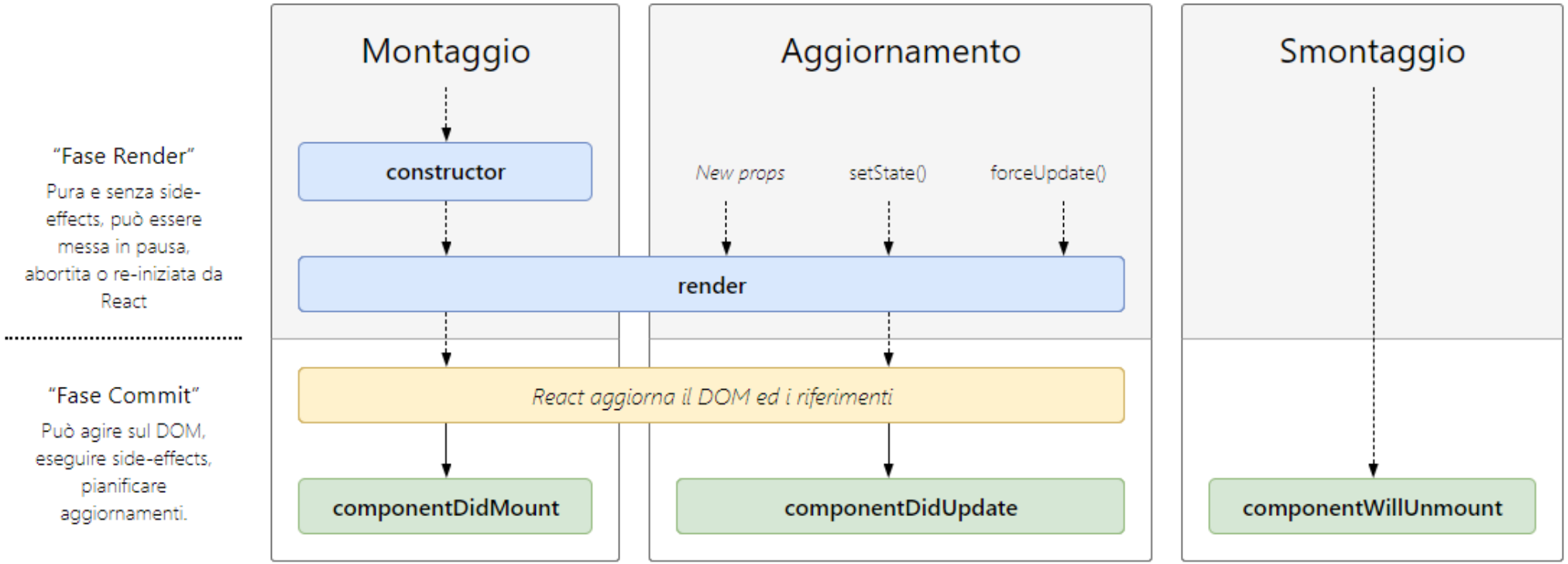
Nelle applicazioni con molti componenti, è molto importante rilasciare le risorse occupate dai componenti quando questi vengono distrutti.

Il metodo **componentDidMount()** viene eseguito dopo che l'output del componente è stato renderizzato nel DOM.

Questo è definito "**mounting**" ("montaggio") in React.

Mentre se il componente dovesse mai essere rimosso dal DOM, React invocherebbe il metodo del lifecycle **componentWillUnmount().**

Lifecycle di un Componente



https://it.reactjs.org/docs/state-and-lifecycle.html

http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

State e Lifecycle

Mounting (Montaggio)

Quando un'istanza di un componente viene creata e inserita nel DOM, questi metodi vengono chiamati nel seguente ordine:

- `constructor()`
- `static getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

https://it.reactjs.org/docs/react-component.html#static-getderivedstatefromprops

State e Lifecycle

constructor()

```
constructor(props)
```

Se non hai bisogno di inizializzare lo stato del componente e di non effettuare il bind di metodi, non c'è bisogno di implementare un costruttore per il tuo componente React.

Il costruttore di un componente React è chiamato prima che il componente venga montato. Quando implementi il costruttore in una sottoclasse di `React.Component`, dovresti chiamare `super(props)` prima di ogni altra istruzione. In caso contrario, `this.props` rimarrebbe *undefined* nel costruttore, il che può causare bug.

Di solito in React i costruttori sono utilizzati solamente per due scopi:

- Inizializzare lo stato locale assegnando un oggetto a `this.state`.
- Effettuare il bind dei gestori di eventi (event handlers) ad una istanza.

```
constructor(props) {
  super(props);
  // Non chiamre this.setState() qui!
  this.state = { contatore: 0 };
  this.gestoreClick = this.gestoreClick.bind(this);
}
```

Il costruttore è l'unico punto in cui puoi assegnare direttamente un valore a `this.state`. In tutti gli altri metodi, devi invece utilizzare `this.setState()`.

State e Lifecycle

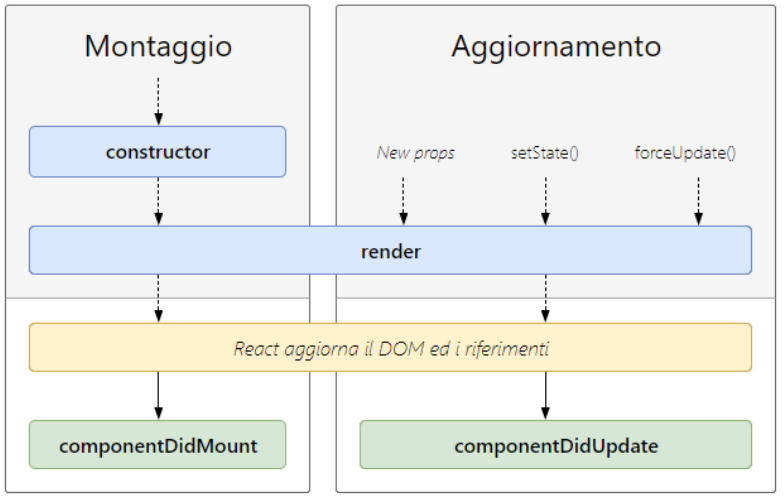
`render()`

```
render()
```

Il metodo `render()` è l'unico metodo obbligatorio in un componente classe.

Il suo compito è esaminare `this.props` e `this.state` e restituire in output uno dei seguenti tipi:

- **Elementi React.** Creati solitamente utilizzando JSX. Ad esempio, `<div />` e `<MyComponent />` sono elementi React che indicano a React di renderizzare rispettivamente un nodo del DOM e un altro componente definito dall'utente.
- **Array e "fragments" (frammenti).** Ti consentono di restituire in output più di un elemento. Leggi la documentazione sui fragments per maggiori informazioni.
- **Portali**. Ti consentono di renderizzare i figli del componente in un sottoalbero del DOM diverso da quello in cui si trova il componente. Leggi la documentazione sui portali per maggiori informazioni.
- **Stringhe e numeri.** Sono renderizzati come nodi di testo nel DOM.
- **Booleani o `null`**. Non renderizzano niente. (Esistono soprattutto per supportare il pattern `return test && <Figlio />`, in cui `test` è booleano.)



State e Lifecycle

React

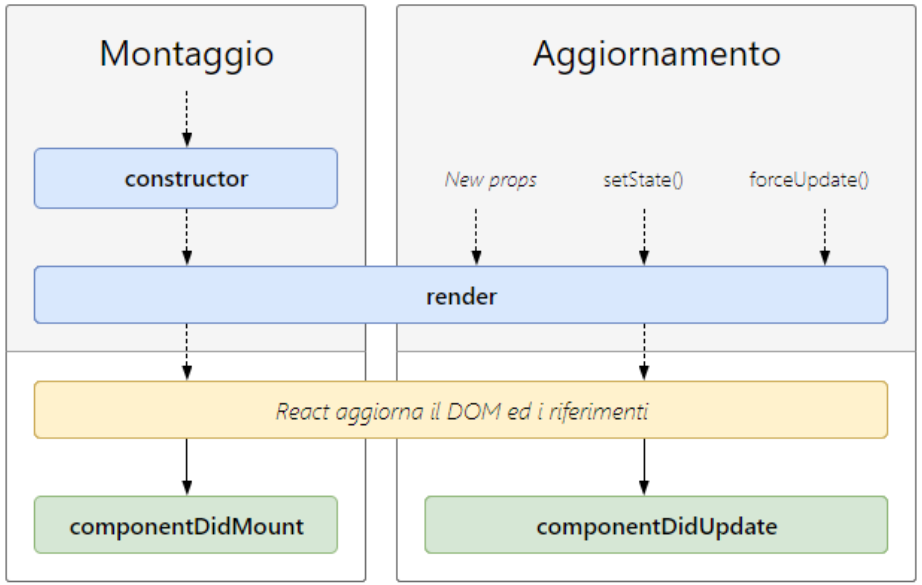
componentDidMount()

```
componentDidMount()
```

`componentDidMount()` è invocato dopo che il componente è montato (cioè inserito nell'albero del DOM). Le logiche di inizializzazione che dipendono dai nodi del DOM dovrebbero essere dichiarate in questo metodo. Inoltre, se hai bisogno di caricare dei dati chiamando un endpoint remoto, questo è un buon punto per istanziare la chiamata.

Questo metodo è anche un buon punto in cui creare le sottoscrizioni. Se lo fai, non scordarti di cancellare in `componentWillUnmount()` tutte le sottoscrizioni create.

Puoi **chiamare `setState()` immediatamente** in `componentDidMount()`. Farlo scatenerà una richiesta di rendering aggiuntiva, che però verrà gestita prima che il browser aggiorni lo schermo. Questo garantisce che l'utente non visualizzi lo stato intermedio anche se il metodo `render()` viene chiamato due volte. Utilizza questo pattern con cautela in quanto spesso causa problemi di performance. Nella maggior parte dei casi, dovresti poter assegnare lo stato iniziale del componente nel `constructor()`. Tuttavia, potrebbe essere necessario utilizzare questo pattern in casi come i popup o i tooltip, in cui hai bisogno di misurare un nodo del DOM prima di renderizzare qualcosa che dipende dalla sua posizione o dalle sue dimensioni.



State e Lifecycle

Aggiornamento

Un aggiornamento può essere causato da cambiamenti alle props o allo state. Quando un componente viene ri-renderizzato, questi metodi sono chiamati nel seguente ordine:

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- **`render()`**
- `getSnapshotBeforeUpdate()`
- **`componentDidUpdate()`**

https://it.reactjs.org/docs/react-component.html#static-getderivedstatefromprops

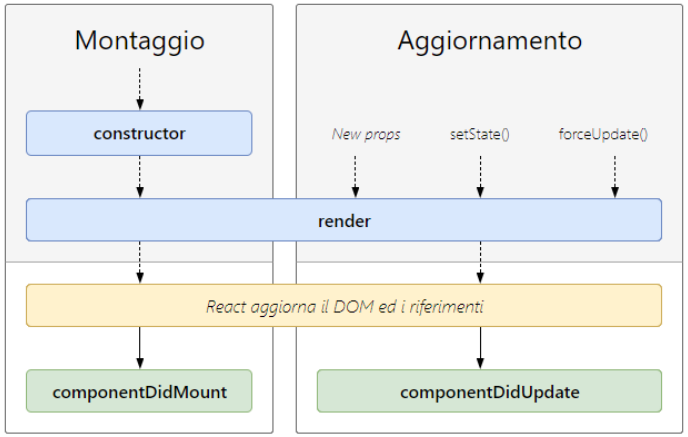
State e Lifecycle

componentDidUpdate()

```
componentDidUpdate(propsPrecedenti, statePrecedente, snapshot)
```

`componentDidUpdate()` è invocato immediatamente dopo che avviene un aggiornamento del componente. Non viene chiamato per la renderizzazione iniziale.

Utilizza questo metodo come un'opportunità di effettuare operazioni sul DOM dopo che il componente si è aggiornato, oppure per effettuare richieste di rete dopo aver comparato i valori attuali delle props con quelli passati (e.g. una richiesta di rete potrebbe non essere necessaria se le props non sono cambiate).



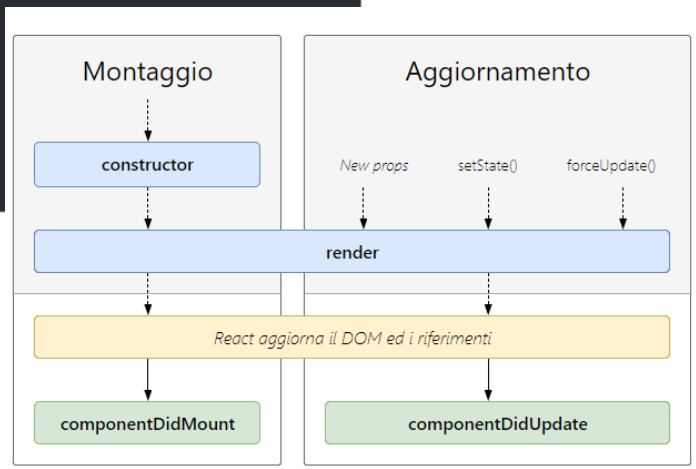
State e Lifecycle

Puoi chiamare **setState()** nel **componentDidUpdate()** ma tieni presente che deve essere racchiuso in una condizione, altrimenti causerai un ciclo infinito.

componentDidUpdate()

```
componentDidUpdate(prevProps, prevState, snapshot)
```

```
componentDidUpdate(prevProps) {
  // Typical usage (don't forget to compare props):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```



Unmounting (Smontaggio)

Quando un componente viene rimosso dal DOM, viene chiamato questo metodo:

- `componentWillUnmount()`

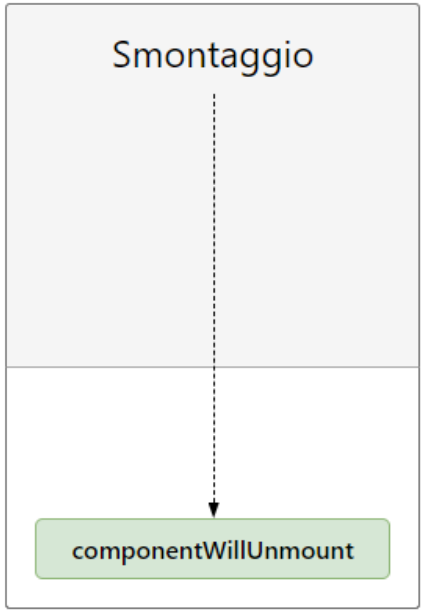
https://it.reactjs.org/docs/react-component.html#static-getderivedstatefromprops

componentWillUnmount()

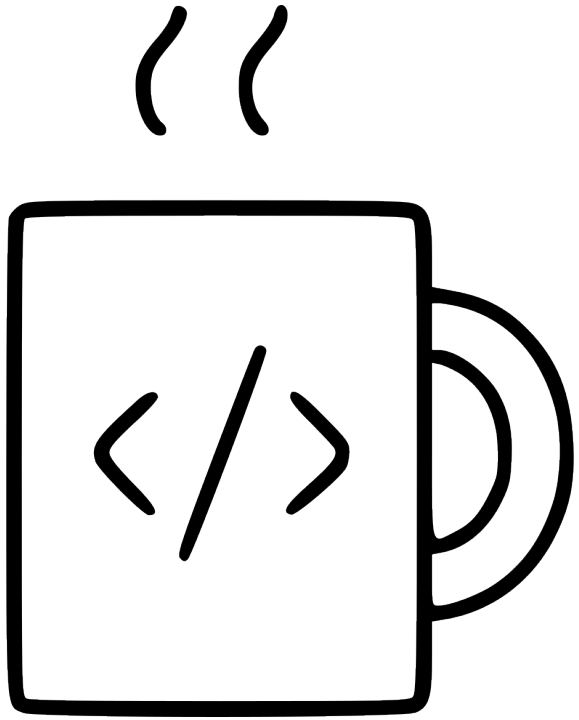
```
componentWillUnmount()
```

`componentWillUnmount()` è invocato subito prima che un componente sia smontato e distrutto. Effettua tutte le necessarie operazioni di pulizia in questo metodo, come la cancellazione di timer, richieste di rete o sottoscrizioni che avevi creato in `componentDidMount()`.

Non dovresti chiamare `setState()` in `componentWillUnmount()` perché il componente non verrà ri-renderizzato. Una volta che un'istanza di un componente è smontata, non verrà mai più montata.



https://it.reactjs.org/docs/react-component.html#static-getderivedstatefromprops



PAUSA

Ci vediamo alle ore 14.00

State e Lifecycle

```jsx
import React from 'react';
import './App.css';
import Clock from './components/Clock';

function App() {
  return (
    <div className="App">
      <Clock timezone='0' country='Rome' />
    </div>
  );
}

export default App;
```

In Rome is 13:26:47

```jsx
class Clock extends Component {
    constructor(){
        super();
        this.state = {
            date: new Date()
        };
    }

    render(){

        const tempo = this.state.date.getTime() + this.props.timezone * 3600 * 1000;
        const data = new Date(tempo);

        return <h2>In {this.props.country} is {data.toLocaleTimeString()} </h2>

    }

    tick = () => {
        //this.state.date = new Date();
        this.setState({
            date: new Date()
        })
    }

    componentDidMount() {
        this.interval = setInterval(this.tick,1000);
    }

    componentWillUnmount() {
        clearInterval(this.interval);
    }
}

export default Clock;
```

Gestione degli eventi

https://it.reactjs.org/docs/handling-events.html

Gestione degli Eventi

La gestione degli eventi negli elementi React è molto simile alla gestione degli eventi negli elementi DOM. Vi sono alcune differenze sintattiche:

- Gli eventi React vengono dichiarati utilizzando camelCase, anziché in minuscolo.
- In JSX, il gestore di eventi (*event handler*) viene passato come funzione, piuttosto che stringa.

Per esempio, l'HTML:

```
<button onclick="attivaLasers()">
  Attiva Lasers
</button>
```

è leggermente diverso in React:

```
<button onClick={attivaLasers}>
  Attiva Lasers
</button>
```

Gestione degli eventi

Un'altra differenza è che, in React, non è possibile ritornare `false` per impedire il comportamento predefinito. Devi chiamare `preventDefault` esplicitamente. Ad esempio, in un semplice codice HTML per impedire il comportamento predefinito del link di aprire una nuova pagina, potresti scrivere:

```
<a href="#" onclick="console.log('Hai cliccato sul link.'); return false">
  Clicca qui
</a>
```

In React, invece sarebbe:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('Hai cliccato sul link.');
  }

  return (
    <a href="#" onClick={handleClick}>
      Clicca qui
    </a>
  );
}
```

Gestione degli eventi

Quando definisci un componente usando una classe ES6, un pattern comune è usare un metodo della classe come gestore di eventi. Ad esempio, questo componente Interruttore renderizza un pulsante che consente all'utente di alternare gli stati "Acceso" e "Spento"

```
class Interruttore extends React.Component {
  constructor(props) {
    super(props);
    this.state = {acceso: true};

    // Necessario per accedere al corretto valore di `this` all'interno della callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(state => ({
      acceso: !state.acceso
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.acceso ? 'Acceso' : 'Spento'}
      </button>
    );
  }
}

ReactDOM.render(
  <Interruttore />,
  document.getElementById('root')
);
```

React

Passare Argomenti ai Gestori di Eventi

All'interno di un ciclo, è comune avere l'esigenza di passare un parametro aggiuntivo ad un gestore di eventi. Ad esempio, avendo `id` come l'identificativo della riga, le seguenti dichiarazioni sarebbero entrambe valide:

```
<button onClick={(e) => this.deleteRow(id, e)}>Elimina riga</button>
<button onClick={this.deleteRow.bind(this, id)}>Elimina riga</button>
```

Le due linee di codice precedenti sono equivalenti e utilizzano le funzioni a freccia e `Function.prototype.bind` rispettivamente.

Gestione degli eventi

```
class LoggingButton extends React.Component {
  // Garantisce che `this` si riferisca all'oggetto originale all'interno di handleClick.
  // Attenzione: questa è sintassi *sperimentale*.
  handleClick = () => {
    console.log('Il valore di `this` è: ', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Clicca qui
      </button>
    );
  }
}
```

Fai attenzione al valore di **this** nelle callback JSX. In JavaScript, i metodi delle classi non sono associati (bound) di default. Se dimentichi di applicare **bind** a **this.handleClick** e di passarlo a onClick, this sarà undefined quando la funzione verrà effettivamente chiamata.

due alternative
a disposizione

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('Il valore di `this` è: ', this);
  }

  render() {
    // Questa sintassi garantisce che `this` sia associato correttamente all'interno di handleClick
    return (
      <button onClick={() => this.handleClick()}>
        Clicca qui
      </button>
    );
  }
}
```

Gestione degli eventi

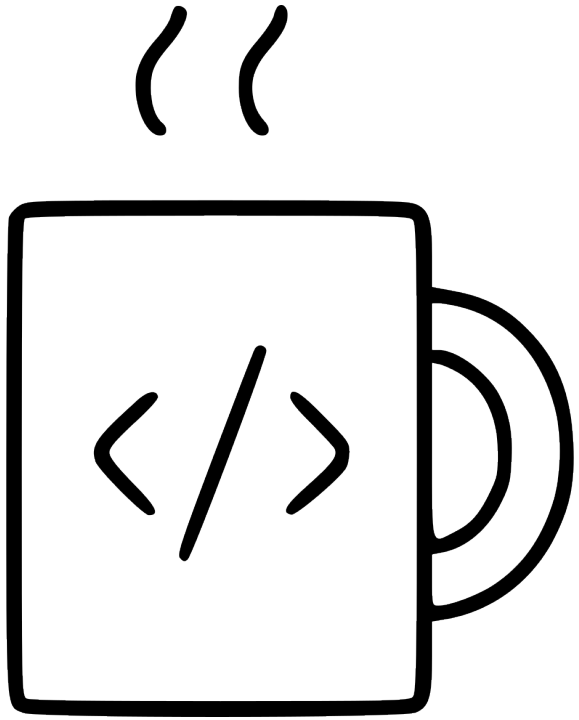
Gestione degli eventi

Se utilizziamo una semplice funzione siamo costretti a fare il **bind** nel costruttore per riferirci al **This** dell'oggetto globale.

Le **arrow function** non hanno un **This** proprio ma si riferiscono al **This** di quello che le circonda.

```
constructor(){
    super();
    this.state = {
        date: new Date(),
        stopped: false
    };
    //this.toggleWatch = this.toggleWatch.bind(this);
}
```

```
/*toggleWatch(ev) {
    //console.log(ev);
    this.setState((prevState, props) =>{
        this.state.stopped ? this.startWatch() : clearInterval(this.interval);
        return {stopped: !prevState.stopped}
    })
}*/

toggleWatch = (ev) => {
    //console.log(ev);
    this.setState((prevState, props) =>{
        this.state.stopped ? this.startWatch() : clearInterval(this.interval);
        return {stopped: !prevState.stopped}
    })
}
```



PAUSA

Ci vediamo alle ore 16.25

Form e Validazione Input lato utente

https://it.reactjs.org/docs/forms.html
https://jaredpalmer.com/formik/docs/overview

Form e Validazione Input lato utente

```
class FormNome extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('E\' stato inserito un nome: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Nome:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Con un **componente controllato**, ogni mutazione dello **stato** deve aver associata una funzione **handler**. Tutto ciò rende il processo di **modifica** e la **validazione** dell'input dell'utente semplice e lineare.

https://it.reactjs.org/docs/forms.html

Form e Validazione Input lato utente

```
state = {
  foo: {
   a: 1,
   b: 2,
   c: 3
  }
}
```

```
this.setState({ foo: {
  ...this.state.foo,
  c: 'updated value'
}})
```

Fare attenzione se si deve modificare una proprietà di un oggetto contenuto all'interno dello **state**.
In questo caso si deve utilizzare lo **spred operators** per evitare la riscrittura completa dell'oggetto.

https://it.reactjs.org/docs/forms.html

Form e Validazione Input lato utente

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Un' alternativa ai Componenti Controllati, che ci evita di scrivere un **event handler** per ogni modo in cui i tuoi dati possono cambiare, sono i **componenti non controllati**, una tecnica alternativa per implementare **forms** ed i relativi **campi di input**.

https://it.reactjs.org/docs/forms.html

Form e Validazione Input lato utente

Una soluzione che include la **validazione dei dati**, il tener traccia dei campi visitati e la sottomissione del form è **Formik**. Comunque, si basa sugli stessi principi dei componenti controllati e della gestione dello stato.

https://jaredpalmer.com/formik/

challenge
NETWORK

shaping the skills of tomorrow

challengenetwork.it