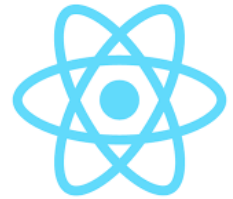
React

Roberto Ajolfi

Senior Developer@icubedsrl

roberto.ajolfi@icubed.it

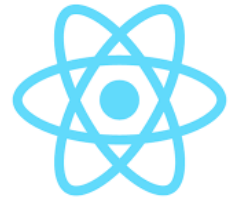
React - Agenda



- Introduzione a React
- Javascript / Typescript / JSX
- Components (Class / Function / Hooks)
- Form
- SPA con React
  - REST API
  - Routing with React Route
- Redux / Redux Toolkit / Redux Form

icubed

React – Prerequisiti SW



- React v18.2 *(latest version – June 2022)*
- Visual Studio Code
- Node JS
  - `create-react-app`

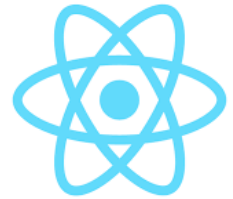
"Ci sono domande ingenue, noiose, mal formulate, domande poste senza avere esercitato un'autocritica sufficiente. Ogni domanda è però un grido per capire il mondo.

***Non esistono domande stupide**."*

(C. Sagan)

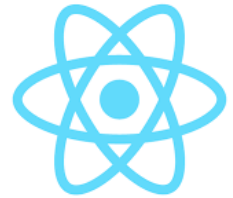
icubed

Cos’è React



- React è una libreria open source grafica per costruire interfacce utente dinamiche e complesse
- Permette la composizione di UI complesse a partire da una serie di componenti (unità di codice piccole ed isolate)
- Ideata dagli sviluppatori di Facebook e mantenuta attualmente da una nutrita community

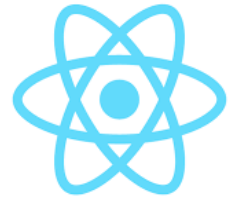
Cos’è React



È facile da imparare

- Si possono utilizzare sia plain Javascript (ECMAScript) che Typescript
- Per lo sviluppo si possono utilizzare
  - IDE complesse come Visual Studio 2019 (ha un template di progetto dedicato in Typescript, basato su ASP.NET Core)
  - editor come Visual Studio Code + create-react-app (CLI)

Cos’è React



Una cosa fondamentale da ricordare quando si lavora con React, quasi un mantra è …

“Tutto è un component.“

Tutto cio che si presenta come

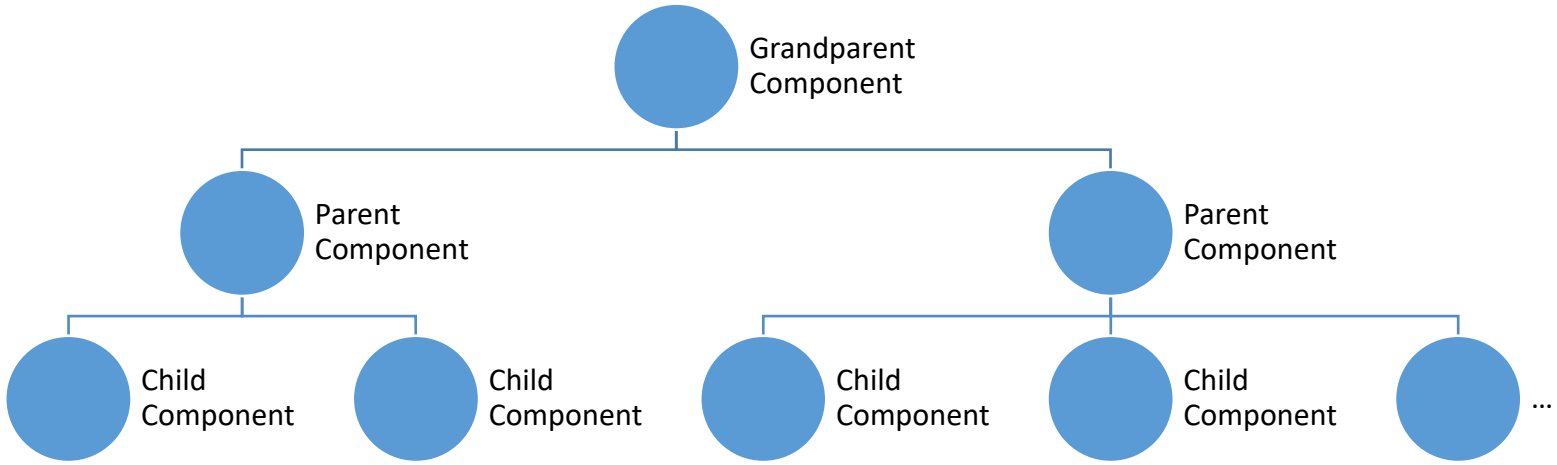
```
<TodoItem />
oppure
<TodoItem></TodoItem>
```

è un component.

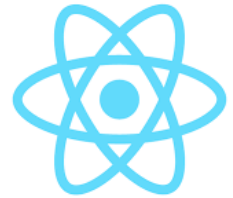
icubed

Cos’è React

Concettualmente tutte le app scritte in React presentano la stessa struttura gerarchica ad albero.



Cos’è React



- NON è un framework completo come AngularJS / Angular
- Ma è in grado di integrare facilmente altre librerie esterne che ne estendano le funzionalità
- Questa sua semplicità ed espandibilità ne permette l’utilizzo per diverse tipologie di sviluppo, dal piccolo add-in fino ad una SPA

Chi usa React

- AirBnB
- Yahoo (mail)
- BBC
- CodeAcademy
- Reddit
- …

Demo

Hello World! (x2)

icubed

Typescript

icubed

Un Javascript ‘migliore’

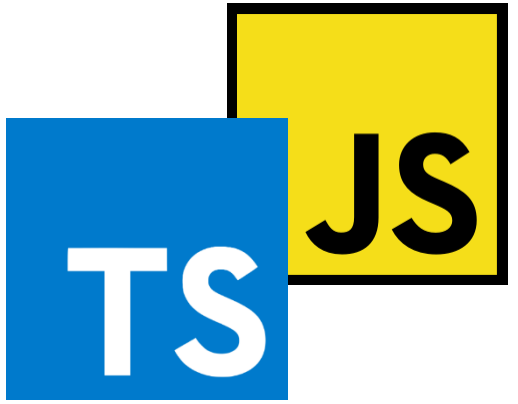
Javascript e oltre

- JavaScript è un linguaggio di scripting
  - Originariamente sviluppato da Brendan Eich della Netscape Communications e rilasciato nel 1995
  - Comunemente utilizzato nella programmazione Web lato client
  - Recentemente esteso anche al lato server

Javascript e oltre

- JavaScript è orientato agli oggetti
  - ma è un orientamento prototipico agli oggetti (**prototypical object orientation**) che è diverso dall'orientamento degli oggetti basato su classi caratteristico ad esempio di C# (**class-based object orientation**)
  - Un oggetto JavaScript è un semplice archivio di coppia chiave-valore, che è molto più simile ai dizionari C#

Javascript e oltre



- JavaScript è stato standardizzato dall'ECMA
  - Dal 1997 esiste **ECMAScript**
  - Ultima versione *ECMAScript 8 (ES2017)* del 2017
  - Nessun browser supporta pienamente le funzionalità di ECMA Script oltre la versione 5
- **TypeScript**
  - è un linguaggio di programmazione open source sviluppato da Microsoft
  - Si tratta di un superset di JavaScript che basa le sue caratteristiche su ECMAScript 6

Javascript e oltre

- **TypeScript**
  - Capo del progetto è Anders Hejlsberg (già papa di Delphi e C#)
  - Transpiling in Javascript
    - Consente di avere ES6 anche se i browser ancora non ce l'hanno
  - Quando si è trattato di riscrivere il loro framework, il team di Angular ha scelto TypeScript per la sua semplicità e potenza

  - Tipizzato

React <3 TypeScript

- La CLI `create-react-app` genera anche progetti basati su Typescript
  - È sufficiente specificare l'opzione `--template typescript`
- Occorre prestare attenzione alla tipizzazione di Typescript
  - https://github.com/typescript-cheatsheets/react-typescript-cheatsheet#reacttypescript-cheatsheets per la documentazione completa all'uso di Typescript in React

TypeScript: un esempio pratico

```tsx
import React, { Component } from 'react';

export default class Slide extends Component<ISliderProps, SliderState> {
    constructor(props: ISliderProps) {
        super(props);

        this.state = new SliderState([]);
    }

    private name: string;

    async componentDidMount() {
        const service = new DataService();

        var data = await service.GetData();

        this.setState({ data });
    }

    handleClick = (evt: React.MouseEvent<HTMLButtonElement, MouseEvent>) => {
        evt.preventDefault();
        alert("Hello!");
    }

    render() {
        return (
            <div>
                <button onClick={this.handleClick}>Click me!</button>
            </div>
        )
    }
}
```

Ha il concetto di *import* per l'acquisizione di elementi esterni al modulo corrente

Ha il concetto di classe e di interfaccia

Possiede modificatori di accesso (`public`, `private`, `protected`)

Le variabili sono tipizzate, ma il tipo `any` può essere usato in casi particolari

Può implementare un *construttore* che permette l'iniezione di input dall'esterno della classe

Gestisce *arrow functions*, *promise* e *async/await*

La parola chiave *this* ha un significato molto simile a quello di C# o Java

icubed

Typescript: Dichiarazione variabili

```
var numero; //dichiara una variabile senza specificare un tipo
var numero: number; //dichiara una variabile di tipo numerico
var numero: number = 1; //dichiara e inizializza una variabile di tipo numerico
var numero = 1; //Type Inference decide tipo

var shipped: boolean;
var shipped = true;

var name: string = "react";
var name = "react"; // type inference
```

Typescript: Dichiarazione variabili: let vs var vs const

var: variabile visibile nel metodo

```
for (var i=0; i<10;
i++){
  ...
}

console.log(i); // 9
```

let: variabile visibile nello scope

```
for (let i=0; i<10;
i++){
  ...
}

console.log(i);
//undefined
```

const: variabile visibile nello scope non modificabile

```
for (const i=0; i<5;
i++){
  ...
}

//non compila
```

Typescript converte let e const in var

Errore solo a livello di TypeScript

Typescript: Dichiarazione variabili: const

const: variabile immutabile e visibile nello scope

- Proprietà modificabili

```
const numero = 1; //immutabile
const persona = { id: 1, nome: "topolino" }; //immutabile
persona = { id: 1, nome: "paperino" }; //non ammesso
persona.nome = "paperino"; //ammesso
```

Typescript: tipi base

Array

```
let numeri: number[];
let numeri = [1,2,3];
```

Tuple

```
let x: [string, number];
x = ["stringa", 1]; // OK
x = [10, "hello"]; //Errore

x[0].replace("a", "p"); //ok

x[1].replace("a", "p");
//Errore
```

Enum

```
enum Color {
  Red, Green, Blue
};
let c: Color = Color.Green;

enum Color {
   Red = 1,
   Green = 2,
   Blue = 3
};

let c: string = Color[3];
```

Ci sono anche i classic

boolean, *string*, *number*

icubed

Typescript: tipi base (ancora)

any

- contiene qualunque valore
- qualunque metodo invocabile (a differenza di Object)

void

- tipo di ritorno da metodi che non restituiscono risultati

Type Assertion (o cast)

```
let s: any = "stringa";
let l: number =
(<string>s).length;
let l: number = (s as
string).length;
```

Typescript: Funzioni

function: come in JavaScript

- Tipizzazione
- Parametri opzionali
- Valore parametri di default
  - Tipo da type inference
- Parametri **rest**
- Arrow function
- Overload

Typescript: Funzioni (2)

```
function sum(x: number, y: number): number { ... }
sum(5, 10) //ok
sum(5) //errore

function sum(x: number, y?: number): number { ... }
function sum(x: number, y?: number = 10): number { ... }
sum(5, 13) //ok
sum(5) //ok, y = 10
sum() //errore
```

Typescript: Funzioni (3)

```
function sum(...numbers: number[]) : number {
  var result = 0;
  for(let i=0; i<numbers.length; i++){
    result += numbers[i];
  }
  return result;
}
sum(1,2,3,4);
```

Typescript: Funzioni (4)

```
var f = (x: number, y: number): number => {
 return x + y;
}

var f = (x: number, y: number): number => x + y;

f(10, 10); // in entrambi i casi la chiamata è la stessa
```

Typescript: Funzioni (5)

```
function pickCard(x: { suit: string; card: number }[]): number;
function pickCard(x: number): { suit: string; card: number };

function pickCard(x: any): any {
  let suits = ["hearts", "spades", "clubs", "diamonds"];

  if (typeof x == "object") {
    let pickedCard = Math.floor(Math.random() * x.length);
    return pickedCard;
  }
  else if (typeof x == "number") {
    let pickedSuit = Math.floor(x / 13);
    return { suit: suits[pickedSuit], card: x % 13 };
  }
}
```

Typescript: Interfacce

- Definiscono la forma di un oggetto, funzione, tipo
  - Proprietà
  - Metodi
- Evitano continue definizioni
- Proprietà opzionali
- Proprietà in sola lettura
- Nessuna traduzione in Javascript

Typescript: Interfacce (2)

```
function addPerson(p: { nome: string, cognome: string }){
  ...
}
function UpdatePerson(p: { nome: string, cognome: string }){
  ...
}
function MarryPerson(p: { nome: string, cognome: string }){
  ...
}
```

```
interface IPerson {
  nome: string;
  cognome: string;
}

function addPerson(p: IPerson) { ... }
```

Typescript: Interfacce (3)

```
//Ok
addPerson({ nome: "Mario", cognome: "Rossi" });
var x = { nome: "Mario", cognome: "Rossi" };
addPerson(x);
```

```
//Errore
addPerson({ nome: "Mario" });
var x = { nome: "Mario" };
addPerson(x);
```

Typescript: Interfacce (4)

```
interface IPerson {
  nome: string;
  cognome: string;
  eta?: number;
}

//Ok
var p1 = { nome: "Mario", cognome: "Rossi" , eta: 25};
var p2 = { nome: "Mario", cognome: "Rossi" };
addPerson(p1);
```

Typescript: Classi

Definiscono una classe in stile ES6

- Proprietà
  - Readonly
  - Getter / Setter
- Metodi
- Accessor
- Costruttori
- Ereditarietà
- Astrazione

Typescript: Definizione di classi

```
class Person {
  nome: string;
  cognome: string;
  readonly nomeCompleto: string;

  saluto() {
    alert("Hello, World!");
  }
}

var p = new Person();
```

Typescript: Costruttore di classi

```
class Person  {
  nome: string;
  cognome: string;
  readonly nomeCompleto: string;

  constructor(nome: string, cognome: string) {
    this.nome = nome;
    this.cognome = cognome;
    this.nomeCompleto = `${this.nome} ${this.cognome}`;
  }

  saluto() { alert("ciao"); }
}

var p = new Person("Mario", "Rossi");
```

Typescript: Costruttore di classi (2)

```
class Person {
  readonly nomeCompleto: string;
  constructor(public nome: string, public cognome: string){
    this.nomeCompleto = `${this.nome} ${this.cognome}`;
  }
  saluto() {
    alert("ciao");
  }
}

var p = new Person("Mario", "Rossi");
```

Typescript: Proprietà getter/setter

```
class Person {
  private _dataNascita: any;
  get dataNascita(): any {
    return this._dataNascita;
  }

  set dataNascita(value: any) {
    this._dataNascita = value;
    console.log(value);
  }
}
```

Typescript: Proprietà con accessor modifier

```
class Person {
  static EmptyPerson = new Person("", "")

  private dataNascita: string;
  protected nome: string;
  protected readonly cognome: string;

  static aStaticFunction(value: number) : number {
       return value + 2;
  }
}
```

Typescript: Classi con ereditarietà

```
class Dirigente extends Person {
  public parcheggio: string;

  saluto(): void {
    alert(`buongiorno dr ${this.nomeCompleto}.
      Il suo posto auto è ${this.parcheggio}`);
  }
}
```

Programmazione Asincrona

Promise

- Gli oggetti Promise sono usati per computazioni in differita e asincrone
- Una Promise rappresenta un'operazione che non è ancora completata, ma lo sarà in futuro
    - Due esiti possibili: risolta o rigettata

```
var myAsyncFunc = () => new
Promise<string>(
   ( resolve, reject ) => {
      // ...
      resolve( 'successPayload' );
      // reject( 'errorPayload' );
} );
```

```
var result;

myAsyncFunc
   .then( res => {
      result = res;
    } )
   .catch( errorCallback )
   .finally( finallyCallback );
```

Programmazione Asincrona

`async` / `await`

- È una sintassi semplificata per gestire le Promise in «modo sincrono»
- Quando mettiamo la parola chiave `async` prima di una dichiarazione di funzione, questa restituirà una Promise
- Al suo interno possiamo usare la parola chiave `await` che blocca il codice fino a quando la Promise viene risolta o rifiutata

```
var myAsyncFunc: string = async
    () => {
        return 'successPayload';
    }
);
```

```
try {
 var result = await myAsyncFunc();
} catch(err) {
 // catch error …
}
```

Generics

In linguaggi come C# e Java, uno dei principali strumenti per la creazione di componenti riutilizzabili sono i **generics**, ovvero essere in grado di creare un componente che può lavorare su una varietà di tipi anziché su uno singolo.

```
function identity<T>(arg: T): T
{
    return arg;
}
```

Si basa sul catturare il tipo di argomento in modo tale che si possa anche usare per indicare ciò che viene restituito.

Per poterlo fare si utilizza una variabile di tipo (**type variable**), un tipo speciale di variabile che funziona sui tipi piuttosto che sui valori.

Generics

Un funzione generica può essere chiamata in due modi.

Il primo modo è passare tutti gli argomenti, incluso il type argument, alla funzione:

```
let output = identity<string>("myString");
```

Il secondo modo è forse il più comune. Qui usiamo la *type argument inference* cioè, si lascia che il compilatore imposti il valore di `T` automaticamente in base al tipo dell'argomento che passiamo:

```
let output = identity("myString");
```

Generics

C'è molto altro che è possibile fare con i generics

- È possibile definire ed utilizzare
  - Generic **Types**
  - Generic **Interfaces**
  - Generic **Classes**

```
interface GenericIdentityFn
{
    <T>
}
```

```
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}
```

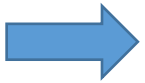
- È possibile definire tutta una serie di **vincoli** (Constraints), per limitare i tipi di dati che possono essere utilizzati

```
function loggingIdentity<T implements Lengthwise>(arg: T): T { ...
```

Optional chaining

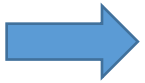
Il Concatenamento opzionale (Optional chaining) ci consente di scrivere codice in cui TypeScript può immediatamente interrompere l'esecuzione di alcune espressioni se ci imbattiamo in un valore nullo o indefinito. La chiave nel concatenamento opzionale è l'utilizzo del nuovo operatore `?.`

```
let x = (foo === null || foo === undefined) ? undefined :
foo.bar.baz();
```



```
let x =
foo?.bar.baz();
```

```
if (foo && foo.bar && foo.bar.baz)
{
    // ...
}
```



```
if (foo?.bar?.baz) {
    // ...
}
```

//@ts-ignore

Un commento `@ts-ignore` elimina tutti gli errori che hanno origine nella riga seguente.

```
if (false) {
   // @ts-ignore: Unreachable code error
   console.log('hello');
}
```

Si consiglia di utilizzare il resto del commento che segue `@ts-ignore` per spiegare quale errore è stato eliminato.

Nota: questo commento elimina **solo la segnalazione** degli errori e si consiglia di utilizzare questi commenti con parsimonia.

Demo

Primi passi con Typescript

icubed

JSX

Costruire gli Elementi del DOM

Cos’è?

```
const element = <h1>Hello, world!</h1>;
```

- Questo è un Elemento
- Può essere considerato una descrizione di ciò che si vede sullo schermo
- React lo utilizza per costruire il DOM e tenerlo aggiornato
- È scritto in JSX
  - è un’estensione della sintassi JavaScript
  - descrive l’aspetto che dovrebbe avere la UI
  - potrebbe ricordare un linguaggio di template

Com’è?

```
const myName = 'Roberto';
const element = <h1>Hello, {myName}!</h1>;
```

- Si può inserire qualsiasi espressione JavaScript all’interno delle parentesi graffe in JSX …

Com’è?

```
const fullName = (first: string, last: string): string => {
       return first + ' ' + last;
}

const element = <h1>Hello, {fullName('Roberto', 'Ajolfi')}!</h1>;
```

- … anche chiamate a funzione

Com’è?

```
const element = (
        <h1>
        Hello, {fullName('Roberto', 'Ajolfi')}!
        </h1>
);
```

- Un elemento può anche occupare più linee
- In tal caso è suggerito raggruppare il codice all’interno delle parentesi

Com’è?

```
const element = (
      <div>
            <h1> Hello, {fullName('Roberto', 'Ajolfi')}!</h1>
            <h2> Welcome in React with JSX.</h2>
      </div>
);
```

- Un elemento JSX può contenere dei figli (e dei nipoti, dei pronipoti …)

Com’è?

```
const element = <img src="./images/hello.png" />;

   const element = <img src={this.imageUrl} />;
```

- È possibile specificare attributi
  - Delimitandoli semplicemente con le virgolette
  - oppure utilizzando le parentesi graffe per includere un’espressione JavaScript. In tal caso non servono le virgolette attorno alle parentesi graffe

JSX

- JSX utilizza la convenzione *camelCase* nell'assegnare il nome agli attributi, invece che quella utilizzata normalmente nell'HTML, e modifica il nome di alcuni attributi

- Ad esempio
  - tabindex diventa tabIndex

Caso particolare: class diventa className

JSX

- Non è obbligatorio utilizzare JSX
- Ma una volta imparata la sintassi, è il modo più semplice di scrivere codice in React
- È utile come aiuto visuale quando si deve lavorare con la UI all'interno del codice JavaScript

JSX

- Ogni cosa è convertita in stringa prima di essere renderizzata
- E React effettua automaticamente l'escape di qualsiasi valore inserito in JSX prima di renderizzarlo
- Questo aiuta a prevenire gli attacchi XSS (cross-site-scripting)

JSX

- Tramite Babel, le espressioni JSX vengono ‘convertite’ in normali chiamate chiamate di funzione JavaScript che producono oggetti Javascript

```
const element = <h1 className="title">Hello, World!</h1>;
```

```
const element = React.createElement(
                      'h1',
                      {className: "title"},
                      'Hello, World!'
                  );
```

Demo

JSX Playground

icubed

I Component

I mattoni delle View

Components

- Sono l’elemento fondamentale per realizzare applicazioni in React
- Unità componibili con cui strutturare le view

Components

- Possono essere creati
  - Come una funzione Javascript (**Function Component**)

    OPPURE
  - Come classi in un file JSX / TSX (**Class Component**)

    OPPURE
  - in JavaScript puro, tramite *React.createClass()*

Components

- Se creato come classe, l'unico metodo obbligatorio è *render()*
- Vengono inseriti all'interno del DOM tramite una chiamata a *ReactDOM.render()*

Demo

Un semplice Component

icubed

Perchè utilizzare JSX?

- Il concetto alla base di tale decisione è che la logica di renderizzazione è per sua stessa natura accoppiata con le altre logiche che governano la UI: la gestione degli eventi, il cambiamento dello stato nel tempo, la preparazione dei dati per la visualizzazione, …
- Invece di separare artificialmente le *tecnologie* (codice di markup e logica in file separati), React separa le *responsabilità* utilizzando i component che contengono entrambi

Components - Props

I components scambiano i dati tra loro attraverso le "props“, attributi valorizzati dal component di livello immediatamente superiore (parent)

```
<TodoItem title='Pay the bill'>
```

La Prop si chiama *title*, il suo valore è '*Pay the bill*'.

Components - Props

Se il component `TodoItem` è un class component, tale informazione è accessibile come membro della classe

```
class TodoItem {
  render() {
    this.props.title // 'Pay the bill’
  }
}
```

Le Props sono read-only (immutable)!

Components - Props

Se il component `TodoItem` è un function component, tale informazione è accessibile come parametro della funzione

```
function TodoItem(props) {
  props.title // 'Pay the bill’
}
```

Le Props sono read-only (immutable)!

Components - Props

Per permettere ad un component di interagire con il suo parent si ricorre ad una callback prop.

```
<TodoItem title='Pay the bill' onComplete={this.handleCompletion}>
```

this.handleCompletion è una funzione definita nel parent component.

Demo

Un semplice Component – Take 2

icubed

Components - State

- I class component posso possedere **un proprio stato interno** e privato

- Non è obbligatorio e in base alla presenza o meno di esso, i components stessi vengono classificati come

    - Stateless

    - Stateful

Components - State

- Lo stato è modificabile (mutable), dati gestiti e modificabili dal component
- Lo stato di un component può essere (e spesso è) una prop per i component figli (o nipoti)

Tutto questo significa che esiste un solo punto dove tale informazione può essere modificata!

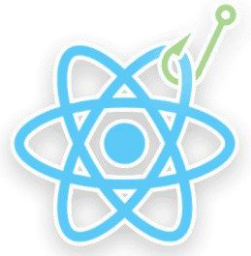
Components - State

- Un class component **stateful** necessita di uno stato iniziale, che viene impostato nel costruttore (utilizzando `this.state`)
- L'accesso in lettura allo stato avviene tramite `this.state`
- L'accesso in scrittura avviene tramite `this.setState()`, con una logica di merge

Components - State

È il cambiamento dello stato a scatenare la chiamata del metodo `render()`

E i Function Component?

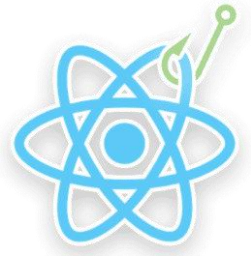


Fino alla versione 16.8.0, la gestione di uno stato era possibile solo utilizzando un class component

Poi sono stati introdotti gli Hook

Gli hook sono funzioni che consentono di "agganciarsi" allo stato di React **da un Function Component**.

Stateful Function Component



useState

Viene utilizzato per lavorare con lo stato di React.

```
import { useState } from 'react'
// ...
const [ state, setState ] = useState(initialState)
```

NOTA: se `initialState` è un oggetto complesso, la funzione di `setState` **NON utilizza una logica di merge**.

Demo

Un Component Stateful

icubed

Rendering Condizionale

- In JSX è possibile utilizzare gli operatori condizionali (if, operatore ternario, operatore &&) per definire delle logiche di rendering
- Il metodo di `render()` può in casi estremi restituire il valore **null**
  - In tal caso il component non verrà renderizzato

Rendering Condizionale

```
render() {
    const isLoggedIn = this.state.isLoggedIn;

    if (isLoggedIn) {
      return <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      return <LoginButton onClick={this.handleLoginClick} />;
    }
}
```

Rendering Condizionale

```
render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
```

Rendering Condizionale

```
render() {
  const unreadMessages = props.unreadMessages;

  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}
```

Rendering Condizionale

```
render() {
    const isLoggedIn = this.state.isLoggedIn;

  if(!isLoggedIn)
     return null;

  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

Demo

Rendering Condizionale

icubed

Liste

```
const items = [
    'Item 1',
    'Item 2',
    'Item 3'
];

<ItemsList items={items}></ItemsList>
```

```
<ul>
    <li>Item 1</li>
    <li>Item 2</li>
    <li>Item 3</li>
</ul>
```

Liste

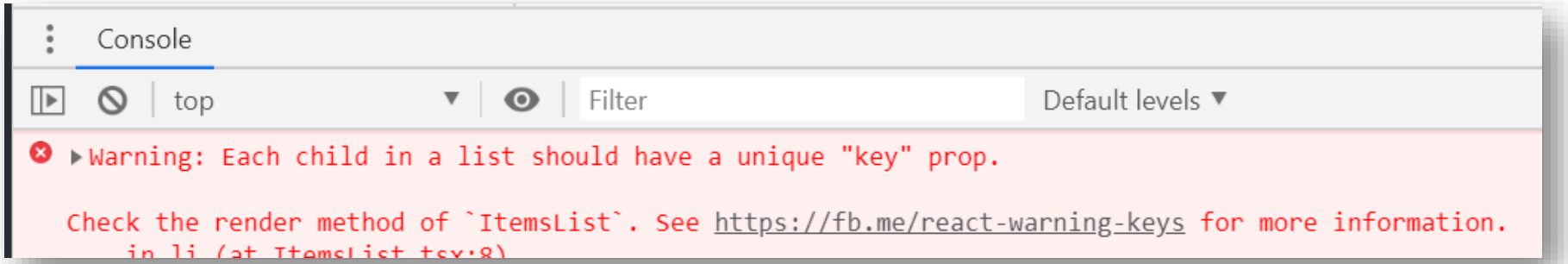
```
// … inside ItemList component
render() {
  const items = this.props.items;

  const listItems = items.map((item) =>
       <li>{item}</li>
  );

  return (
    <ul>{listItems}</ul>
  );
}
```

Parentesi JS: le Array Functions

```
//MAP
const listItems = items.map( (item) => <li>{item}</li> );

// FILTER
const selectedListItems = items.filter((item) => item.Id > 10 );

// REDUCE
const listItems = items.reduce( (total, item) => total += item.Price, 0 );

// FOREACH
items.forEach( (item) => console.log(item.Name) );
```

Keys

Quando il codice precedente viene eseguito, un warning ci avvisa che per ogni item dovrebbe essere fornita una chiave (key).



Una "key" è un attributo speciale che va incluso quando si crea una lista di elementi.

Liste e Keys

```
// … inside ItemList component
render() {
  const items = this.props.items;

  const listItems = items.map((item) =>
       <li key={item}>{item}</li>
  );

  return (
    <ul>{listItems}</ul>
  );
}
```

Keys

- Le keys servono a React per identificare quali elementi della lista sono stati aggiunti, rimossi o cambiati
- La key deve identificare univocamente un elemento all'interno di una lista ma non deve essere globalmente univocal (ci possono essere key uguali in liste diverse)

Keys

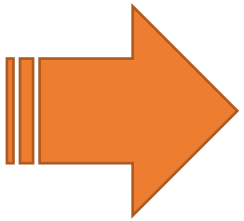
- Nel caso in cui non sia possible identificare una key univoca, è possibile ricorrere all'indice dell'element
  - Viene passato come secondo parametro della funzione passata a map

```
// … inside ItemList component
// …

  const listItems = items.map(
       (item, index) =>
            <li key={index}>{item}</li>
  );

// …
```

Props Children (pre v18)

props.children è una prop implicita dei *component che* viene utilizzata per visualizzare tutto quanto viene incluso tra il tag di apertura e quello di chiusura di un component.

```
const Picture = (props) => {
  return (
    <div>
      <img src={props.src}/>
      {props.children}
    </div>
  )
}
```



```
<Picture key={picture.id} src={picture.src}>
    //what is placed here is passed as props.children
</Picture>
```

Props Children (dalla v18)

props.children non esiste più come prop implicita dei *component*.

Se si usa Typescript, va aggiunta esplicitamente alla interfaccia utlizzata per definire le props.

```
interface IMyComponentProps {
  children: ReactNode | ReactElement | JSX.Element
  // o array degli stessi …
}

const MyComponent: React.FC<IMyComponentProps> = (props) => {
  return(<div>{props.children}</div>
};
```

Demo

Liste e prop.children

icubed

Fragments

Un pattern comune in React è quello di un component che restituisca più elementi.

I Fragments permettono di restituire un insieme di elementi figli senza aggiungere nodi supplementari al DOM.

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

Fragments

I Fragments dichiarati utilizzando il tag esplicito `<React.Fragment>` possono avere delle **Key** associate.

Questo è utile se si deve mappare una collezione in un array di frammenti — ad esempio, per creare una description list.

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Without the `key`, React will fire a key warning
        <React.Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </React.Fragment>
      ))}
    </dl>
  );
}
```

Demo

Fragments

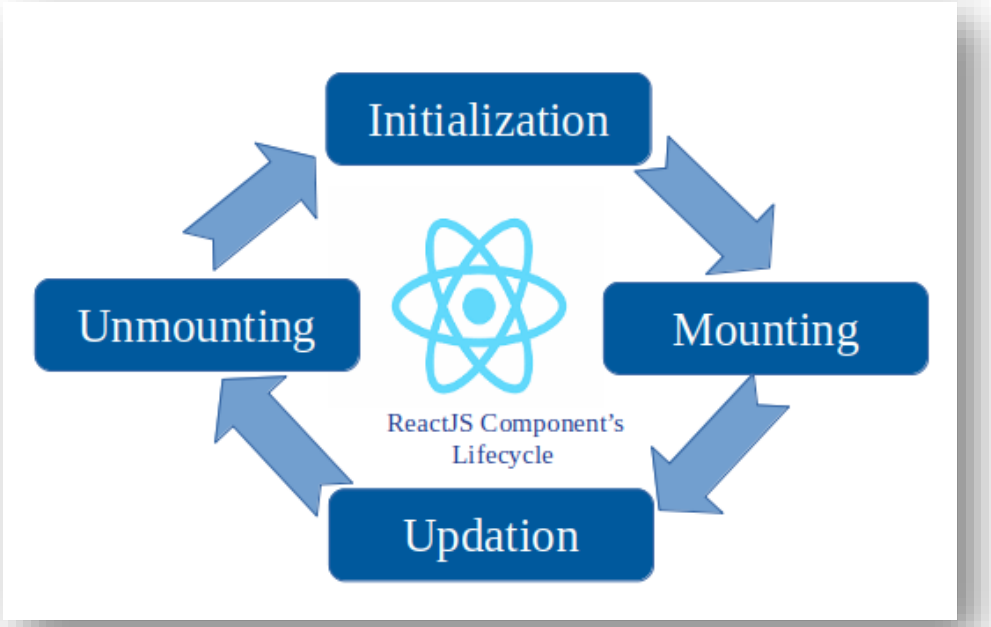
icubed

Components - Lifecycle

- Ogni componente di React attraversa un ciclo di vita

- Lungo questo ciclo di vita, React genera una serie di eventi

- Questi eventi possono essere ‘intercettati’ **in un class component**

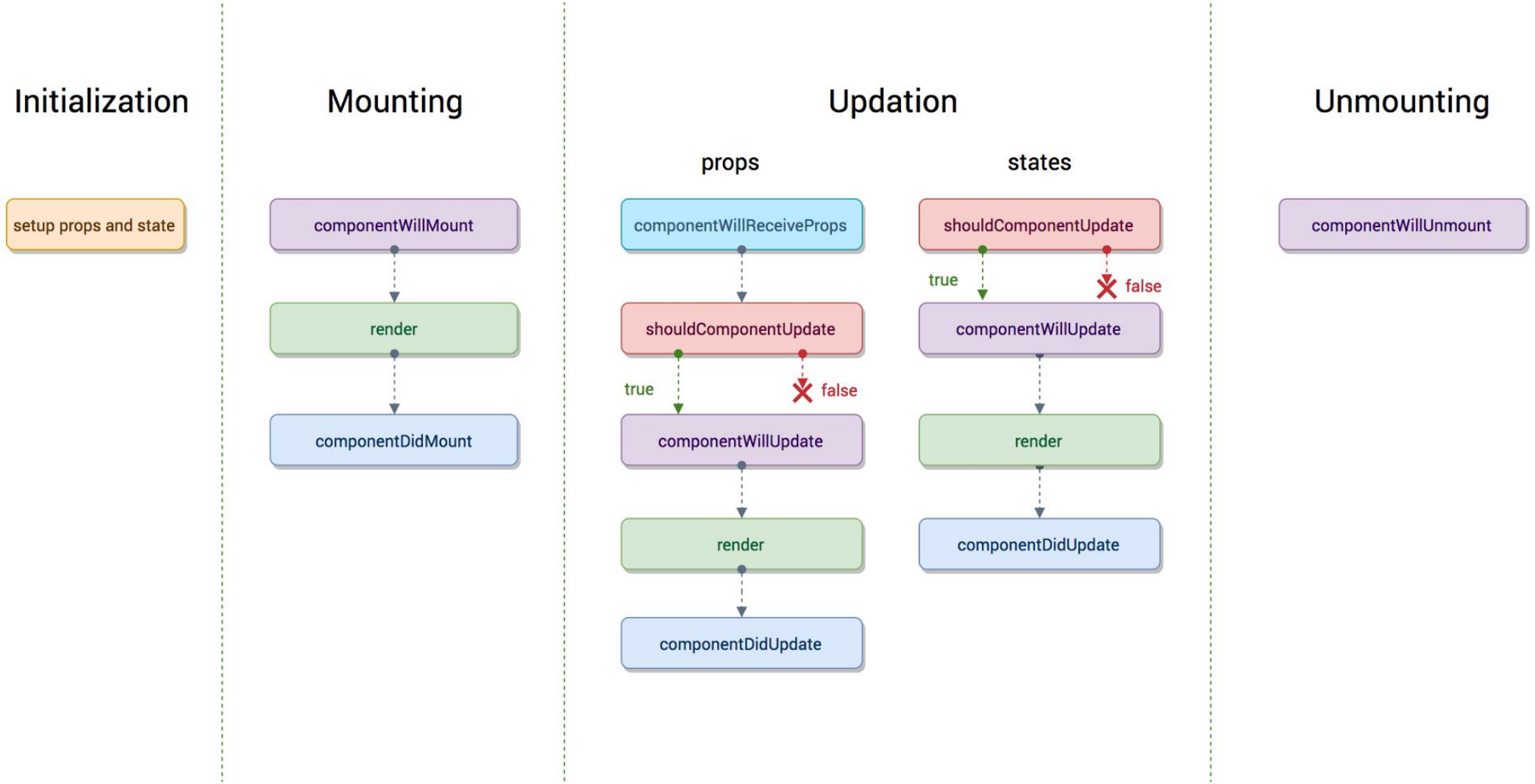


Components – Lifecycle (fino a v16.8)

- `componentWillMount()` – richiamata immediatamente prima che l'istanza di un componente venga agganciata o ridisegnata
- `componentDidMount()` – richiamata immediatamente dopo che l'istanza di un componente è stata agganciata o ridisegnata
- `componentWillUnmount()` – richiamata immediatamente prima che una istanza di un component venga sganciata o rimossa
- `componentWillReceiveProps()` – richiamata quando le Prop dell'istanza di un componente vengono aggiornate

Components – Lifecycle (fino a v16.8)

- `shouldComponentUpdate()` – utilizzato per decidere se l'istanza del componente deve essere ridisegnata
- `componentWillUpdate()` – richiamata immediatamente prima che l'istanza di un component venga ridisegnata
- `componentDidUpdate()` – richiamata immediatamente dopo che l'istanza di un component venga ridisegnata


Initialization
setup props and state
Mounting
componentWillMount
render
componentDidMount
Updation
props
componentWillReceiveProps
shouldComponentUpdate
true
false
componentWillUpdate
render
componentDidUpdate
states
shouldComponentUpdate
true
false
componentWillUpdate
render
componentDidUpdate
Unmounting
componentWillUnmount

Components – Lifecycle (from v16.9)

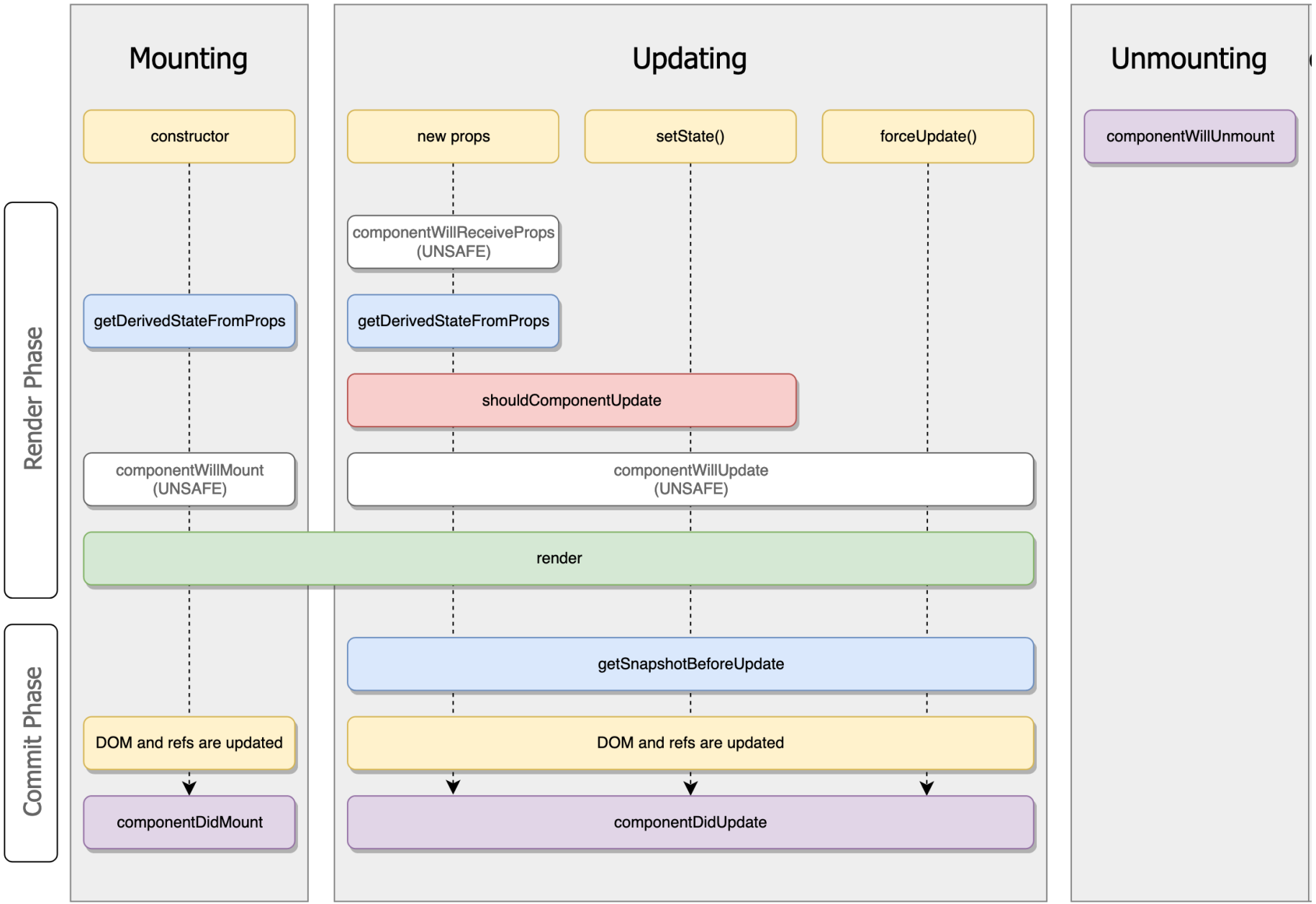
- `UNSAFE_componentWillMount()`
- `componentDidMount()`
- `componentWillUnmount()`
- `UNSAFE_componentWillReceiveProps()`
- `shouldComponentUpdate()`
- `UNSAFE_componentWillUpdate()`
- `componentDidUpdate()`

Components – Lifecycle (from v16.9)

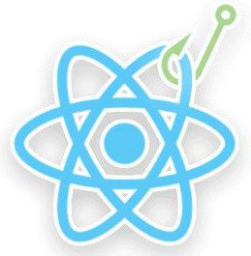
Nuovi Handler

- *[static]* `getDerivedStateFromProps()` - invocato dopo l'istanziazione di un component e prima che venga eseguito nuovamente il rendering

- `getSnapshotBeforeUpdate()` - invocato subito prima che il DOM venga aggiornato. Non viene invocato per il rendering iniziale


Mounting
Updating
Unmounting
constructor
new props
setState()
forceUpdate()
componentWillUnmount
componentWillReceiveProps
(UNSAFE)
getDerivedStateFromProps
getDerivedStateFromProps
shouldComponentUpdate
componentWillMount
(UNSAFE)
componentWillUpdate
(UNSAFE)
render
getSnapshotBeforeUpdate
DOM and refs are updated
DOM and refs are updated
componentDidMount
componentDidUpdate
Render Phase
Commit Phase

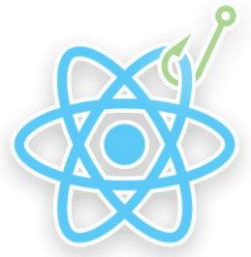
E i Function Component?



Come per la gestione dello stato, fino alla versione 16.8.0, si poteva accedere ai metodi di lifecycle solo utilizzando un class component

Poi sono stati introdotti gli Hook

Lifecycle nei Function Component

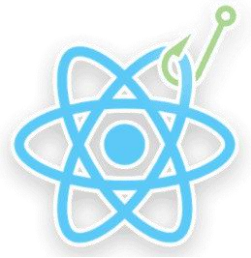


useEffect

Viene utilizzato per lavorare con il ciclo di vita del component in React.

```
import { useEffect } from 'react'
// ...
useEffect(() => {
    // effect code ...
    return () => { // cleanup code ... };
  })
```

Lifecycle nei Function Component



useEffect

```
useEffect(() => {
    // effect code ...
})
```

componentDidMount + componentDidUpdate

```
useEffect(() => {
    // effect code ...
  }, [])
```

componentDidMount ONLY

```
useEffect(() => {
    // effect code ...
    return () => { // cleanup code ... };
})
```

componentDidMount + componentDidUpdate + componentWillUnmount

icubed

<React.StrictMode>

In React (a partire dalla versione 16.3), c'è un componente chiamato `<StrictMode>` che

- Emette avvisi nella console ogni volta che rileva componenti con ciclo di vita non sicuro, vecchi `ref` e altro
- Tutti i controlli vengono eseguiti solo in fase di sviluppo e non influiscono sulla build di produzione
- Non visualizza nulla nella UI (come i Fragment)

```
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

Demo

Lifecycle methods

icubed

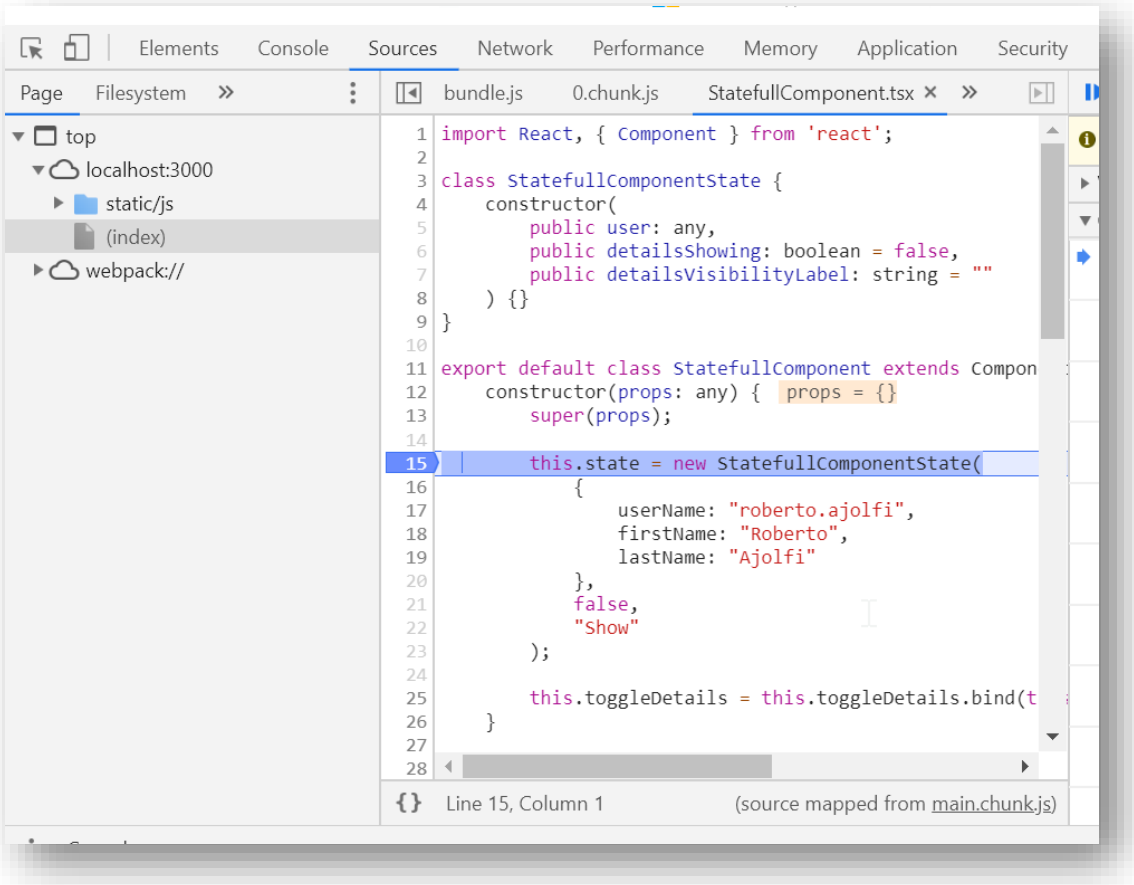
Debugging

Come faccio il debugging dei miei Component?

Debugging

Per il debugging sono disponibili 2 opzioni:

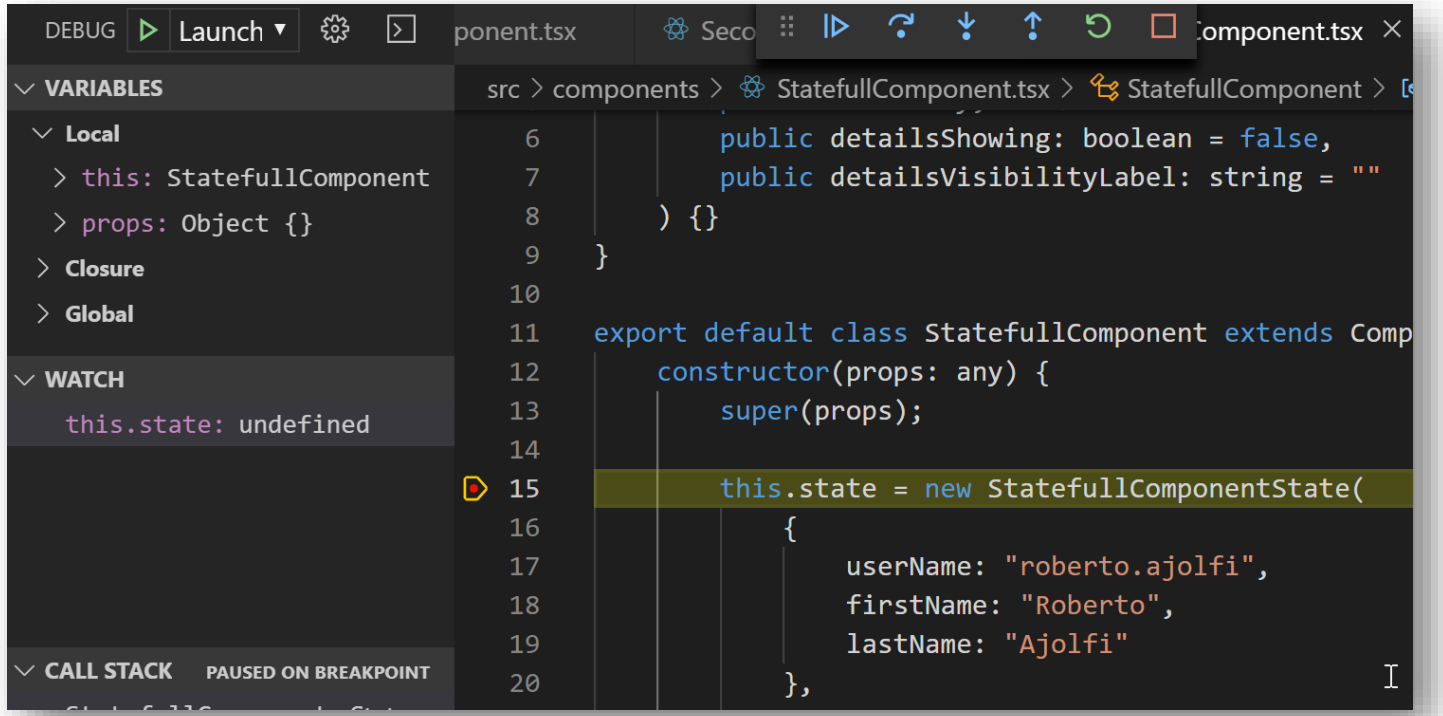
- Utilizzare i Development tools del browser (F12)
  - Chrome è il browser che offre l'esperienza migliore



icubed

Debugging

- Utilizzare il Debugger di VSCode
  - Consente di fare il debug di una applicazione da VSCode



Demo

Debug

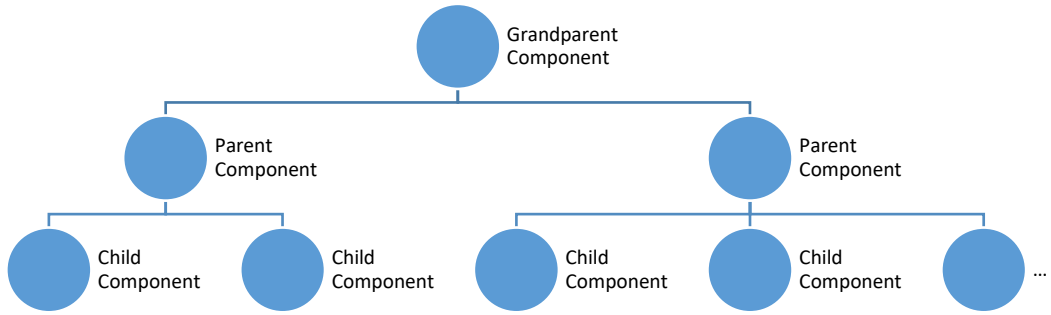
icubed

Rendering

Come vengono aggiornate le view?

Struttura di una View

Abbiamo visto che i Component in una view di React sono organizzati in una gerarchia ad albero.



Virtual DOM

Per poter ottimizzare al massimo la velocità di rendering di una view, React mantiene in memoria una sua rappresentazione del DOM che deriva dall'esecuzione dei metodi *render()* dei vari Class Component o dall’esecuzione di un Function Component.

Tale rappresentazione è chiamata **Virtual DOM**.

Virtual DOM

Nel momento in cui lo stato di un Component viene modificato, da una chiamata a

- *setState()*
- *Un setter dell'Hook useState()*
- *forceUpdate()*
- *ReactDOM.render()*

React genera una nuova versione del Virtual DOM.

Reconciliation

Viene eseguita una comparazione tra le due versioni degli alberi, in modo da identificare i cambiamenti che si sono verificati.

Solo quanto risulta diverso viene effettivamente renderizzato nel *DOM "reale"*.

Questo processo viene definito **Reconciliation**.

Altri dettagli: https://reactjs.org/docs/reconciliation.html

Renderizzazione

La renderizzazione del DOM "reale" avviene con un **metodo a due passate**:

- prima viene generato il markup
- una volta iniettato il markup nel DOM "reale", gli vengono agganciati gli eventi

Forms

Gestione delle form, validazione

Form

- React è una libreria che non ha nessuna specifica funzionalità di gestione delle form
  - es. Angular reactive forms o template-driven

- La gestione delle form passa attraverso i **Controlled Components** ed eventuali librerie aggiuntive esterne

Form

```
export default class Form extends Component<any, any> {
  render() {
    return (
        <form>
            <div className="form-group row">
                <label htmlFor="name">Name</label>
                <div className="col-sm-10">
                    <input className="form-control" type="text" name="name" />
                </div>
            </div>
            <div className="form-group row">
                <div className="col-sm-10">
                    <input className="btn btn-primary" type="submit" value="Submit" />
                </div>
            </div>
        </form>
    )
  }
}
```

Controlled Component

- Gli elementi di una form HTML, come <input>, <textarea>, and <select> mantengono un loro stato interno che aggiornano sulla base della interazione con l'utente
- In React, lo stato viene gestito a livello di Component e aggiornato solo tramite chiamate a *setState()* o utilizzando l'hook *useState()*
- È possibile combinare le due cose rendendo lo stato in React la "singola sorgente di verità"
  - Allora il Component che fa il rendering della form controllerà anche cosa accade alla form in seguito alla interazione con l'utente

Controlled Component

Un elemento di input di una form il cui valore è controllato da React è chiamato **Controlled Component**

Form

```
handleChanges = (event: any) => {
    this.setState({ name: event.target.value });
}

handleSubmit = (event:any) => {
    event.preventDefault();
    alert('You submitted: ' + JSON.stringify(this.state));
}
```

```
render() {
    return (
        <form onSubmit={this.handleSubmit}>
            <div className="form-group row">
                <label htmlFor="name">Name</label>
                <div className="col-sm-10">
                    <input className="form-control" type="text" name="name"
                        value={this.state.value} onChange={this.handleChange}/>
                </div>
            </div>
            <div className="form-group row">
                <div className="col-sm-10">
                    <input className="btn btn-primary" type="submit" value="Submit" />
                </div>
            </div>
        </form>
    )
```

Demo

Una semplice form

icubed

Form

Ovviamente ci sono più campi in una form e allora bisogna modificare leggermente il metodo *handleChange()*

```
handleChange(event: any) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }
```

Demo

Una semplice form

icubed

Uncontrolled Component

- Per un elemento di una form HTML, <input type='file'>, non è purtroppo possibile utilizzare lo stesso approccio
- Il motivo risiede nel fatto che il valore di questo tipo di elemento non può essere impostato programmaticamente
- Occorre usare la File API per interagire con esso

Si rende necessario creare una ref per accedere direttamente all'elemento del DOM nell'handler di submit.

Form

```
constructor(props: any) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);

    this.fileInputTag = React.createRef();
}
```

```
handleSubmit(event: any) {
    event.preventDefault();
    alert(`è stato selezionato il file: ${this.fileInputTag.current.files[0].name}`);
}
```

```
render() {
  return (
      <form onSubmit={this.handleSubmit}>
      <label>
          Scegli un file:
          <input type="file" ref={this.fileInputTag} />
      </label>
      <br />
      <button type="submit">Upload</button>
      </form>
  )
}
```

icubed

Demo

Una form con <input type='file'>

icubed

Validazione

- Anche per la validazione, React non offre nulla direttamente ma ci sono diverse alternative
  - Si puo ad esempio usare la **validazione nativa di HTML5**

https://www.npmjs.com/package/simple-react-validator

- Questo è un esempio di una libreria aggiuntiva disponibile
  - Se ne trovano altre
  - Se ne può scrivere una ☺

Demo

Una semplice form con validazione

icubed

Form

```
constructor(props: any) {
    super(props);

    this.state = { name: '', role: '' };

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

    this.validator = new SimpleReactValidator({
        className: 'errorMessage'
    });
}
```

```
handleSubmit = (evt: FormEvent) => {
    evt.preventDefault();

    if(this.validator.allValid()) {
        alert(JSON.stringify(this.state));
    } else {
        this.validator.showMessages();
    }
};
```

```
<div className="form-group row">
    <label htmlFor="name">Name</label>
    <input className="form-control" type="text" name="name"
        value={this.state.name} onChange={this.handleChange}/>
    {this.validator.message('name', this.state.name, 'required')}
</div>
```

icubed

Demo

Una semplice form con validazione

icubed

Dal Component alla SPA

Utilizzare React per realizzare una SPA

SPA

- Un'applicazione web o un sito web che può essere usato o consultato su una singola pagina web
- Tutto il codice necessario è recuperato in un singolo caricamento della pagina
- Le risorse appropriate sono caricate dinamicamente e aggiunte alla pagina quando necessario

SPA

- React può essere utilizzato per realizzare una Single Page Application
- Per farlo però occorre aggiungere una serie di moduli che gli permettano di gestire alcune funzionalità fondamentali
  - Routing (React Routing)
  - Interazione con le API (Fetch)
  - State management avanzato (Context / Redux)
  - …

Il Routing

Navigare le viste della SPA

Routing

- Una delle caratteristiche fondamentali di una SPA è la capacità di navigare tra diverse “pagine”
  - Ovviamente non sono vere pagine (altrimenti non sarebbe una SPA!), ma sembrano tali dal punto di vista dell’utente
- Poichè non si tratta di caricare nuove pagine dal server, occorre definire una modalità di navigazione che non sia quella standard del browser. Questa modalità è il *routing* e ogni SPA framework ne fornisce una sua implementazione (Router)

Routing

- La descrizione delle “pagine” accessibili tramite il Router viene definita da un insieme di Route
  - si possono definire Route statiche (*/about*) o dinamiche (*/album/:id*, dove *:id* rappresenta una variabile)
  - Nested Routing (routing su più livelli)
- Il Router confronta l’URL della richiesta con tutte le Route alla ricerca di un match
  - Se c’è un match, il Router avvia il re-rendering della applicazione

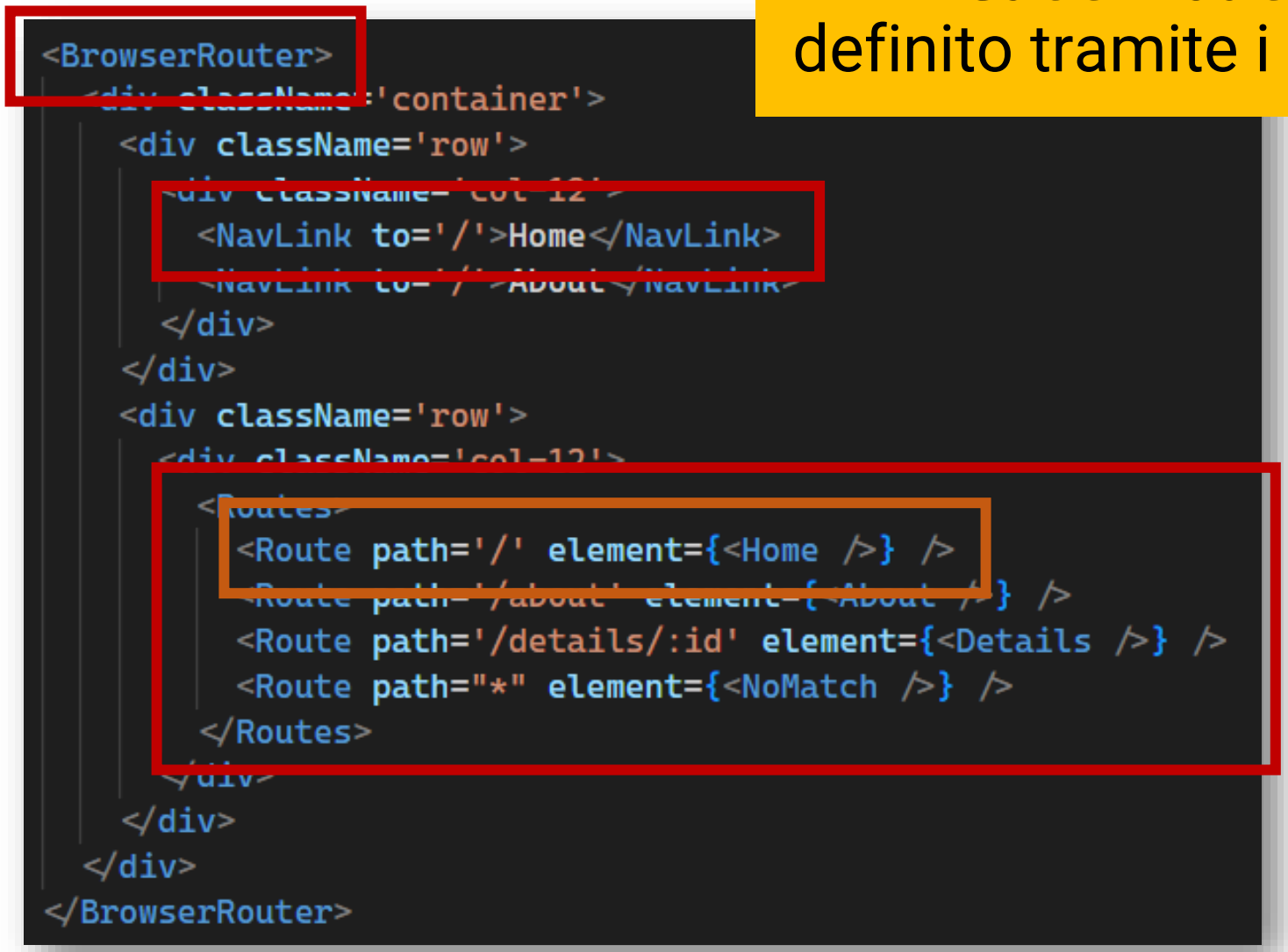
Routing

- React non include un proprio componente di routing
- Esiste però una libreria esterna, **react-router**, che è uno standard de-facto
  - I suoi ideatori parlano di dynamic routing (in confronto allo static routing di altri framework come Angular)
  - Non esiste una configurazione delle route come parte della inizializzazione della applicazione

A partire dalla versione 6, `react-router` supporta direttamente i soli function component

Routing

In `react-router` tutto è definito tramite i Component



Nested Routes

- React-router non ha una API specifica per gestire le nested route, perchè non gli serve!
- Se tutto è un Component, allora basta aggiungere un blocco `<Routes>` / `<Route>` dove serve!

```
<Route path='/subroute/*' element={<SubRoute />} />
```

```
<Routes>
  <Route path="/" element={<AdminHome />}>
    <Route index element={<AdminUsers />}/>
    <Route path='computers' element={<AdminComputers />}/>
  </Route>
</Routes>
```

```
<div>
  <NavLink to='/subroute'>Users</NavLink> | 
  <NavLink to='/subroute/computers'>Computers</NavLink>
</div>
<Outlet />
```

icubed

Demo

Simple Routing

Nested Routing

Una vera SPA – Take 1

icubed

Utilizzare le API

Ovvero l’accesso ai dati

Accesso ai dati da React

- L’approccio più diffuso per accedere ai dati da una SPA è utilizzando chiamate a API REST
- Anche in questo caso React non ha funzionalità ‘native’
  - È quindi possible integrare una qualsiasi libreria AJAX come Axios, jQuery AJAX, o la funzione window.fetch (built-in nel browser)
- Il punto corretto in cui effettuare tali chiamate AJAX è nel metodo *componentDidMount*
  - In questo modo è possibile utilizzare il metodo setState con i dati ricevuti

Accesso ai dati (REST Web Service)

Per accedere ai dati tramite un servizio REST bastano

- Un URL

```
https://servername/api/resourcename
```

- L'utilizzo degli HTTP Verbs

HTTP Verbs	CRUD Operations
GET	READ (one or many records)
POST	CREATE
PUT	UPDATE
DELETE	DELETE

Accesso ai dati (REST Web Service)

Esempi di chiamate

HTTP Verbs	URL	Body	Returns (*)
GET (list of resources)	https://*servername*/api/*resourcename*	-	JSON array of resources
GET (single resource by ID)	https://*servername*/api/*resourcename/ID*	-	Single resource as JSON
POST	https://*servername*/api/*resourcename*	Resource as JSON	-
PUT	https://*servername*/api/*resourcename/ID*	Resource as JSON	-
DELETE	https://*servername*/api/*resourcename/ID*	-	-

(*) errors in case of issues

icubed

Accesso ai dati (REST Web Service)

- L’autenticazione viene gestita in genere tramite l’aggiunta di un Authentication Header alla chiamata HTTP
  - BASIC Authentication
  - JWT (Token based)
  - …
- HTTPS ! Always !!!

Demo

Chiamate AJAX

Demo

Una vera SPA – Take 2

icubed

Deployment

Il deployment dell’applicazione

Deployment

- Il processo di deployment della SPA inizia eseguendo il commando

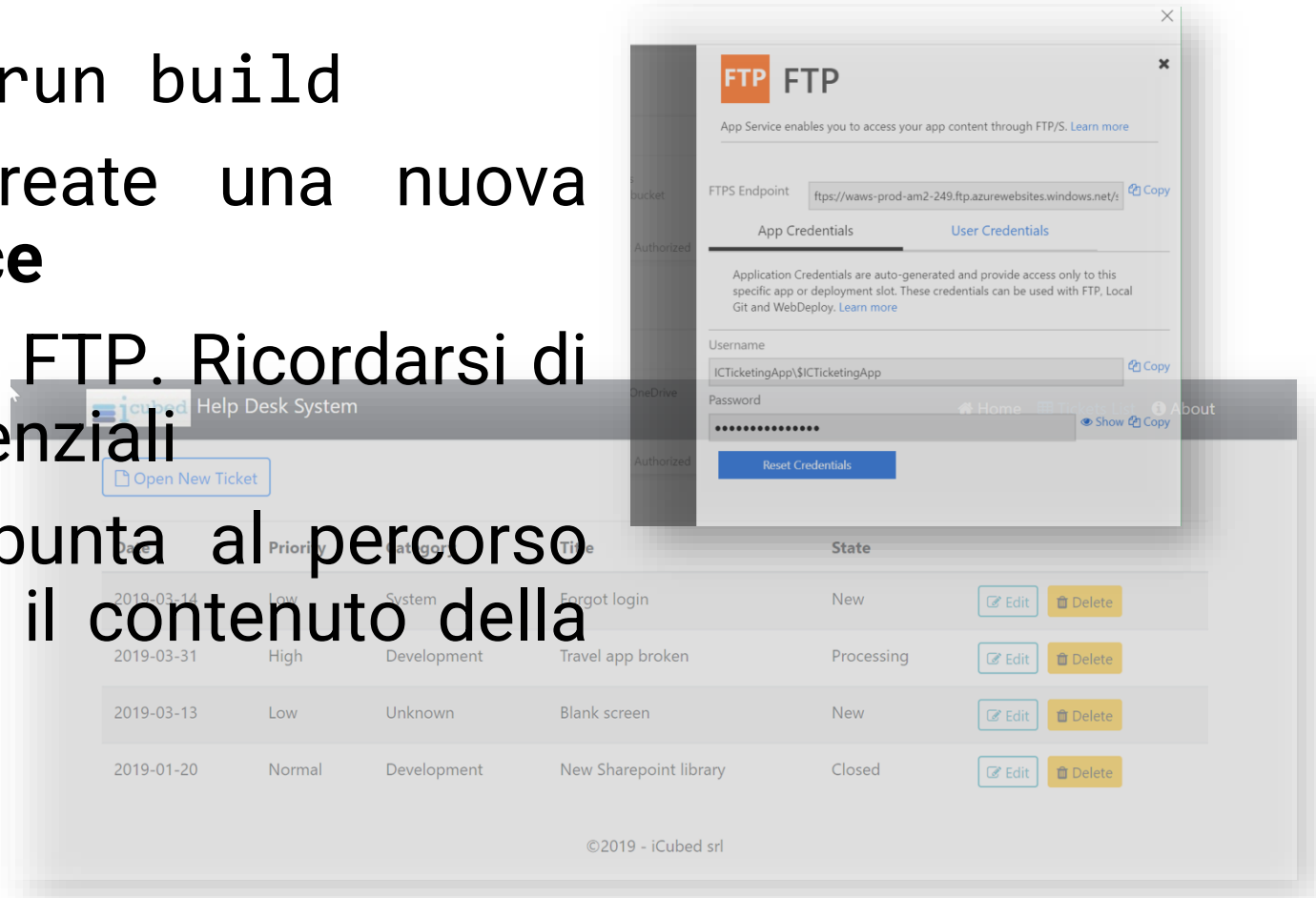
```
npm run build
```

- Il commando crea una nuova cartella **build** che contiene il codice pronto per essere rilasciato in produzione
- È sufficiente una qualsiasi soluzione di hosting di siti statici per poter eseguire la applicazione
- Alcune delle opzioni disponibili:

https://facebook.github.io/create-react-app/docs/deployment

Deployment in Azure

- Si parte eseguendo `npm run build`
- Nell'*Azure Portal* va create una nuova risorsa di tipo **App Service**
- Va configurato l'accesso FTP. Ricordarsi di prendere nota delle credenziali
- Con un client FTP, si punta al percorso indicato e carica li tutto il contenuto della cartella **build**



icubed

Demo

Una vera SPA – Take 3

icubed

Gestire lo Stato con Redux

Gestione avanzata dello Stato con la libreria Redux

Un problema da risolvere

All’aumentare di dimensioni e complessità di una SPA, anche il codice che ne deve gestire lo stato cresce.

Lo stato arriva ad includere risposte dal server, dati in cache e dati locali. Anche lo stato della UI vede la propria complessità aumentare (routes dinamiche, tabs, spinners, paginazione, …).

Ad un certo punto diventa quasi impossibile capire cosa accade nella app, il Come-Quando-Perchè del suo stato.

Introducing Redux

Redux è un container per lo stato delle app Javascript, che permette di rendere le mutazioni dello stato predicibile, tramite l'imposizione di alcune restrizioni su come e quando tali mutazioni possono avvenire.

Si basa su **tre principi.**

Introducing Redux

1. **Single source of truth** – l'intero stato dell'applicazione è mantenuto (come oggetto javascript) all'interno di un unico store
2. **State is read-only** – lo stato NON può essere modificato direttamente, ma solo tramite alcune azioni, oggetti che descrivono cosa accade allo stato stesso
3. **Changes are made with pure functions** – come lo stato viene trasformato dalle azioni è specificato da una funzione pura (reducer)

Store

Lo Store è l'oggetto responsabile di:

- Conservare lo stato dell'applicazione
- Consentire l'accesso allo stato corrente (chiamata al metodo *getState()*)
- Permettere la modifica dello stato tramite il dispatch delle azioni
- Mantenere l'elenco degli elementi interessati ai cambi di stato (listener), tramite il metodo *subscribe()*

È importante ricordare che esiste sempre uno e un solo Store in una applicazione che usa Redux.

Actions

Le Action sono l'unico metodo disponibile all'applicazione per alterare il proprio stato, tramite l'invio di dati allo Store.

Sono costituite da un descrittore dell'azione e dal suo payload (dati).

Vengono inviate allo Store tramite l'invocazione del metodo *store.dispatch()*

Reducers

I Reducer specificano come lo stato dell'applicazione cambia in risposta ad una Action inviata allo Store.

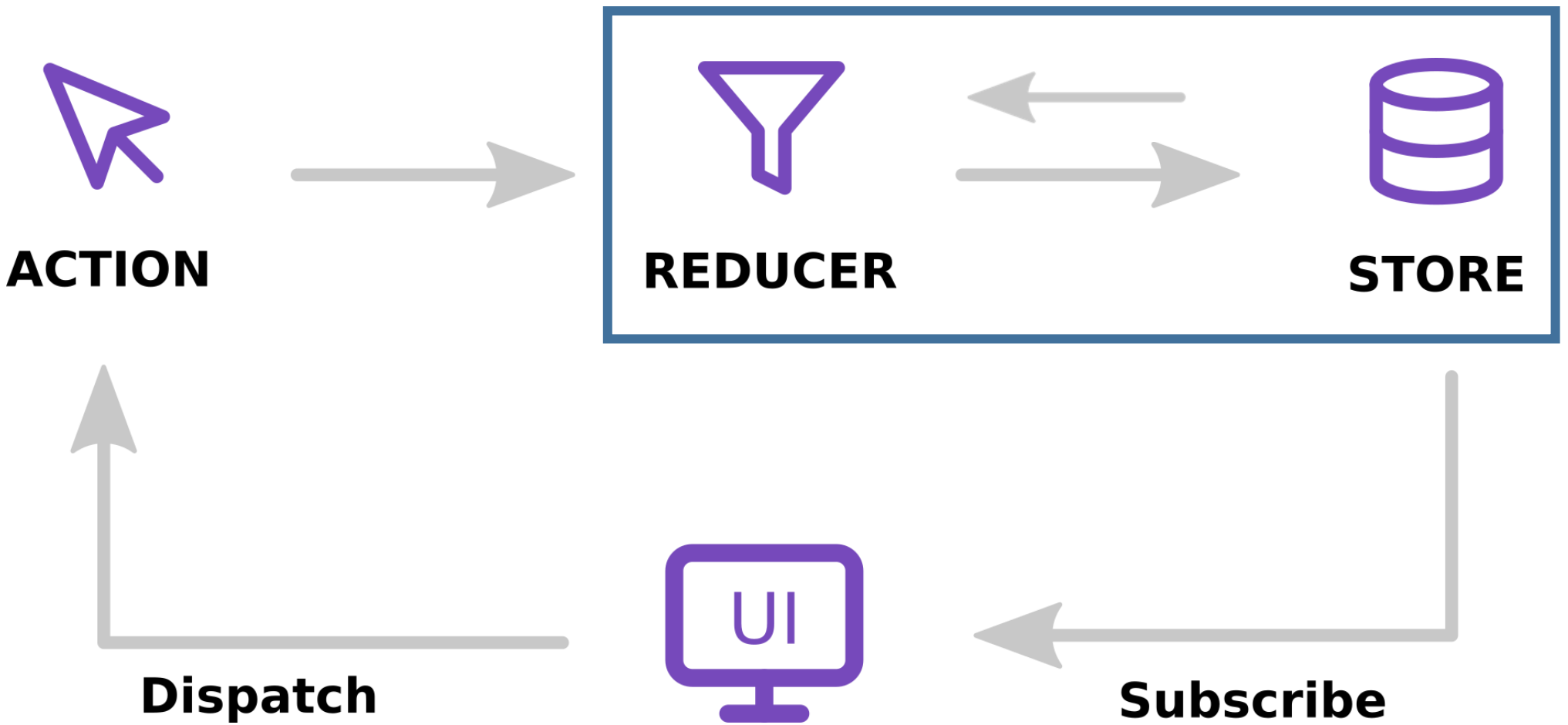
Il reducer è una funzione pura, che prende lo stato corrente e una Action come input. Restituisce il nuovo stato.

(previousState, action) => newState

Cose che NON possono avvenire all'interno del Reducer:

- Modificare i propri argomenti
- Eseguire codice con evidenti effetti collaterali i.e. chiamate ad API o transizioni di routing
- Chiamare funzioni non pure i.e. *Date.now()* o *Math.random()*

Come ‘gira’ Redux



Aggiornamento dei dati in Redux

Nelle applicazioni che usano Redux l'aggiornamento dei dati segue sempre gli stessi 4 passi:

- Viene chiamata *store.dispatch(action)*
- Lo Store invoca il Reducer
  - Il reducer principale può dover combinare il risultato dell'invocazione di più reducers in un singolo stato
- Lo Store salva il nuovo stato ritornato dal Reducer. Questo è il nuovo stato della applicazione! Ogni elemento registrato riceve una notifica dell'avvenuto cambio di stato (e può agire di conseguenza i.e. un cambio della UI)

Integrare Redux nella UI

Integrare Redux in un qualsiasi layer di UI richiede sempre lo stesso insieme di passaggi:

- Creare un Redux Store
- I componenti della UI che ne hanno bisogno si registrano allo Store per ottenere gli update (subscription)

Integrare Redux nella UI

Integrare Redux in un qualsiasi layer di UI richiede sempre lo stesso insieme di passaggi:

- All'interno della callback di subscription:
  - Viene recuperato lo stato attuale
  - Da questo, vengono estratti i dati necessari all'aggiornamento della specifica parte della UI (responsabilità del componente)
  - Viene aggiornata la UI (con logica interna al componente)
- Se necessario, renderizzare la UI con lo stato iniziale
- Reagire agli input dell'utente, tramite il dispatch delle Azioni

React Redux

- React Redux è la libreria ufficiale di binding di Redux in React
- Permette ai component React di leggere dati da uno Store e di effettuare il dispatch delle Azioni per aggiornare lo stesso Store
- Gestisce la logica di interazione con lo Store, in modo che questa parte non debba essere scritta dallo sviluppatore dell'applicazione

React Redux

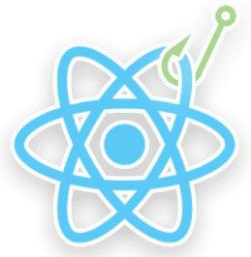
- React Redux fornisce il componente `<Provider />`, che consente la comunicazione con lo Store dell'intera applicazione
- React Redux fornisce una funzione *connect()* che consente di connettere un Class Component allo Store dati e handler di un componente
- La libreria implementa una serie di ottimizzazioni per garantire le migliori performance all'applicazione

Demo

Utilizzare Redux (Class Component)

icubed

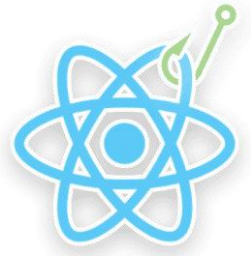
E i Function Component?



Come per la gestione dello stato e del ciclo di vita, fino alla versione 16.8.0, si poteva accedere a Redux solo utilizzando un class component

Poi sono stati introdotti gli Hook

Redux & Function Component

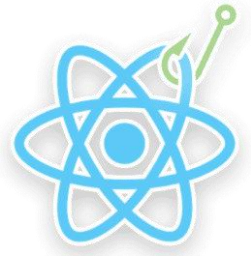


useSelector

Viene utilizzato per estrarre informazioni dallo stato di Redux.

```
import { useSelector } from 'react-redux'
// ...
const darkTheme = useSelector<StateType>(state => state.valueFromState);
```

Redux & Function Component



useDispatch

Viene utilizzato per fare il dispatch di un messaggio verso lo store di React.

```
import { useDispatch } from 'react'
// ...
const dispatch = useDispatch();
// ...
dispatch(action_instance);
```

Demo

Utilizzare Redux (Function Component)

icubed

Redux Thunk

- Redux-Thunk è un middleware che consente di effettuare il dispatch di funzioni, anziché solo di azioni, verso lo store di Redux
- Uno dei principali casi d'uso di questo middleware riguarda la gestione di azioni asincrone
  - Es. azioni che utilizzano `axios` per inviare una richiesta ad un servizio web
- Redux-Thunk ci consente di inviare tali azioni in modo asincrono e risolvere la `Promise` che viene restituita

Redux Thunk

- Va installato il pacchetto di redux-thunk

  `npm install redux-thunk`

- Poiché è un middleware di Redux, deve essere abilitato utilizzando la funzione `applyMiddleware` in fase di creazione dello store

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';  427 (gzipped: 269)
import rootReducer from './reducers';

const store = createStore(
  rootReducer,
  applyMiddleware(thunk)
);

export default store;
```

Redux Thunk

- Viene scritta una funzione che restituisce un oggetto di tipo `ThunkAction` (thunk action builder)
- All'interno di tale funzione
  - Si può operare in modo asincrono
  - Si fa il dispatch di altre azioni

```
export const addUsers = (newUser: User): ThunkAction<
  void,
  UserState,
  null,
  AddUserBegin | AddUserFailure | AddUserSuccess
> => {
  return async (dispatch, getState) => {
    dispatch(addUsersBegin());

    const svc = new UserDataService(Settings.UserAPIUrl);

    try {

        const result = await svc.addUser(newUser);

        if(result) {
            dispatch(addUsersSuccess(newUser))
        } else {
            dispatch(addUsersFailure("Error saving the User"));
        }
    } catch(err: any) {
        dispatch(addUsersFailure(err));
    }
  };
};
```

Redux Thunk

- Ciascuna delle azioni di cui si fa il dispatch va gestita da un reducer

```
case ActionTypes.ADDUSER_BEGIN:
  return {
    users: [...currentState.users],
    loading: true,
    error: undefined,
  };
case ActionTypes.ADDUSER_SUCCESS:
  return {
    ...currentState,
    users: [...currentState.users, action.payload],
    loading: false,
    error: undefined,
  };
case ActionTypes.ADDUSER_FAILURE:
  return { users: [], loading: false, error: action.payload };
```

- Il dispatch della ThunkAction avviene nello stesso modo delle azioni ‘normali’

```
dispatch(addUsers({
  id: uuid(),
  firstname: firstName,
  lastname: lastName
}));
```

icubed

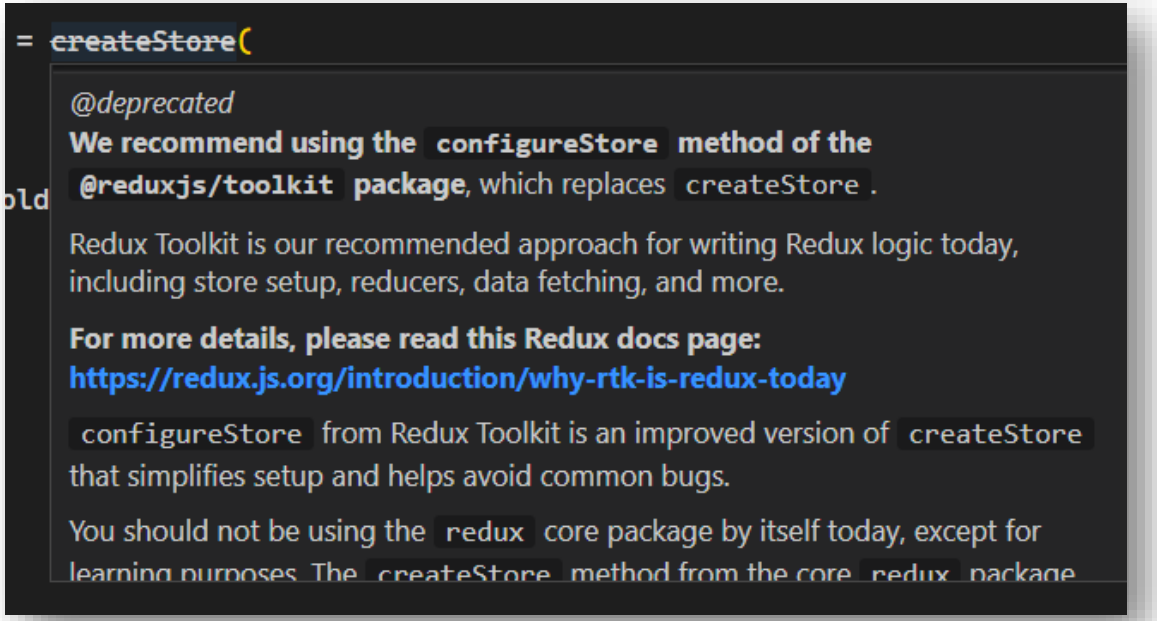
Redux Toolkit

icubed

Usare Redux in modo più semplice

Redux Toolkit

A partire dalla v18 di React, i metodi di Redux come `createStore()` vengono segnati come deprecati e viene raccomandato il passaggio a Redux Toolkit.



icubed

Redux Toolkit

Redux Toolkit è pensato come *«il modo standard per scrivere la logica Redux»*.

È stato creato per aiutare a risolvere tre problemi comuni evidenti nell'utilizzo di Redux:

- *"Configurare uno store di Redux è troppo complicato"*
- *"Devo aggiungere molti pacchetti per fare in modo che Redux faccia qualcosa di utile"*
- *"Redux richiede troppo boilerplate code"*

Redux Toolkit - Setup

Aggiungere ad una applicazione React esistente Redux Toolkit è semplice.

Insieme ai pacchetti base (`redux` e `react-redux`), va installato solo un altro package

```
> npm install @reduxjs/toolkit
```

Redux Toolkit - Setup

L’utilizzo del toolkit prevede sempre di wrappare il componente root dell’applicazione con il component `<Provider>` fornito da `react-redux`.

```
root.render(
  <React.StrictMode>
    <Provider store={store}>
      <App />
    </Provider>
  </React.StrictMode>
);
```

Redux Toolkit - Setup

L’utilizzo del toolkit prevede sempre di wrappare il componente root dell’applicazione con il component `<Provider>` fornito da `react-redux`.

L’interazione con lo store avviene ancora tramite `connect()` per i class component …

```
const MapStateToProps =
  (store: AppState) => ({
    todoList: store.todo.todos,
    darkMode: store.theme.darkMode,
    loading: store.todo.loading,
  });

const MapDispatchToProps =
  (dispatch: Dispatch<AppStateAction>) => ({
    addTodo: (item: Todo) => dispatch(addTodo(item)),
  });

export default connect(
  MapStateToProps,
  MapDispatchToProps
)(TodoListClass);
```

Redux Toolkit - Setup

L’utilizzo del toolkit prevede sempre di wrappare il componente root dell’applicazione con il component `<Provider>` fornito da `react-redux`.

… e gli hook `useDispatch()` e `useSelector()` per i function component.

```
const dispatch = useDispatch();

const users = useSelector<AppState, User[]>(
  (s) => s.user.users);
const loading = useSelector<AppState, boolean>(
  (s) => s.user.loading);
```

Redux Toolkit - Setup

L'utilizzo del toolkit prevede sempre di wrappare il componente root dell'applicazione con il component `<Provider>` fornito da `react-redux`.

Quello che cambia è il modo in cui

- si definiscono azioni e reducer
- viene inizializzato lo store

```
export const themeSlice = createSlice({
  name: 'theme',
  initialState,
  reducers: {
    toggleDarkTheme: (state) => {
      state.darkMode = !state.darkMode;
    }
```

```
const store = configureStore({
  reducer: {
    theme: themeReducer,
    users: usersReducer
  },
});
```

Redux Toolkit - Slice

In Redux Toolkit vengono create delle **slice** (utilizzando la funzione `createSlice`), allo scopo di descrivere come viene gestita una parte dello stato.

```
export const themeSlice = createSlice({
  name: 'theme',
  initialState,
  reducers: {
    toggleDarkTheme: (state) => {
      state.darkMode = !state.darkMode;
    }
  }
});

export const { toggleDarkTh                    ons;

export default themeSlice.reducer;
```

… lo stato può essere modificato direttamente

I metodi nell'oggetto reducer sono le azioni …

Esportazione degli action builder

Esportazione del reducer (per configurare lo store)

icubed

Redux Toolkit - Store

La configurazione dello store avviene tramite il metodo configureStore (che rimpiazza createStore). le slice per configurare lo store …

```
const store = configureStore({
  reducer: {
    theme: themeReducer,
    users: usersReducer
  },
});
```

… specificando le slice che costituiscono lo stato …

```
const store = configureStore({
  reducer: {
    theme: themeReducer,
    users: usersReducer
  },
  middleware: (getDefaultMiddleware) => getDefaultMiddleware({
    thunk: {
      extraArgument: new TodoDataService(Settings.TodoAPIUrl)
    }
  }).concat(logger)
});
```

… e se serve anche i middleware

icubed

Redux Toolkit – Async with Thunk

ReduxThunk può essere utilizzato per definire della azioni asincrone, create attraverso la funzione `createAsyncThunk`

```
// async actions
export const addTodoAsync = createAsyncThunk(
  'todos/addTodo',
  async (newTodo: Todo, { re
    // the code required to
    const svc = new TodoData

    const response = await s

    // The value we return b
    if(response)
      return newTodo;
    else
      return rejectWithValue
  }
);
```

```
extraReducers: (builder) => {
  builder
    .addCase(addTodoAsync.pending, (state) => {
      // if you need a 'loading ...' message, a spinner (state driven)
    })
    .addCase(addTodoAsync.fulfilled, (state, action) => {
      state.todos.push(action.payload);
    })
    .addCase(addTodoAsync.rejected, (state, action) => {
      console.log(action.payload);
    });
```

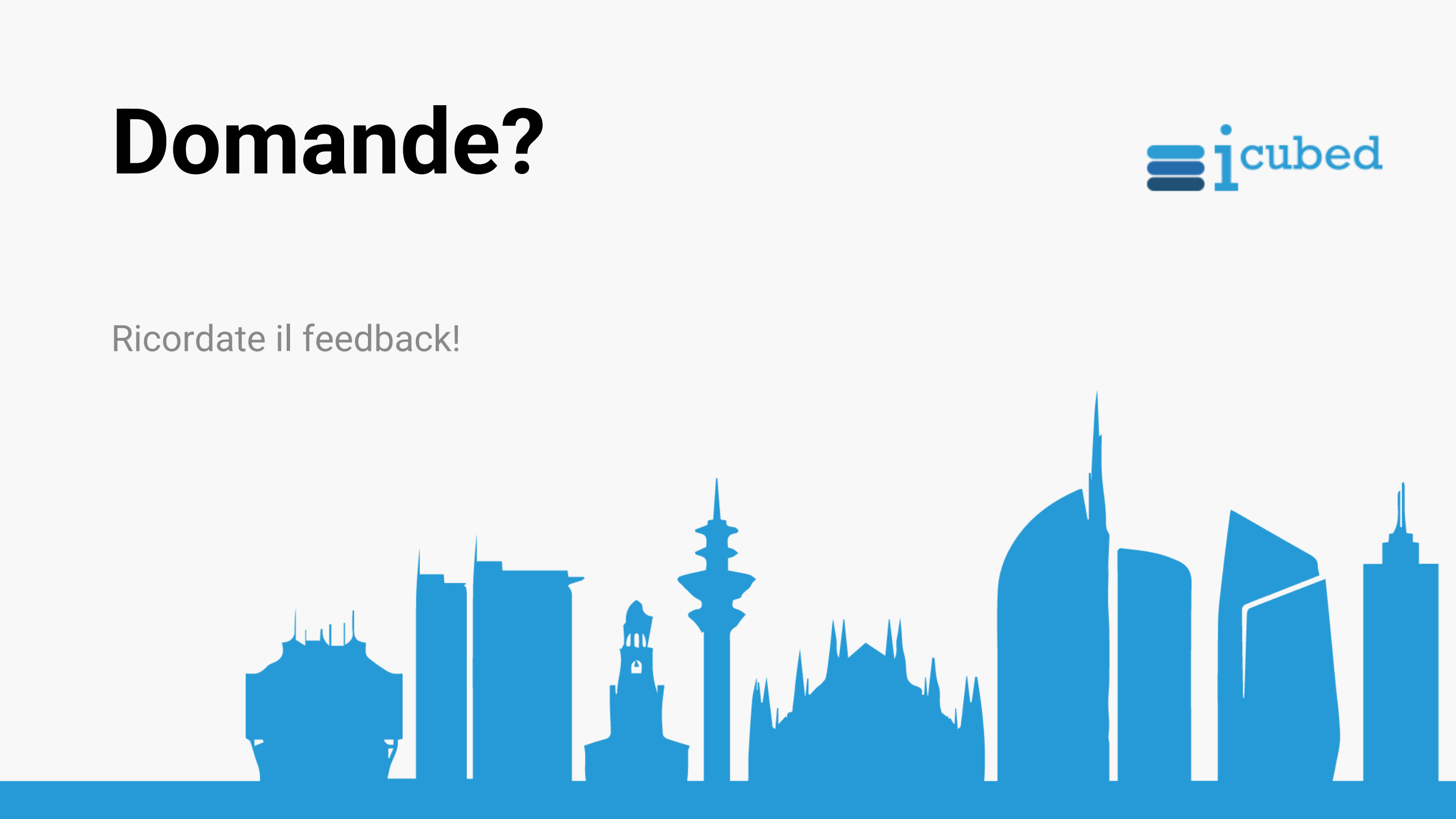
icubed

Demo

Utilizzare Redux Toolkit

icubed

Domande?



Ricordate il feedback!

© 2022 iCubed Srl

iCubed s.r.l.

Piazza Duca D'Aosta, 12 20124 MILANO

Phone: +39 02 57501057

P.IVA 07284390965