

Advanced ReactJS



Roberto Ajolfi

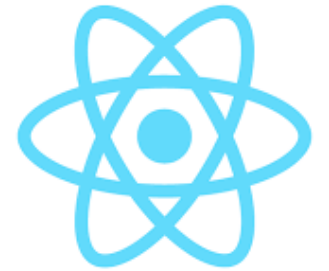
Senior Developer@icubedsrl

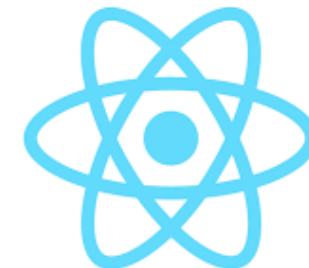
roberto.ajolfi@icubed.it




ReactJS - Agenda

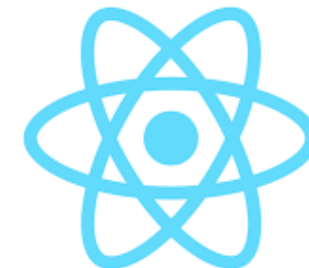
- Introduzione a React
- Javascript / Typescript / JSX
- Components
- Form
- SPA con React
 - REST API
 - Routing with React Route
- Redux / Redux Form





Agenda

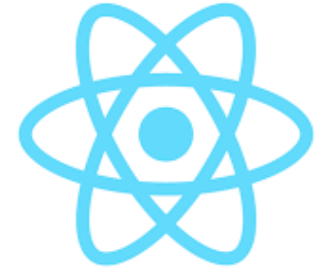
- Components
 - Error Boundary
 - Render Props
 - PropTypes 
 - Component Lazy Loading
 - HOCs (High Order Components)
- Context API
- i18n (Internationalization)
- Portals



Agenda

- REFs in depth
- Memoization
- Hooks
 - Out-of-the-box Hooks
 - Community Hooks
 - Custom Hooks
- Profiler
- Testing (Jest)

Prerequisiti SW



- Visual Studio Code
- Node JS
 - create-react-app
 - http-server
 - json-server

Error Bounding



Come gestire gli errori?



Error Boundaries

- Un errore nel codice JavaScript di una parte della UI non dovrebbe provocare il fallimento dell'intera app
- Per risolvere questo problema, React v16 ha introdotto il concetto di “error boundary”

Error Boundaries

- Gli Error Boundary sono particolari Component che
 - Intercettano gli errori Javascript (in uno qualsiasi dei figli del loro sottoalbero)
 - Permettono di loggare questi errori
 - Visualizzano una 'fallback UI' al posto del Component che ha generato l'errore
- Gli Error Boundary intercettano gli errori durante il rendering, nei metodi di lifecycle e nella costruzione del sottoalbero

Error Boundaries

Gli Error Boundary non intercettano:

- Errori negli Event handler
- Errori nel codice asincrono (es. `setTimeout` o le callback di `requestAnimationFrame`)
- Errori in caso di rendering server side
- Errori generati nell'Error Boundary stesso

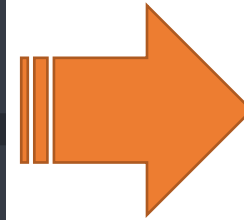
Error Boundaries

Un Component diventa un error boundary se definisce uno o entrambi dei seguenti metodi

- `getDerivedStateFromError()` – metodo static per fare il render di una fallback UI
- `componentDidCatch()` – utilizzato per loggare le informazioni relative all'errore

Error Boundaries

```
class ErrorBoundary extends React.Component {  
  constructor(props) {  
    super(props);  
    this.state = { hasError: false };  
  }  
  
  static getDerivedStateFromError(error) {  
    // Update state so the next render will show the fallback UI.  
    return { hasError: true };  
  }  
  
  componentDidCatch(error, info) {  
    // You can also log the error to an error reporting service  
    logErrorToMyService(error, info);  
  }  
  
  render() {  
    if (this.state.hasError) {  
      // You can render any custom fallback UI  
      return <h1>Something went wrong.</h1>;  
    }  
  
    return this.props.children;  
  }  
}
```



```
<ErrorBoundary>  
  <MyWidget />  
</ErrorBoundary>
```

Error Boundaries

- Gli Error Boundary funzionano come un blocco try / catch, ma a livello di componente
- Solo un class component può essere utilizzato come Error Boundary
- Un Error Boundary intercetta gli errori dei component del sottoalbero
- Un Error Boundary non può intercettare errori che si generano all'interno di se stesso

Error Boundaries

- La granularità degli Error Boundary è una scelta personale
 - Si può incapsulare il componente di routing al top della gerarchia per mostrare un messaggio tipo “Something went wrong” all’utente (come fanno spesso i framework server-side)
 - È possibile altresì incapsulare singoli widget per evitare che possano causare il crash dell’intera applicazione

Error Boundaries

- Gli Error Boundary non intercettano gli errori negli event handlers
 - Al contrario degli eventi di lifecycle e del metodo render(), gli event handler non vengono eseguiti durante il rendering del component. Se accade qualcosa negli event handler React è comunque in grado di aggiornare la UI
 - All'interno di un event handler, va benissimo usare il solito blocco try / catch

Demo

Error Boundaries



Render Props



Riusare funzionalità cross-component



Render Props

I componenti sono l'unità principale di riutilizzo del codice in React, ma non è sempre ovvio come condividere lo stato o il comportamento che un componente incapsula con altri componenti che necessitano dello stesso stato.

Render Props

Dato, ad esempio, un componente che tiene traccia della posizione del mouse in una app, come possiamo riutilizzare questo comportamento in un altro componente?

In altre parole, se un altro componente deve conoscere la posizione del cursore, possiamo incapsulare quel comportamento in modo da poterlo condividere facilmente con quel componente?

Render Props

È una tecnica di sviluppo in React, attraverso la quale un componente condivide funzionalità e informazioni di stato con altri componenti figli.

```
<Hover>
  {(hovering: boolean) => (
    <Info hover={hovering} />
  )}
</Hover>
```

Il suo principale obiettivo è la riusabilità.

Render Props

Un componente “con render prop” accetta una funzione

- come parametro (prop)

```
// Definizione prop render come funzione che li renderizza  
<MyParentComponent render={ () => <MyChildComponent /> } />
```

Render Props

- Oppure come elemento figlio (prop.children)

```
// Definizione elementi figli (children) come funzione che li renderizza
<MyParentComponent>
|   { () => <MyChildComponent /> }
</MyParentComponent>
```

La funzione viene richiamata dal componente stesso nella fase di render e affianca la logica di render interna della vista.

Render Props

Usando una prop per definire ciò che viene renderizzato, il componente inietta

- le funzionalità
- i dati (stato interno, ...)

senza bisogno di sapere come essi vengono applicati all'interfaccia utente.

Render Props

```
class WindowInfo extends React.Component {  
  state = {  
    width: window.innerWidth,  
    height: window.innerHeight  
  };  
  
  handleResizedScreen = () => {  
    this.setState({  
      width: window.innerWidth,  
      height: window.innerHeight  
    });  
  };  
  
  componentDidMount() {  
    window.addEventListener("resize", this.handleResizedScreen);  
  }  
  
  componentWillUnmount() {  
    window.removeEventListener("resize", this.handleResizedScreen);  
  }  
  
  render() {  
    const { width, height } = this.state;  
    return (  
      <div>  
        Window size: {width} x {height}  
      </div>  
    );  
  }  
}
```

Render Props

```
class WindowInfo extends React.Component {  
  state = {  
    width: window.innerWidth,  
    height: window.innerHeight,  
  };  
}
```

```
const PixelAmount = ({ width, height }) => (  
  <div>Total pixels: {width * height}</div>  
);
```

```
componentDidMount() {  
  window.addEventListener("resize", this.handleResizedScreen);  
}  
componentWillUnmount() {  
  window.removeEventListener("resize", this.handleResizedScreen);  
}  
render() {  
  const { width, height } = this.state;  
  return (  
    <div>  
      Window size: {width} x {height}  
    </div>  
  );  
}
```


Render Props

```
class WindowInfo extends React.Component {  
  state = {  
    width: window.innerWidth,  
    height: window.innerHeight  
  };  
  
  handleResizedScreen = () => {  
    this.setState({  
      width: window.innerWidth,  
      height: window.innerHeight  
    });  
  };  
  
  componentDidMount() {  
    window.addEventListener("resize", this.handleResizedScreen);  
  }  
  
  componentWillUnmount() {  
    window.removeEventListener("resize", this.handleResizedScreen);  
  }  
  
  render() {  
    const { width, height } = this.state;  
    return (  
      <div>  
        Window size: {width} x {height}  
        <PixelAmount width={width} height={height} />  
      </div>  
    );  
  }  
}
```

Render Props - render

```
class WindowInfo extends React.Component {  
  <WindowInfo render={({width, height}) => <PixelAmount width="width" height="height" /> }/>  
  // ...  
  handleResizedScreen = () => {  
    this.setState({  
      width: window.innerWidth,  
      height: window.innerHeight,  
    });  
  };  
  componentDidMount() {  
    window.addEventListener('resize', this.handleResizedScreen);  
  }  
  componentWillUnmount() {  
    window.removeEventListener('resize', this.handleResizedScreen);  
  }  
  render() {  
    const { width, height } = this.state;  
    return (  
      <div>  
        Window size: {width} x {height}  
        <PixelAmount width={width} height={height} />  
      </div>  
    );  
  }  
}
```

Render Props - children

```
<WindowInfo>
  ({width, height}) => <PixelAmount width={width} height={height} />
</WindowInfo>
```

```
class WindowInfo extends React.Component {
  state = {
```

```
    handleResizedScreen = () => {
      this.setState({
        width: window.innerWidth,
        height: window.innerHeight
      });
    };
    componentDidMount() {
      window.addEventListener("res
```

```
    }
    componentWillUnmount() {
      window.removeEventListener("
    }
    render() {
      const { width, height } = this
      return (
        <div>
          Window size: {width} x {he
```

```
          <PixelAmount width={width}
        </div>
      );
    }
  }
}
```

```
render() {
  const { width, height } = this.state;
  const { children } = this.props;

  return (
    <div>
      Window size: {width} x {height}
    </div>
    /* <PixelAmount width={width} height={height}></PixelAmount> */
    {
      // @ts-ignore Ignore children error
      typeof children === 'function' &&
      children(this.state.width, this.state.height)
    }
  );
}
```

Render Props

Perché il nome “render prop”?

Il nome risale all’iniziale modalità di implementazione del pattern. Nella pratica, veniva definita una prop con nome “*render*” a cui era assegnata la funzione che specificava, come mostrato in precedenza, la vista del componente.

Il nome del parametro “render” può essere assegnato in maniera arbitraria, in quanto non costituisce una prop speciale.

Render Props

Chi usa “render prop”?

Context API

```
const Green = () => (  
  <div className="box green">  
    <AppContext.Consumer>  
      {(context: any) => {  
        return (  
          <div>  
            <div>{context.number}</div>  
            <button onClick={context.inc}>INC</button>  
          </div>  
        );  
      }}  
    </AppContext.Consumer>  
  </div>  
)
```

```
<Switch>{/* Match a single route */}  
  <Route exact path="/" component={Home} /> |  
  <Route path="/about" component={About} />  
  <Route path="/topics" component={Topics} />  
  <Route path="/details/:id/:pageSize?" component={Details} />  
  <Route comp  
</Switch>
```

```
function FadingRoute({ component: Component, ...rest }) {  
  return (  
    <Route  
      {...rest}  
      render={routeProps => (  
        <FadeIn>  
          <Component {...routeProps} />  
        </FadeIn>  
      )}  
    />  
  );  
}
```

React Router

Demo

Render Props



PropTypes



Tipizzazione di base con Javascript



Prop Types

Man mano che le app crescono, la possibilità di lavorare con codice tipizzato consente di rilevare molti bug.

Se possibile è consigliato utilizzare estensioni JavaScript come Flow o TypeScript per l'intera applicazione.

Ma se non è possibile o non si desidera farlo, React ha alcune abilità di controllo dei tipi integrate.

È possibile utilizzare la proprietà propTypes

Prop Types

È sufficiente aggiungere ad ogni component (Class o Function) la proprietà propTypes:

```
PizzaCard.propTypes = {  
  name: PropTypes.string.isRequired,  
  price: PropTypes.number.isRequired,  
  description: PropTypes.string,  
  toppings: PropTypes.arrayOf(PropTypes.string),  
  discount: PropTypes.bool  
};
```

A questa proprietà (un oggetto) vanno aggiunte le specifiche caratteristiche delle prop usando i membri della classe **PropTypes**

Prop Types

È possibile indicare il tipo specifico della prop ...

```
Component.propTypes = {  
  propName: PropTypes.string  
  // number, bool, object, func, array, ...  
}
```

... oppure una serie di opzioni di tipo ...

```
Component.propTypes = {  
  propName: PropTypes.oneOfTypes([string, number])  
}
```

Prop Types

... limitare i valori della Prop ad un insieme fisso (enum) ...

```
Component.propTypes = {  
  propName: PropTypes.oneOf([true, false, 0, 'Unknown']),  
}
```

... rendere una prop obbligatoria ...

```
Component.propTypes = {  
  propName: PropTypes.any.isRequired  
}
```

Prop Types

... prop di tipo array ...

```
Component.propTypes = {  
  propName: PropTypes.arrayOf(  
    PropTypes.instanceOf(Person)  
  ),  
  
  propName: PropTypes.arrayOf(  
    PropTypes.oneOfType([  
      PropTypes.number,  
      PropTypes.string  
    ]) )  
})
```

Prop Types

... oggetti più generici o istanze ...

```
Component.propTypes = {  
  propName: PropTypes.node,    // un qualsiasi oggetto che si possa visualizzare  
  
  propName: PropTypes.element, // un elemento React  
  
  propName: PropTypes.elementType, // uno specifico elemento React  
  
  propName: PropTypes.instanceOf(Message), // l'istanza di una specifica classe  
}
```

Prop Types

... verificare le proprietà di un oggetto assegnato alla prop ...

```
Component.propTypes = {  
  // Un oggetto con un insieme di proprietà (ma non limitato a quelle)  
  propName: PropTypes.shape({  
    color: PropTypes.string,  
    fontSize: PropTypes.number  
  }),  
  
  // Un oggetto con un specifico insieme di proprietà  
  propName: PropTypes.exact({  
    name: PropTypes.string,  
    quantity: PropTypes.number  
  }), // ...  
}
```

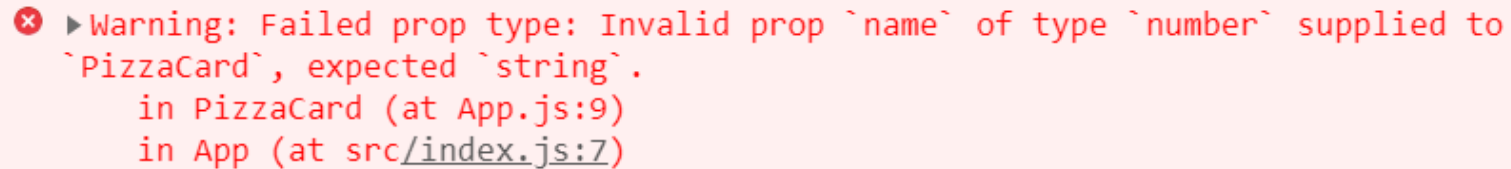
Prop Types

... definire categorie custom di dati (con relativo validatore) ...

```
const isEmail = function(props, propName, componentName) {  
  const regex = /^(((^[^<>()[]\.,;:s@" ]+(.[^<>()[]\.,;:s@" ]+)*)|("[.+""))@((((([0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3}.[0-9]{1,3})|([a-zA-Z0-9]+.[a-zA-Z]{2,})))?)?$/;  
  
  if (!regex.test(props[propName])) {  
    return new Error(`Invalid prop `${propName}` passed to `${componentName}`.  
Expected a valid email address.`);  
  }  
}  
  
Component.propTypes = {  
  propName: isEmail, // ...  
}
```

Prop Types

Quello che si ottiene aggiungendo la proprietà propTypes è una serie di messaggi di warning nella console del browser.



```
✖ Warning: Failed prop type: Invalid prop `name` of type `number` supplied to  
  `PizzaCard`, expected `string`.  
    in PizzaCard (at App.js:9)  
    in App (at src/index.js:7)
```

*Nota: tale comportamento è attivo solo in fase di sviluppo
(Development environment)*

Prop Types

Oltre alle Prop Types è possibile specificare valori di default tramite la proprietà `defaultProps`

```
Component.defaultProps = {  
  propName: 'Stranger'  
};
```

Demo

PropTypes



Component Lazy Loading



Ottimizzare il caricamento dell'app



Component Lazy Loading

Il bundling è il processo che prende i file che compongono la nostra applicazione e li unisce in un singolo file: un "pacchetto".

Questo pacchetto può quindi essere incluso in una pagina Web per caricare contemporaneamente l'intera app.

Component Lazy Loading

Il bundling è molto utile, ma man mano che la app cresce, anche le dimensioni del "pacchetto" aumenteranno.

Occorre tenere d'occhio il codice che viene incluso nel bundle in modo da non renderlo accidentalmente così grande che la app impiega molto tempo a caricarsi.

Per evitare di ritrovarsi con un pacchetto di grandi dimensioni, è bene anticipare il problema e iniziare a "dividere" il pacchetto.

Component Lazy Loading

La suddivisione del codice è una funzione supportata dai bundler come Webpack:

- Si possono creare più bundle, che possono essere caricati dinamicamente in fase di esecuzione
- La suddivisione del codice della app permette di "caricare lentamente" solo le cose che sono attualmente necessarie all'utente

Component Lazy Loading

La suddivisione del codice è una funzione supportata dai bundler come Webpack:

- Non si riduce la quantità complessiva di codice nella app, si evita di caricare codice che l'utente potrebbe non aver mai bisogno
- Si riduce la quantità di codice necessaria durante il caricamento iniziale

Webpack nelle applicazioni generate con create-react-app ha già una configurazione di bundling orientata alla suddivisione del codice. Ma noi possiamo modificare tale comportamento.

Component Lazy Loading

Il modo migliore per introdurre la suddivisione del codice nelle app è attraverso la sintassi dinamica di `import`.

In particolare in accoppiata con la funzione `React.lazy`, che consente di eseguire il rendering di un'importazione dinamica come componente normale.

```
const ComponentA = React.lazy(() => import("./ComponentA"));
```


Component Lazy Loading

Ciò caricherà automaticamente il bundle contenente `ComponentA` al primo rendering di questo componente.

`React.lazy` accetta una funzione che deve chiamare un `import` dinamico. Questo restituisce una promise, che si risolve in un modulo con un'esportazione di default che è un componente React.

```
const ComponentA = React.lazy(() => import("./ComponentA"));
```

```
export default class ComponentA extends Component {  
  render() {  
    const style = { border: "1px solid white", background: "white" };  
    return (  
      <div style={style}>  
        <div style={style}>  
          <div style={style}>  
            <div style={style}>  
              <div style={style}>  
                <div style={style}>  
                  <div style={style}>  
                    <div style={style}>  
                      <div style={style}>  
                        <div style={style}>  
                          <div style={style}>  
                        </div>  
                      </div>  
                    </div>  
                  </div>  
                </div>  
              </div>  
            </div>  
          </div>  
        </div>  
      </div>  
    );  
  }  
}
```

Component Lazy Loading

Il render del componente lazy deve essere fatto all'interno di un componente **Suspense**, che consente di mostrare alcuni contenuti di fallback (es. un indicatore di caricamento) mentre si aspetta il caricamento del componente lazy.

```
<ErrorBoundary>
  <React.Suspense fallback={<div>Loading...</div>}>
    <ComponentA></ComponentA>
    <ComponentB></ComponentB>
  </React.Suspense>
</ErrorBoundary>
```

Conviene piazzare il componente **Suspense** all'interno di un componente di error boundary.

Component Lazy Loading - Routes

Decidere in quale punto della app introdurre la suddivisione del codice può essere un po' complicato.

Bisogna assicurarsi di scegliere luoghi che suddividano i bundle in modo uniforme, ma non compromettano l'esperienza dell'utente.

Un buon punto di partenza è sfruttare le Route.

```
<BrowserRouter>
  <React.Suspense fallback={<div>Loading...</div>}>
    <Switch>
      <Route exact path="/" component={ComponentA}/>
      <Route path="/compB" component={ComponentB}/>
    </Switch>
  </React.Suspense>
</BrowserRouter>
```

Demo

Component Lazy Loading



High Order Components



Ancora sul riuso dei Component



High Order Components (HOCs)

Gli HOC sono simili alle Higher-Order Functions utilizzate ampiamente nella programmazione funzionale.

Un HOC è una funzione che accetta un Component come argomento e restituisce un nuovo Component.

High Order Components (HOCs)

- La funzione utilizzata per gli HOC è una funzione pura
- Inoltre un HOC
 - non modifica il componente che gli viene passato
 - non ha altri effetti collaterali
 - generalmente avvolge il componente passato in un altro per aggiungere un comportamento e/o iniettare alcune props

High Order Components (HOCs)

La funzione può restituire un Function Component ...

```
function withWelcomeHeader<T>(Component: React.ComponentType<T>) {  
  return (props: T) => (  
    <>  
      <h1>Welcome Mr. Component</h1>  
      <Component {...props} />  
    </>);  
}
```

```
function withWelcomeHeaderClass<T>(Component: React.ComponentType<T>) {  
  return class extends React.Component<T> {  
    render() {  
      return(<>  
        <h1>Welcome Mr. Component</h1>  
        <Component {...this.props as T} />  
      </>);  
    }  
  }  
}
```

... oppure un Class Component

High Order Components (HOCs)

Gli HOC possono essere suddivisi in due modelli di base, a seconda dell'utilizzo:

- **Enhancers** (Potenziatori): restituiscono un componente che 'avvolge' quello ricevuto con funzionalità / props aggiuntive
- **Injectors** (Iniettori): iniettano props nel componente ricevuto

Un HOC può rientrare in una o entrambe queste categorie.

High Order Components (HOCs)

Gli HOC sono comuni nelle librerie di React di terze parti

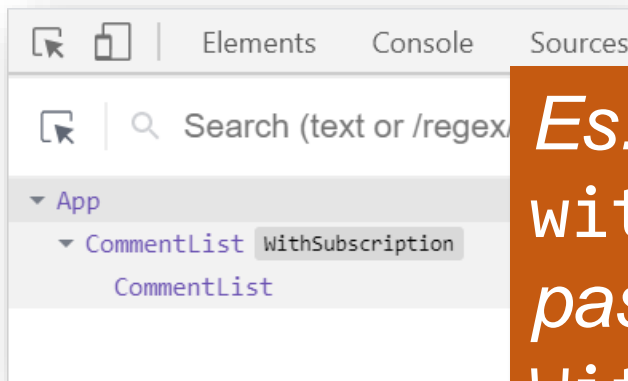
Es. Redux

```
class TodoContainer extends Component<TodoContainerProps, TodoContainerState> {  
  // ...  
}  
  
> const MapStateToProps = (store: MyTypes.ReducerState) => { ...  
};  
  
> const MapDispatchToProps = (dispatch: Dispatch<MyTypes.RootAction>) => ({ ...  
});  
  
export default connect(  
  MapStateToProps,  
  MapDispatchToProps  
) (TodoContainer);
```

HOCs – Best Practices

Modificare il displayName del Component

Per facilitare il debug, è meglio personalizzare il `displayName` del Component generato tramite un HOC in modo che, quando vien visualizzato nei React Development Tools, la sua origine sia evidente.



Es. se la funzione dell'HOC è `withSubscription` e quello del componente passato è `CommentList` ► utilizzare il nome `WithSubscription (CommentList)`

HOCs – Best Practices

I metodi statici devono essere copiati

Quando si applica un HOC a un componente, il nuovo componente non ha nessuno dei metodi statici del componente originale.

Per risolvere questo problema, è possibile copiare i metodi nel contenitore prima di restituirlo; tuttavia, ciò richiede che si conoscano esattamente i metodi da copiare.

È possibile copiare automaticamente questi metodi
utilizzando la libreria
`hoist-non-react-statics`

Demo

HOC (High Order Component)



Gestire lo Stato con Context API



Gestione avanzata dello Stato con le Context API di React



Gestione dello Stato

Tipicamente in una applicazione scritta con React, i dati vengono passati in modalità top-down (dal padre al figlio nell'albero dei Component) tramite le props.

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



```
this.props.title // 'Pay the bill'
```

Gestione dello Stato

Questo può essere scomodo

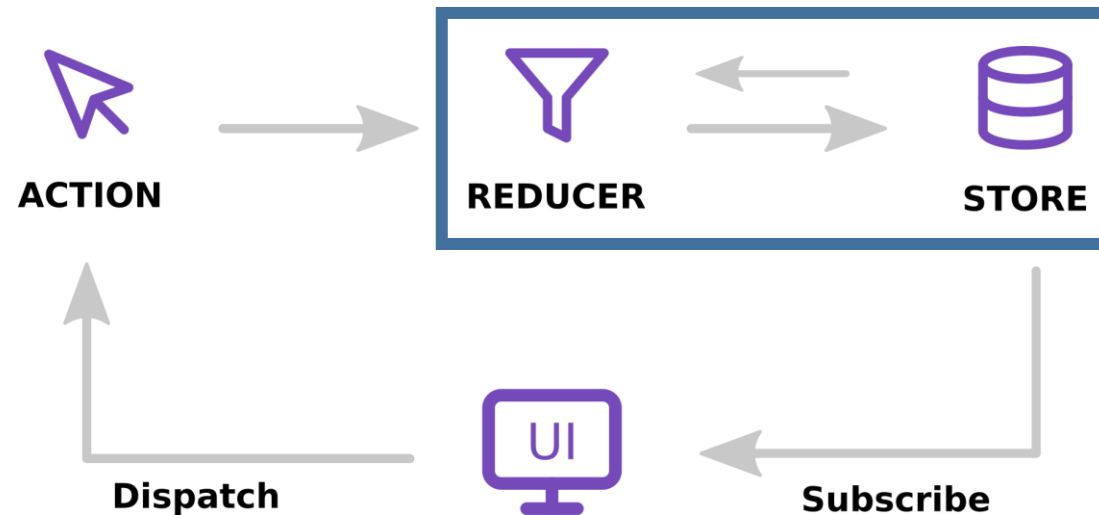
- Al crescere della complessità / profondità dell'albero
- Al crescere degli elementi che necessitano di accedere allo stesso pezzo di informazione

In questi casi può essere utile rivedere la modalità di fruizione e passaggio dei dati, introducendo una gestione avanzata dello stato.

Redux



Una possibile opzione in questo caso è la libreria Redux ...



Context

La Context API fornisce un metodo per passare dati attraverso l'albero dei Component senza dover passare props verso i figli ad ogni livello.

Il Context è stato realizzato per condividere dati che possano considerare 'globali' all'interno dell'albero dei Component, come ad esempio

- I dati dell'utente autenticato
- Il tema della UI
- Il linguaggio selezionato
- ...

Context – Come si usa

1. Creare a context (central repository of data)

```
const MyContext = React.createContext({/* some value */});
```

Context – Come si usa

2. Creare un Context Provider (un componente che funga da wrapper alla parte della app interessata ad utilizzare il Context)

```
<MyContext.Provider value={/* some value */}>  
    // . . .  
</MyContext.Provider>
```

Context – Come si usa

3. I component che desiderano utilizzare i dati del Context, inseriscono la loro logica all'interno di un Consumer Component

```
<MyContext.Consumer>
  {
    (context) => {
      /* some code */
    }
  }
</MyContext.Consumer>
```

Context – Più contesti

```
// Theme context, default to light theme
const ThemeContext = React.createContext('light');

// Signed-in user context
const UserContext = React.createContext({
  name: 'Guest',
});

class App extends React.Component {
  render() {
    const {signedInUser, theme} = this.props;

    // App component that provides initial context values
    return (
      <ThemeContext.Provider value={theme}>
        <UserContext.Provider value={signedInUser}>
          <Layout />
        </UserContext.Provider>
      </ThemeContext.Provider>
    );
  }
}
```

```
// A component may consume multiple contexts
function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}
```

Demo

Utilizzare Context API



Localizzazione (i18n)



Localizzare una applicazione



Localizzazione

La localizzazione di una applicazione permette ad una applicazione di essere fruibile al di là dei confini geografici.

Si tratta di una attività che non comprende solo la traduzione della UI ma anche la gestione dei dati (esempio supremo, la gestione delle date).

Localizzazione

Ci si riferisce a tutte queste attività utilizzando il termine
internationalization

solitamente abbreviato in

i18n

(“**i**+ *nternationalizatio* + **n**” o “*i* (più 18 letters, più) *n*”).

Localizzazione con i18next

In React esistono diverse librerie per gestire la parte di traduzione della UI.

Quella più diffusa è **i18next**

```
$ npm install i18next
```

```
$ npm install react-i18next (integrazione in React)
```

Localizzazione con i18next

- Le traduzioni vengono predisposte in formato JSON (all'interno dell'app o in file JSON esterni).
- Viene attivata la libreria i18next
 - Su function component si utilizza il metodo `useTranslation()`
 - I class component vengono 'marcati' tramite il metodo `withTranslation()`
- In entrambi i casi, le label vengono sostituite da chiamate al metodo `t()`, passando un identificativo univoco
Es. `t('mycomponent_title')`

Demo

Utilizzare i18next



Portals



Componenti a spasso per il DOM



Portals

"Una porta immaginaria o magica che ti consente di spostarti in luoghi diversi attraverso l'universo"

I React Portals offrono un modo per eseguire il rendering di Component in un nodo DOM esistente al di fuori della gerarchia DOM del Component padre.

Portals

Normalmente, quando si restituisce un elemento dal metodo di rendering di un componente, questo viene montato nel DOM come figlio del nodo padre.

Tuttavia, a volte è utile inserire l'elemento figlio in un'altra posizione nel DOM.

Portals

Un tipico caso d'uso per i Portals è quando un componente padre ha nel suo stile overflow: hidden o z-index, ma è necessario che il figlio "rompa" visivamente il suo contenitore.



Portals



Ad esempio

- Finestre di dialogo (Modal)
- Hovercard
- Tooltips

Portals

Per creare un Portal si utilizza la funzione `createPortal`

```
ReactDOM.createPortal(child, container)
```

- `child` è un qualsiasi elemento React che si desidera renderizzare nel portale
- `container` è l'elemento del DOM effettivo in cui si desidera renderizzare `child`

Portals

Il container viene aggiunto al DOM. Spesso come elemento separato da quello in cui si monta il Virtual DOM della applicazione React.



Portals

Esempio di
Componente

```
const modalRoot = document.getElementById('modal-root');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(
      this.props.children,
      this.el,
    );
  }
}
```



Esempio di
Componente
Padre

```
// from parent component ...
render() {
  return (
    <div>
      <Modal>
        <h1>Hello World</h1>
      </Modal>
    </div>
  );
}
```

Portals

I nodi all'interno del Portal fanno parte a tutti gli effetti della struttura di React (Virtual DOM) indipendentemente dalla posizione nella struttura DOM.

Anche se un portale può trovarsi ovunque nel DOM tree della pagina, i nodi al suo interno si comportano come normali elementi React in tutti gli altri modi.

Portals

Ciò include l'event bubbling.

Un evento generato dall'interno di un Portal si propaga agli antenati del Virtual DOM che contiene gli elementi renderizzati, anche se quegli elementi non sono antenati nel DOM tree.

```
<body>

  <div id="root"></div>

  <div id="notification-root"></div> <!-- THE PORTAL !!! -->

</body>
```

Un componente in #root è in grado di catturare un evento non gestito dal nodo fratello #notification-root

Demo

Portals



REFs



Accesso diretto al DOM



REFs

Nel tipico flusso di dati React, le props sono l'unico modo in cui i componenti interagiscono con i loro figli. Per modificare un figlio, esso va renderizzato di nuovo con nuove props.

Tuttavia, ci sono alcuni casi in cui è necessario modificare un figlio tassativamente al di fuori del flusso di dati tipico, sia esso un'istanza di un componente React o un elemento del DOM.

REFs

React supporta un attributo speciale che rappresenta un riferimento associabile a qualsiasi componente. L'attributo `ref` può essere

- un oggetto creato dalla funzione `React.createRef()`
- una funzione di callback
- una stringa *(legacy)*

Non è possibile utilizzare l'attributo `ref` nei Function Component.

Per adesso ...

REFs

Quando l'attributo `ref` è una funzione di callback, questa funzione riceve come argomento (a seconda del tipo di elemento)

- l'elemento DOM sottostante *oppure*
- l'istanza di classe

Ciò consente di accedere direttamente all'elemento DOM o all'istanza del componente.

REFs

Sono solo alcuni i casi d'uso per i refs:

- Gestione del focus, selezione del testo o riproduzione multimediale
- Attivazione di animazioni
- Integrazione con librerie DOM di terze parti

Bisogna evitare di usare refs per tutto ciò che può essere fatto in modo dichiarativo.

REFs

`React.createRef()`
crea una reference
che può essere
collegata agli
elementi React
tramite l'attributo `ref`

```
class MyComponent extends React.Component {  
  constructor(props) {  
    super(props);  
  
    this.inputRef = React.createRef();  
  }  
  
  render() {  
    return <input type="text" ref={this.inputRef} />  
  }  
  
  componentDidMount() {  
    this.inputRef.current.focus();  
  }  
}
```

Quando un ref viene passato a un elemento in render, il riferimento al nodo diventa accessibile nell'attributo `current`

REFs

Il valore di `ref` varia in base al tipo di nodo:

- Quando l'attributo `ref` viene utilizzato su un elemento HTML, la reference creata con `React.createRef()` riceve l'elemento DOM sottostante come proprietà `current`
- Quando l'attributo `ref` viene utilizzato su un Class Component custom, la reference riceve l'istanza montata del componente come proprietà `current`

Demo

REFs



Memoization

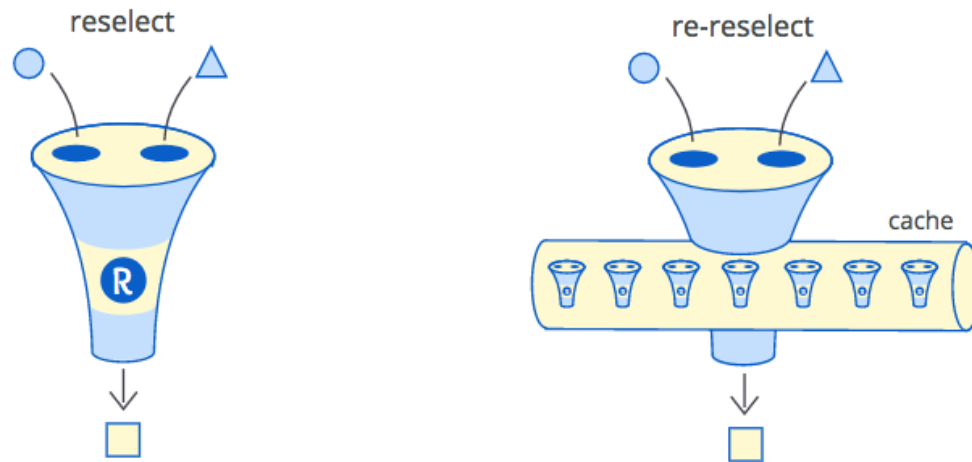


Migliorare le performance



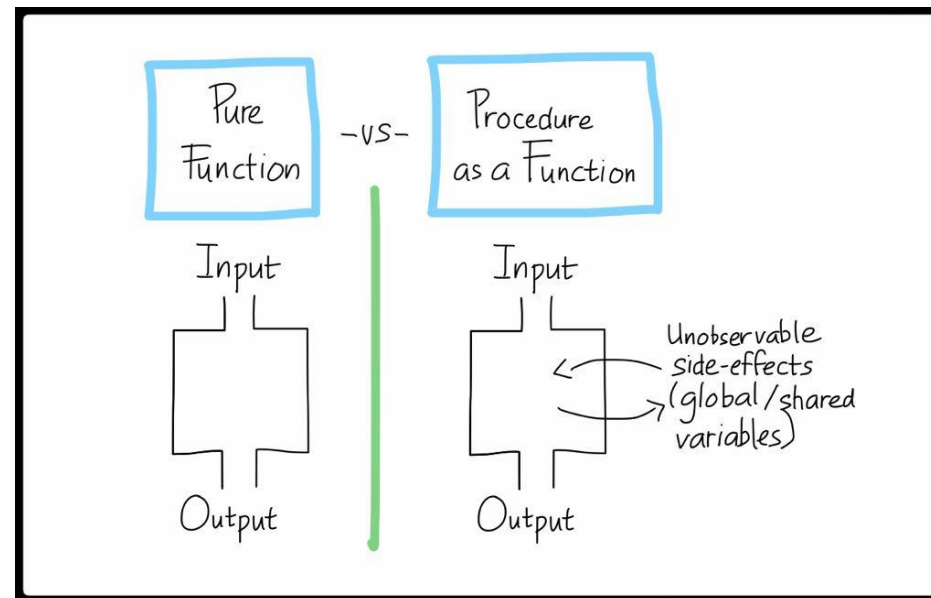
Memoization

La memoizzazione è una tecnica di programmazione che consiste nel salvare in memoria i valori restituiti da una funzione in modo da averli a disposizione per un riutilizzo successivo senza doverli ricalcolare.

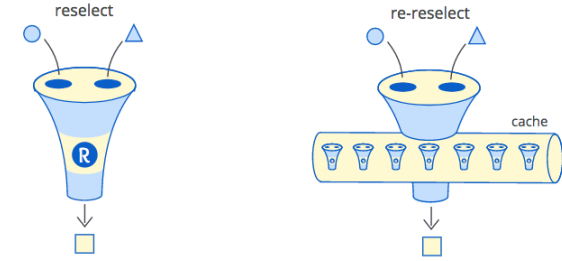


Memoization

Una funzione può essere "memoizzata" soltanto se soddisfa la trasparenza referenziale, cioè se non ha effetti collaterali e restituisce sempre lo stesso valore quando riceve in input gli stessi parametri. Insomma se è una Pure Function.



Memoization



Sebbene correlata al caching, la memoizzazione si riferisce a un caso specifico di questa ottimizzazione e si distingue dalle forme di memorizzazione in cache come il buffering o la sostituzione della pagina.

La memoizzazione è un modo per ridurre il costo in termini di tempo di una funzione in cambio del costo in termini di spazio; cioè, le funzioni memoizzate vengono ottimizzate per la velocità in cambio di un maggiore utilizzo dello spazio di memoria del computer (**Time / Space Tradeoff**).

Memoization in React

In React è possibile applicare il meccanismo di memoizzazione anche ai componenti.

Se un componente mostra lo stesso risultato con le stesse props, si può utilizzare uno dei metodi disponibili per un aumento delle prestazioni memorizzando il risultato del render.

Ciò significa che, fino a quando non varierà il risultato, React salterà il rendering del componente e riutilizzerà l'ultimo risultato renderizzato.

Memoization in React

Class Component

- È sufficiente crearli ereditando da `React.PureComponent`
- In questo modo il componente verrà dotato del metodo di lifecycle `shouldComponentUpdate()`, che farà un controllo superficiale (shallow) delle props
 - `PureComponent` va utilizzato solo quando si prevede di avere dati in props "semplici"

```
class MyComponent extends React.PureComponent {  
  render() {  
    // ...  
  }  
}
```

Memoization in React

Function Component

- È sufficiente wrappare la funzione utilizzando `React.Memo()`

```
const MyComponent = React.memo(function MyComponent(props) {  
  /* render */  
});
```

- In questo modo il componente effettuerà un controllo superficiale (shallow) delle props

Memoization in React

Function Component

- Se si vuole personalizzare il controllo è possibile passare una funzione apposita come secondo parametro

```
function unchangedData(prevProps, nextProps) {  
  /*  
   restituisce true se passare a render() le nextProps restituisce  
   lo stesso risultato che passare le prevProps,  
   altrimenti ritorna false  
  */  
}  
  
export default React.memo(MyComponent, unchangedData);
```


Demo

Memoization



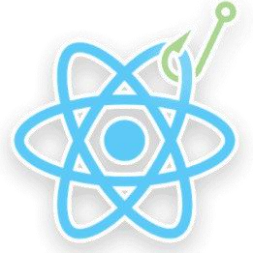
Hooks



Più potenza per i Function Component



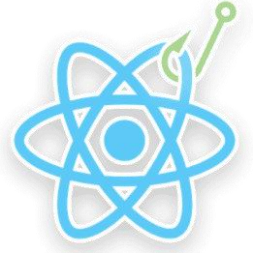
Hooks



Prima di continuare, è importante tenere presente che gli hook sono:

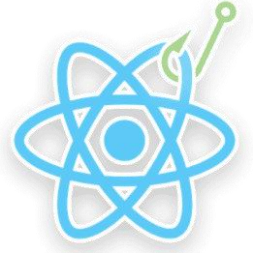
- **Completamente opt-in:** si possono provare gli hook in alcuni componenti senza riscrivere il codice esistente
- **100% retrocompatibili:** gli hook non contengono cambiamenti che non siano retrocompatibili
- **Disponibili dalla versione 16.8.0** di React

Hooks



- **Non ci sono piani per rimuovere le classi da React:** Il team di React ha già definito una strategia di adozione graduale per chi è interessato a questo approccio
- **Gli hook non sostituiscono nessuno dei concetti base di React:** Al contrario, gli hook forniscono un'API più diretta a concetti già noti: props, state, contesto, ref e ciclo di vita.

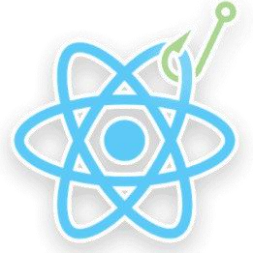
Hooks



Gli hook risolvono una vasta gamma di problemi apparentemente non connessi che il team di sviluppo di React ha riscontrato in cinque anni di attività di scrittura e manutenzione di decine di migliaia di componenti:

- È difficile riutilizzare logiche statefull tra i componenti
- I component complessi diventano difficili da capire
- L'utilizzo delle class confondono sia le persone che le macchine

Hooks



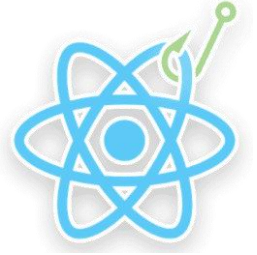
È difficile riutilizzare logiche statefull tra i componenti

- React non offre un modo per "associare" comportamenti riutilizzabili a un componente
- Occorre ricorrere a concetti come render props o HOC
 - Ma questi schemi richiedono di ristrutturare i componenti, il che può essere scomodo e può rendere il codice più difficile da seguire (wrapper hell)

Utilizzando gli Hooks è possibile

- estrarre la logica stateful da un componente in modo che possa essere testato in modo indipendente e riutilizzato
- riutilizzare la logica con stato senza modificare la gerarchia dei componenti

Hooks



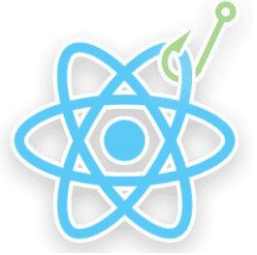
I component complessi diventano difficili da capire

- In molti casi non è possibile suddividere questi componenti
- codice correlato che cambia insieme viene diviso, ma codice completamente non correlato viene combinato in un unico metodo (lifecycle methods)
- molte persone preferiscono combinare React con una libreria di gestione dello stato separata (Redux?!)

Gli Hooks consentono di

- dividere un componente in funzioni più piccole in base a quali pezzi sono correlati
- scegliere di gestire lo stato locale del componente con un reducer per renderlo più prevedibile

Hooks



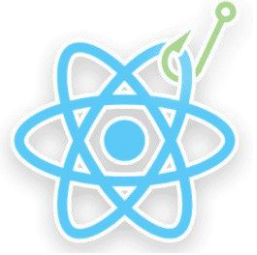
L'utilizzo delle class confonde sia le persone che le macchine

- Le classi possono rappresentare un grosso ostacolo all'apprendimento di React (come funzionano in JavaScript, associare i gestori di eventi, il `this ...`)
- Le classi non si minimizzano molto bene (performance dell'hot reloading)
- La distinzione tra componenti di funzione e classe in React e quando utilizzarli mette in disaccordo anche gli sviluppatori React esperti

Gli Hooks consentono di

- utilizzare più funzionalità di React senza ricorrere alle classi
- non richiedono di apprendere complesse tecniche di programmazione funzionale o reattiva

Hooks



Ma quindi COSA SONO GLI HOOKS?

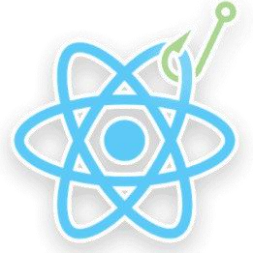
Gli hook sono funzioni che consentono di "agganciarsi" allo stato di React e al ciclo di vita **da un Function Component.**

Demo

Class vs. Hooks

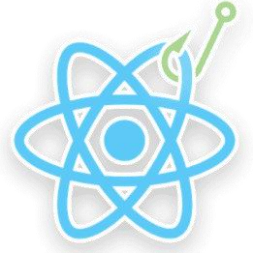


Hooks



React offre già vari hook per diverse funzionalità (out-of-the-box Hooks)

- Basic Hooks
 - useState
 - useEffect
 - useContext
- Additional Hooks
 - useRef
 - useReducer
 - useMemo
 - useCallback
 - useEffect
 - useDebugValue



Hooks - Basic

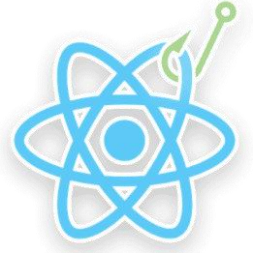
useState

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.**

Hooks - Basic

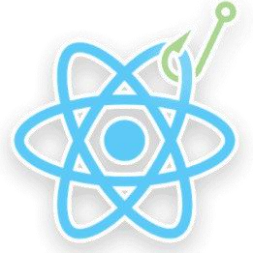


useEffect

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

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

Hooks - Basic

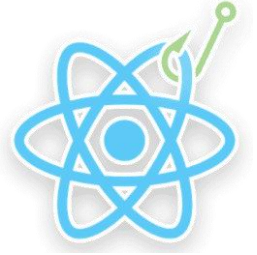


useEffect

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

```
useEffect(() => {  
  // effect code ...  
  return () => { // cleanup code ... };  
})  
componentDidMount + componentDidUpdate + componentWillUnmount
```

Hooks - Basic



useEffect

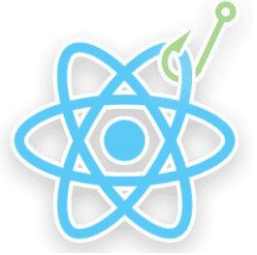
```
useEffect(() => {  
  // effect code ...  
}, [dependsOn, ...])
```

componentDidMount + componentDidUpdate ONLY on [dependsOn, ...] changes

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

componentDidMount ONLY (no updates)

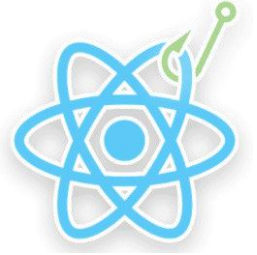
Hooks - Basic



useContext

Viene utilizzato per gestire il contesto (Context API) di React.

```
import { useContext } from 'react'
import { myContext } from 'appContext' // context from Context API
// ...
const context = useContext(myContext)
```

Hooks - Additional

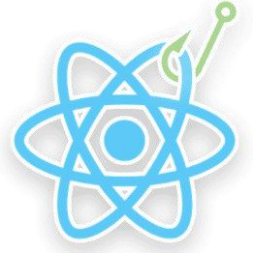
useRef

Restituisce un oggetto ref mutable, in cui la proprietà `.current` è inizializzata usando argomento passato (valore iniziale).

```
import { useRef } from 'react'
// ...
const refComponent = useRef(null)
// ...
<MyComponent ref={refComponent} />
```

La mutazione della proprietà `.current` non provoca un nuovo rendering.

Hooks - Additional

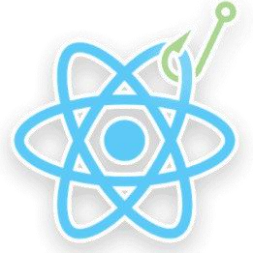


useReducer

Viene utilizzato per gestire logiche di stato complesse in alternativa a useState. Funziona in modo simile a **Redux**.

```
import { useReducer } from 'react'  
// ...  
const [ state, dispatch ] = useReducer(reducer, initialArg, init)
```

Hooks - Additional



useReducer

una funzione di tipo
(*currState*, *action*) =>
newState

il valore
iniziale dello
stato

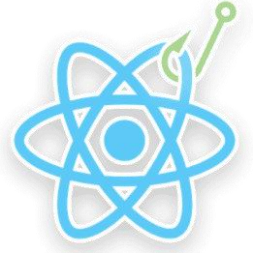
una funzione per
inizializzare lo
stato (lazy
initialization)

```
import { useReducer } from 'react'
// ...
const [ state, dispatch ] = useReducer(reducer, initialArg, init)
```

Lo stato attuale

la funzione utilizzata
per 'inviare azioni' al
reducer

Hooks - Additional

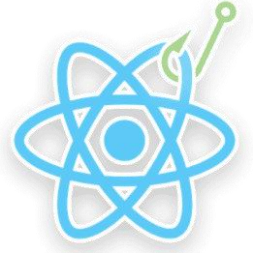


useMemo

Viene utilizzato per sfruttare la tecnica di memoization su un calcolo complesso.

```
import { useMemo } from 'react'  
// ...  
const memoValue = useMemo(() => expensiveCalculation(a,b), [a, b])
```

Hooks - Additional



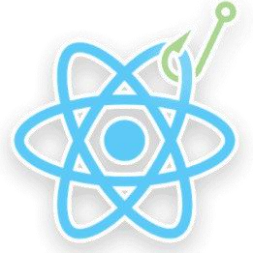
useCallback

Viene utilizzato per ottimizzare il passaggio di funzioni di callback come props a component figli, utilizzando la memoization.

```
import { useCallback } from 'react'
// ...
const callback = useCallback(fn, [a, b]) \\ fn = () => { // ... }

// ... equivale a ...
const callback = useMemo(() => fn, [a, b])
```

Hooks - Additional

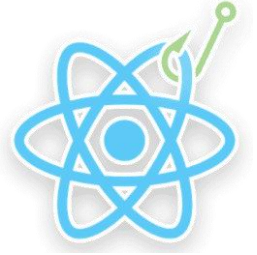


useLayoutEffect

È identico a `useEffect`, ma si attiva (in modo sincrono) solo dopo aver applicato mutazioni al Document Object Model (DOM).

```
import { useLayoutEffect } from 'react'  
// ...  
useLayoutEffect(() => { // effect code ... })
```

Hooks - Additional



useDebugValue

Viene utilizzato per visualizzare un'etichetta nei React DevTools quando si sviluppano hook custom.

```
import { useDebugValue } from 'react'
// ...
useDebugValue(date, date => date.toISOString())
```

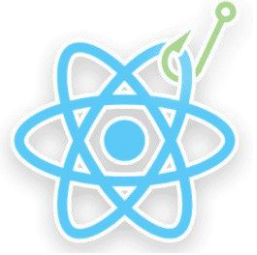
TBD: Add a screenshot of the React DevTools

Demo

Out-of-the-box Hooks



Rules of Hooks

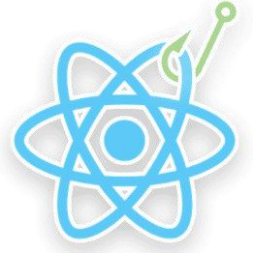


Gli hook sono funzioni JavaScript, ma impongono due regole aggiuntive:

- Non invocare hook all'interno di loop, condizioni o funzioni nidificate
- Invocare hook solo da Function Component. Non chiamare hook da normali funzioni JavaScript

C'è solo un altro posto valido per chiamare un hook: all'interno di un Hook custom.

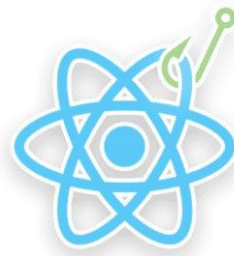
Hooks dalla community



La community di utenti di React si è fin dall'inizio interessata agli Hooks. Esistono moltissimi hook realizzati e messi a disposizione di tutti.

Un esempio è react-hookedup, una libreria con decine di hook già pronti da integrare facilmente nelle app:

<https://github.com/zakariaharti/react-hookedup>



Hooks dalla community

È sufficiente installare la libreria via npm:

```
npm install --save react-hookedup
```

E poi utilizzarla (come visto per quelli out-of-the-box)

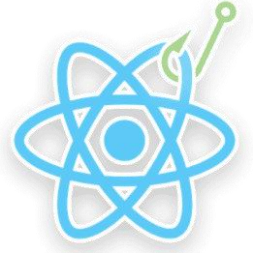
```
import { useArray } from 'react-hookedup'  
// ...  
const { add, clear, removeIndex, value: currentArray } =  
useArray(['cat', 'dog', 'bird']);
```

Demo

Community Hooks



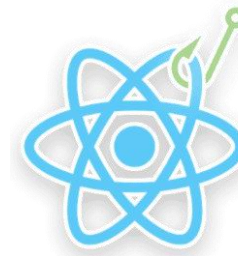
Custom Hooks



Gli hook forniscono un'API diretta a tutti i concetti base di React.
È possibile definire dei custom hook al fine di incapsulare una logica specifica senza dover scrivere un HOC ed evitare il wrapper hell.

Costruire i propri hook consente di estrarre la logica dei componenti in funzioni riutilizzabili.

Rules of Hooks (Parte 2)

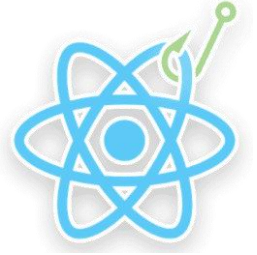


Oltre alle Rules già viste, esiste una convenzione secondo cui
Le funzioni che implementano un hook devono sempre avere un nome che inizia con use, seguito dal nome dell'hook con la prima lettera maiuscola

Es. `useState`, `useEffect`, `useResource` ...

Questa convenzione sulla denominazione non è tecnicamente necessaria, ma rende la vita molto più facile per gli sviluppatori.

Custom Hooks



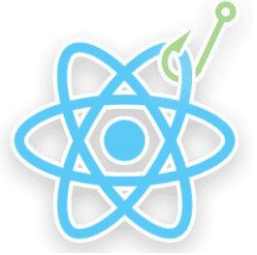
Si possono sviluppare hook personalizzati che coprono una vasta gamma di casi d'uso

- gestione dei moduli
- animazione
- subscription
- timer ...

... E molti altri che non abbiamo preso in considerazione.

Non è difficile creare hook facili da usare
come quelli già integrati di React.

Custom Hooks



Di solito, ha più senso scrivere prima i component e poi in seguito estrarre un hook personalizzato da esso se risulta evidente che un codice simile viene riutilizzato su più component.

Ciò evita la definizione prematura di Hook personalizzati per poi realizzare che si è reso il progetto inutilmente complesso.

Demo

Custom Hooks



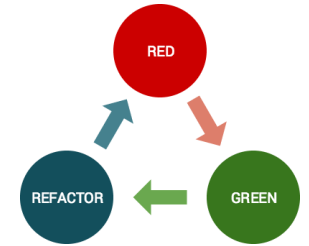
Testing



Verifica del codice e "rete di sicurezza"



Testing

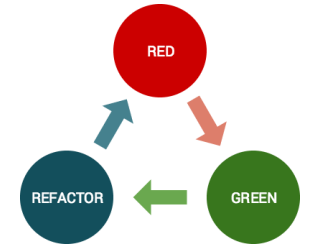


Sviluppare ed eseguire una batteria di test che copra il funzionamento delle applicazioni è fondamentale.

Non importa con quale strategia: *TDD*, *BDD*, *Integration Test*, ... o una qualsiasi combinazione. **L'importante è mantenere i test.**

- È documentazione del codice
- Protezione dalle regressioni
- **Fa risparmiare tempo**
- Favorisce il refactoring
- Irrobustisce l'uso della CI / CD
- Garantisce la qualità del sw

Testing

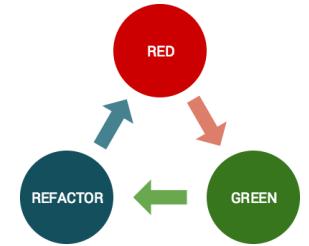


È possibile testare i componenti React in modo simile a quanto si fa con ogni altro codice JavaScript.

In generale, le modalità di esecuzione dei test si dividono in due categorie:

- Rendering del component tree in un ambiente di test semplificato e asserzione sul relativo output
- Esecuzione dell'app completa in un ambiente realistico (noto anche come test *"end-to-end"*)

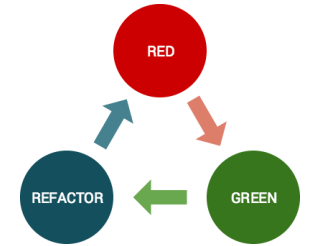
Testing



Quando si scelgono gli strumenti di test, occorre considerare alcuni compromessi:

- **Velocità di iterazione vs ambiente realistico**
 - alcuni strumenti offrono un ciclo di feedback molto rapido tra apportare una modifica e vedere il risultato, ma non modellano con precisione il comportamento del browser
 - altri strumenti utilizzano un ambiente browser reale, ma a scapito della velocità di iterazione e sono più inaffidabili su un server di CI

Testing



Quando si scelgono gli strumenti di test, occorre considerare alcuni compromessi:

- **Quanto ricorrere ai mock**
 - con i componenti, la distinzione tra *unit test* e *integration test* può non essere così netta come in altri casi
 - Se si sta testando una form, il test dovrebbe anche testare i pulsanti all'interno di essa? O il componente Pulsante dovrebbe avere un proprio gruppo di test? Il refactoring di un pulsante dovrebbe far fallire i test della form?

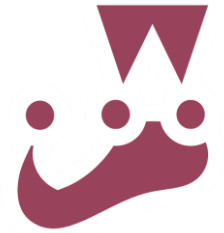
Testing



Jest è un test runner per JavaScript che consente di accedere al DOM tramite **jsdom** (un'approssimazione di come funziona il browser, abbastanza buona per testare i componenti React).

Jest offre una grande velocità di iterazione combinata con potenti funzionalità come mock e timer in modo da poter avere più controllo su come viene eseguito il codice.

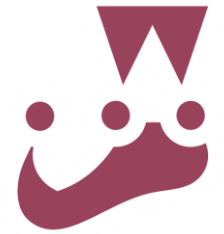
Scrivere test con Jest



Le app generate tramite la CLI `create-react-app` sono già configurate per utilizzare `Jest`:

- È possibile eseguire i test tramite il comando `npm tests`
 - Questo comando funziona in watch mode
 - Include un'interfaccia interattiva con la possibilità di eseguire tutti i test o concentrarsi su un criterio di ricerca
- Viene utilizzata la convenzione di default su posizione e nome dei file contenenti i test

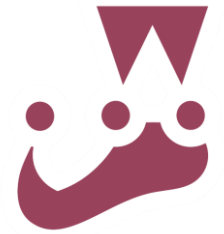
Scrivere test con Jest



Le app generate tramite la CLI `create-react-app` sono già configurate per utilizzare `Jest`:

- Supporto per la realizzazione di mock, che consentono di testare il codice sostituendo l'implementazione effettiva di una funzione / modulo / component da cui tale codice dipende
 - acquisendo le chiamate alla funzione (e ai parametri passati in tali chiamate)
 - catturando istanze di funzioni costruttore quando ne viene creata un'istanza
 - consentendo la configurazione in tempo di test dei valori restituiti

Scrivere test con Jest



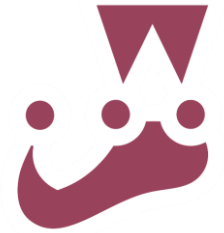
Le app generate tramite la CLI `create-react-app` sono già configurate per utilizzare `Jest`:

- Integra anche un report di code coverage che funziona bene con ES6 e non richiede alcuna configurazione. Per includere il report di code coverage, eseguire il test con il comando

```
npm -- --coverage
```

NOTA: I test verranno eseguiti molto più lentamente in questa modalità, pertanto è consigliabile eseguirli separatamente dal normale flusso di lavoro.

Scrivere test con Jest



- **Jest** cercherà i file di test con una delle seguenti convenzioni di denominazione popolari:
 - File con estensione `.js` nelle cartelle `__tests__`
 - File con estensione `.test.js`
 - File con estensione `.spec.js`
- I file `.test.js` / `.spec.js` (o le cartelle `__tests__`) possono trovarsi a qualsiasi livello nella cartella `src`
 - Si consiglia di inserire i file di test (o le cartelle `__tests__`) accanto al codice che stanno testando in modo che i relativi `import` risultino più brevi

Scrivere test con Jest



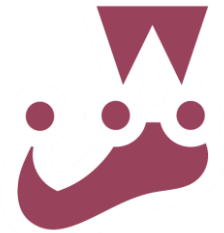
Per simulare l'interazione con l'applicazione **Jest** utilizza **jsdom**, un'implementazione light del browser che viene eseguita all'interno di *Node.js*

Nella maggior parte dei casi, **jsdom** si comporta come un normale browser, ma senza le caratteristiche come il layout e la navigazione.

jsdom permette di modellare le interazioni dell'utente:

- i test possono inviare eventi su nodi del DOM
- quindi osservare gli effetti di queste azioni ed eseguire degli assert sui risultati

Scrivere test con Jest



Scrivere un test e verificare il comportamento (assert)

```
test('desc', () => {  
  // test code ...  
  expect(cond).toBe(value)  
})
```

```
it('desc', () => {  
  // test code ...  
  expect(cond).not.toBe(value)  
})
```

.not nega il matcher

Jest utilizza

- le *"expectation"* (expect())
 - i *"matchers"* (toBe(), toBeTruthy(), toBeNaN() ...)
- per testare i valori in modi diversi.

Scrivere test con Jest



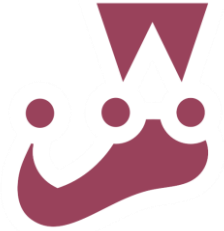
Configurare l'ambiente per ogni test

Per ogni test di un component, in genere si desidera eseguire il rendering in un elemento DOM *dedicato* associato al documento.

Questo è importante in modo da poter evitare interferenze con altri elementi del DOM e per indirizzare precisamente eventuali eventi DOM.

Al termine del test, è buona norma "pulire" e smontare l'albero dal DOM.

Scrivere test con Jest



Configurare l'ambiente per ogni test

Un modo comune per farlo è quello di utilizzare una coppia di blocchi `beforeEach()` e `afterEach()`

```
let container = null;
beforeEach(() => {
  container = document.createElement("div"); // setup a DOM element
  document.body.appendChild(container);
});

afterEach(() => {
  unmountComponentAtNode(container); // cleanup on exiting
  container.remove();
  container = null;
});
```



Scrivere test con Jest

Garantire la correttezza del DOM

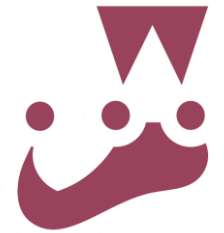
Quando si scrivono test dell'interfaccia utente, attività come

- il rendering
- gli eventi utente
- il recupero dei dati

devono essere considerati come "unità" di interazione con un'interfaccia utente.

React fornisce un helper chiamato `act()` che assicura che tutti gli aggiornamenti relativi a queste "unità" siano stati elaborati e applicati al DOM prima di effettuare eventuali assert.

Scrivere test con Jest

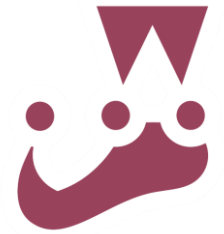


Garantire la correttezza del DOM

```
test('desc', () => {  
  act(() => {  
    render(<Hello />, container); // container from beforeEach()  
  });  
  
  expect(container.textContent).toBe('Hello, my friend');  
});
```

Si utilizza il modulo **react-dom** per manipolare il DOM.

Scrivere test con Jest

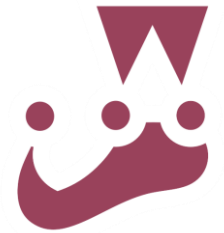


Usare mock per invocare API

Invece di chiamare API reali in tutti i test, è possibile simulare le richieste. In questo modo si impedisce il fallimento dei test dovuto a un back-end non disponibile e li rende più veloci.

```
// restituisce una funzione mock,  
jest.fn();  
  
// opzionalmente si può specificare una implementazione  
let implementation = () => { // container ... };  
jest.fn(implementation);  
  
// restituisce una funzione mock, intercetta le chiamate a object[methodName]  
jest.spyOn(object, methodName);
```

Scrivere test con Jest



Usare mock per generici moduli / component

Alcuni moduli / component potrebbero non funzionare bene all'interno di un ambiente di test o potrebbero non essere altrettanto essenziali per il test stesso.

Realizzare dei mock di questi moduli può rendere più facile scrivere test per il proprio codice.

```
// questo è il componente 'vero' ...  
import MyComponent from "./mycomponent";  
// ... e qui il mock che ne prende il posto  
jest.mock("./mycomponent", () => {  
  return function MockMyComponent(props) {  
    return ( <h1>Hello, {props.name}</h1> );  
  };  
});
```

Scrivere test con Jest

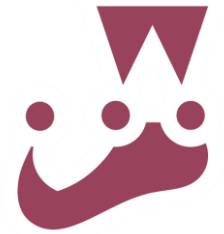


Eventi del DOM

In caso di test con eventi DOM si può ricorrere alla combinazione `act()` + `react-dom` per fare il `dispatch` e in seguito lavorare con `expect()`

```
// into a test ...  
const button = document.querySelector("[data-testid=toggle]");  
expect(button.innerHTML).toBe("Turn on");  
  
act(() => {  
  button.dispatchEvent(  
    new MouseEvent("click", { bubbles: true })  
  );  
});  
  
// ... Go ahead with test ...
```

Scrivere test con Jest

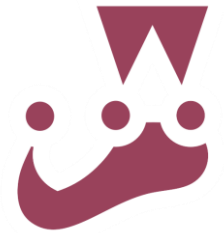


Timer

Se il codice da testare utilizza funzioni basate su timer (come `setTimeout()`), Jest può utilizzare funzioni che consentono di controllare il passare del tempo.

```
// ...
jest.useFakeTimers();
// into a test ...
act(() => {
  jest.advanceTimersByTime(100); // in ms
});
// ... Go ahead with test ...
```

Scrivere test con Jest



Raggruppare i test

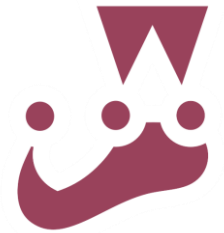
Ogni file che rispetti la naming convention di **Jest** è considerato una **Test Suite**, ovvero un blocco che raggruppa test relativi ad un modulo / component.

È possibile utilizzare anche la funzione `describe()` per raggruppare i test all'interno di una Test Suite.

Non è obbligatorio Ma questo può essere utile se si preferisce avere i test siano organizzati in gruppi.

```
describe('group_desc', () => {  
  test('desc_one', () => {  
    // test code ...  
  });  
  
  //test('desc_two', () => { ...  
});
```

Scrivere test con Jest

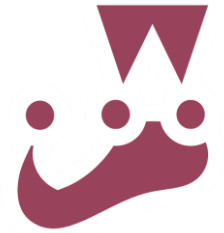


Selezionare i test da eseguire

Per impostazione predefinita **Jest** esegue solo i test relativi ai file modificati dall'ultimo commit (un messaggio esplicito indicherà questo comportamento ad ogni run).

È anche possibile premere **'a'** in modalità watch (*console*) per forzare **Jest** a eseguire comunque tutti i test.


Scrivere test con Jest



Selezionare i test da eseguire

È possibile dare istruzioni a **Jest** su quali test eseguire utilizzando dei metodi di test ad-hoc

- `test.skip()`, il test non viene eseguito
- `test.only()`, viene eseguito solo questo test



Vale all'interno di ogni
singola test suite

Demo

Testing



Esercitazione

https://github.com/roberto-ajolfi/spa_skeleton_adv-react.git

L'elenco delle attività da svolgere si trova nel file README.md

Esercitazione

1. Aggiungere la list view con
 - Open New Ticket (button)
 - Edit / Delete del singolo
2. Aggiungere la form di inserimento
3. Aggiungere la form di modifica
4. Aggiungere la cancellazione di un Ticket
5. Aggiungere un componente di notifica Toastr-like, utilizzando un Portal component
6. Realizzare un componente che usi HOC oppure Render Props oppure Hook per incapsulare la chiamata al servizio dati
7. Aggiungere alcuni test con Jest per validare il funzionamento di alcune parti (a scelta) della applicazione

Tutte le view andranno create come **Function Component**, utilizzando gli hooks dove necessario.

Tutte le funzionalità dovranno utilizzare un servizio che implementa le funzionalità CRUD (GET, POST, PUT, DELETE) comunicando con le REST API disponibili all'URL indicato nella documentazione.

Profiler



Identificare i problemi di performance



Profiler

Lo scopo del Profiler è di aiutare ad identificare parti di un'applicazione che sono lente e che possono beneficiare di ottimizzazioni come la memoization.

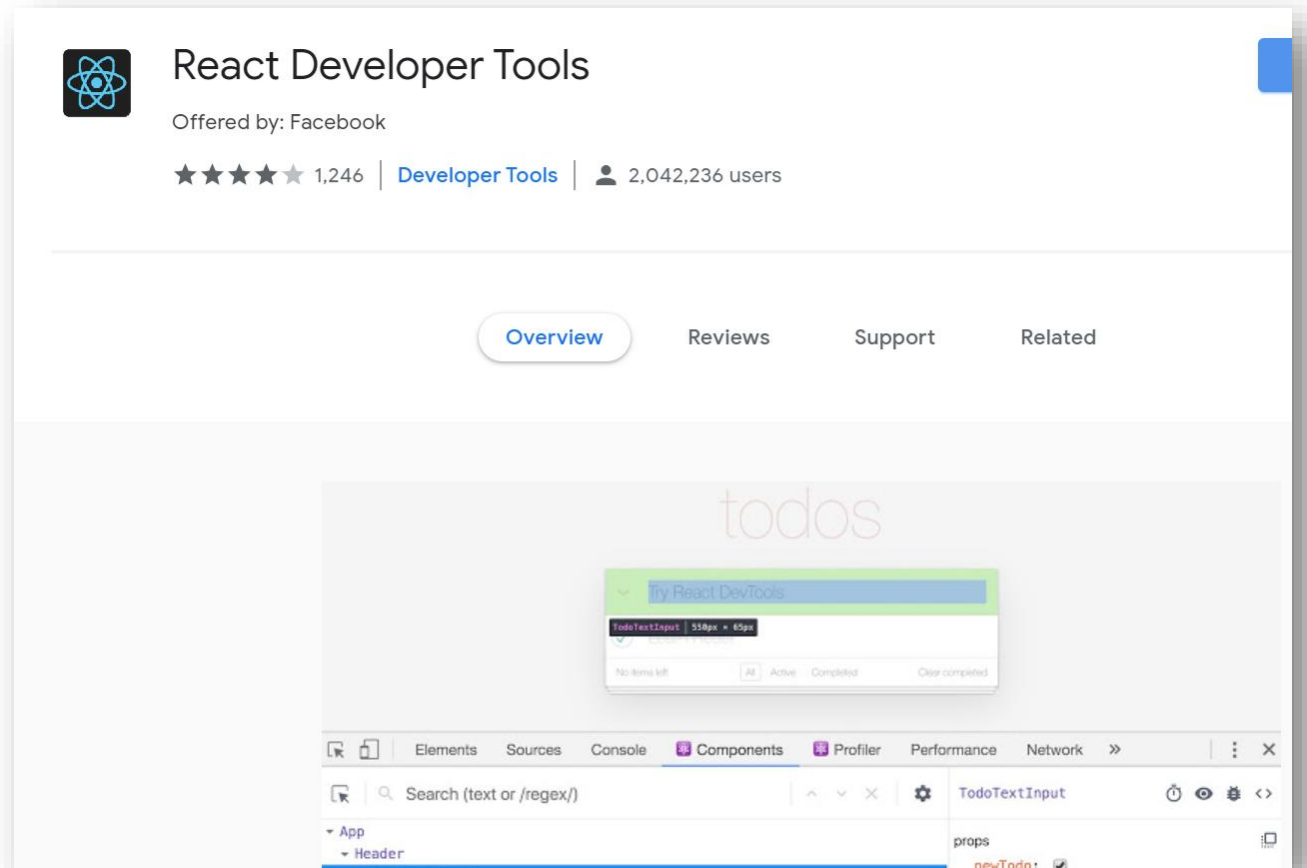
Il Profiler misura la frequenza di rendering di un'applicazione React e qual è il "costo" del rendering.

NOTA: Il Profiler aggiunge un sovraccarico aggiuntivo all'esecuzione, quindi è disabilitata nella build di produzione.

Profiler

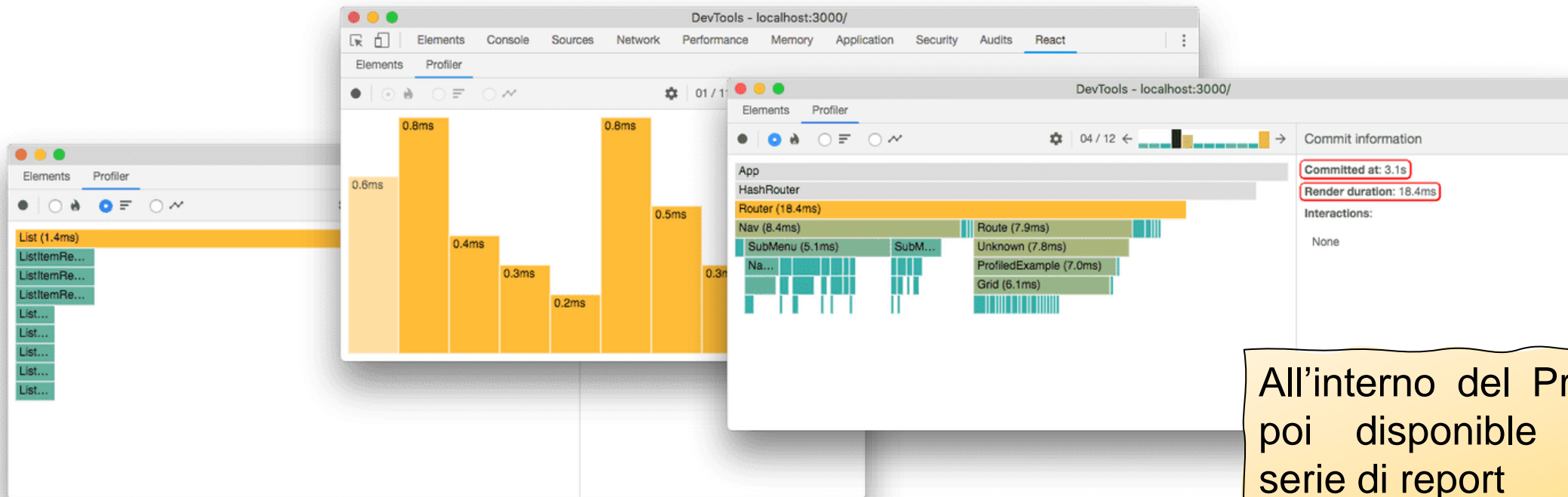
Per utilizzare il Profiler,
occorre installare dal
Chrome Web Store
l'estensione

React Developer Tools



Profiler

Dal Profiler è possibile avviare / arrestare la registrazione dei dati mentre si interagisce con l'applicazione.



All'interno del Profiler sono poi disponibile tutta una serie di report

Profiler API

È l'API utilizzata dal Profiler. Un componente Profiler può essere aggiunto ovunque nel Virtual DOM per misurare il costo del rendering di quella parte della applicazione.

Richiede 2 props:

- un id (di tipo string)
- una callback onRender che React invocherà ogni volta che un componente nella struttura monitorato richiederà un render ("commit")

```
render() {  
  return(  
    <App>  
      <Profiler id="Navigation" onRender={callback}>  
        <Navigation {...props} />  
      </Profiler>  
      <Main {...props} />  
    </App>  
  );  
}
```


Profiler API

- È possibile utilizzare più componenti Profiler per misurare parti diverse di un'applicazione
- I componenti Profiler possono anche essere nidificati per misurare componenti diversi all'interno della stessa sottostruttura

Sebbene Profiler sia un componente leggero, dovrebbe essere usato solo quando necessario; ogni utilizzo aggiunge un certo sovraccarico di CPU e memoria a un'applicazione.

Profiler API

La funzione di callback riceve i parametri che descrivono ciò che è stato renderizzato o e il tempo impiegato.

```
function onRenderCallback(  
  id, // the "id" prop of the Profiler tree that has just committed  
  phase, // either "mount" (if the tree just mounted) or "update" (if it re-rendered)  
  actualDuration, // time spent rendering the committed update  
  baseDuration, // estimated time to render the entire subtree without memoization  
  startTime, // when React began rendering this update  
  commitTime, // when React committed this update  
  interactions // the Set of interactions belonging to this update  
) {  
  // Aggregate or log render timings...  
}
```

Profiler - Opzioni

- **Profiler React Dev Tools:** registrazione e analisi di una sessione di interazione con l'applicazione
- **Profiler API** (componente `<Profiler>`): permette di intercettare le richieste di render e utilizzare una callback per ... "farci qualcosa"
- **Interaction Tracing:** usare la libreria *scheduler/tracing* per definire delle Interaction da visualizzare nel Profiler (React Dev Tools) e da rendere disponibili alla callback delle Profiler API

Demo

Profiler API



Domande?



Ricordate il feedback!



© 2020 iCubed Srl



La diffusione di questo materiale per scopi differenti da quelli per cui se ne è venuti in possesso è vietata.

iCubed s.r.l.

Piazza Duca D'Aosta, 12 20124 MILANO

Phone: +39 02 57501057

P.IVA 07284390965

