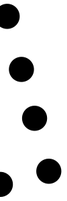
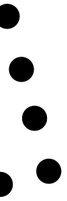
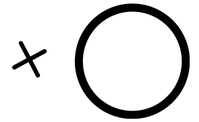
ReactJs Overview



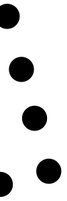
Cos'è REACT





Libreria UI

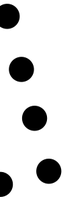
- Rappresenta la View in un pattern MVC (Model View Controller)
- Con l'aggiunta di librerie, può essere assimilabile ad un Framework

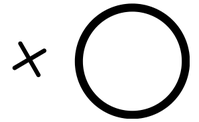


Libreria UI

- Renderizzare Componenti Reattivi
  - `a = b + c`

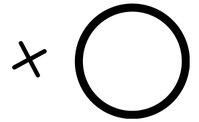
  Imperativo: valore di a non cambia al variare di b o c
  Reattivo: valore di a non cambia al variare di b o c





Libreria UI

- Processo di rendering attraverso Virtual Dom
  - Copia virtuale del DOM
  - Comparazione delle modifiche ad ogni update del DOM (Reconciliation)

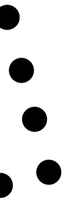


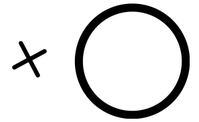
Quando è raccomandato

- Provenienza da Stack HTML / JS / CSS
- Integrare oggetti su progetti esistenti
- Modifica del DOM live
- Contesto di Sviluppo Web App + Mobile App
- Poco tempo a disposizione

Quando si potrebbe NON usare

- Forti nozioni di altri framework
- siti web statici o piccoli
- Sviluppo esclusivamente mobile (Flutter, sviluppo nativo)
- Contesto di Sviluppo Web App + Mobile App
- numero esteso di form



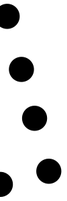


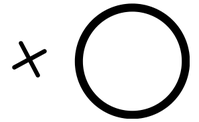
PRO

- Performance
- Standardizzazione
- Produttività
    - Suddivisione in Componenti
    - Riusabilità del codice
- Scalabilità
- create-react-app boilerplate
- Curva di apprendimento lineare

PRO

- JSX
- Disponibilità Libs e plugin
- Facebook / Community
- Unidirectional Data - Flow
    - Stato -> UI
- **It's just Javascript!**





CONTRO

- JSX
- Quantità di codice per piccole operazioni
- Complessità di gestione dello stato generale applicativo
- Scaffolding non standardizzato
    - Non scalabile

Come si usa

Javascript Injecting

```
<script type="text/babel">

  ReactDOM.render(
    <h1>Hello, world!</h1>,
    document.getElementById('root')
  );

</script>
```

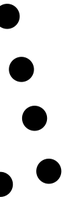
Come si usa

ES6 + JSX

```
import React from 'react';
import ReactDOM from 'react-dom';

ReactDOM.render(
  <h1>Hello, world!</h1>,
  document.getElementById('root')
);
```

React Basics



Elementi

- **JSX**

```
ReactDOM.render(
<h1>Hello, world!</h1>,
document.getElementById('root')
);
```

- **Babel Transpiler**
- **Webpack**

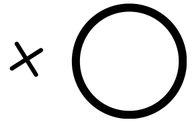
Sintassi JSX

```
import React, { Component } from 'react';
import './App.css';

const helloWorld = 'Hello World';

class App extends Component {
  render() {
    return (
    <div className="App">
       <h2>{helloWorld}</h2>
    </div>);
  }
}

export default App;
```



Sintassi JSX

- Elementi DOM convertiti in istruzioni JSX
- HTML class -> JSX className
- HTML on-click -> JSX onClick

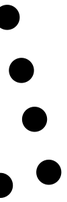
- **REACT DOM Elements**

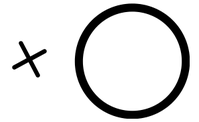
Componenti

Concettualmente, i componenti sono come funzioni Javascript.

Accettano valori in input, definiti **props**, e restituiscono elementi DOM che descrivono cosa deve comparire sullo schermo.

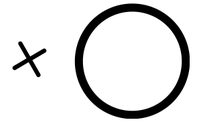
Lo **stato** di un componente, è un oggetto che determina come esso si renderizza e si comporta. In altre parole, è ciò che permette di creare componenti dinamici ed interattivi.



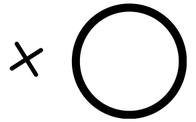


Scopo

- Spezzare applicativo in piccole parti
    - containers - components
    - Single responsibility
- Riusabilità del codice
- Manutenzione del codice
- Scalabilità

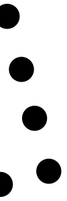


Tipologie di componenti



Functional Components

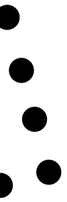
```
const Search = ({ value, onChange, children }) => <form>
      {children}
      <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
```

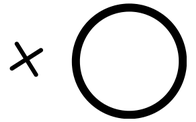


Class Components

```
class Hello extends Component {
    constructor(props) {
        super(props);
    }

    render() {
        const {property} = this.props
        return(
            <div>Hello {property}</div>
        )
    }
}
```



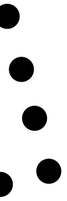


Stateless components

- Non prevedono la gestione dello stato interno
- Possono essere sia Class che Functional Components

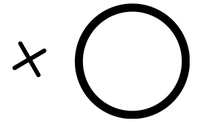
Stateful Components

- Sono sempre Class Components (fino ad oggi)
- Gestiscono il loro stato internamente



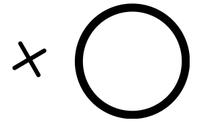
Presentational Components

- Ricevono proprietà dall'esterno
- Possono avere uno stato interno
- Non dialogano con le action o i reducers
- Potrebbero essere Functional Components

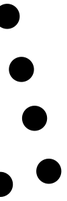


Containers

- Contengono le logiche di funzionamento ed interfacciamento con action e reducers
- Passano le proprietà ai Presentational Components



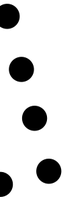
Lifecycle dei Componenti



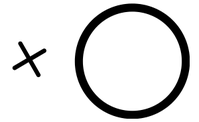
Lifecycle dei Componenti

constructor()

- Chiamata quando viene inizializzato il componente
- Utilizzata per settare uno stato iniziale del componente ed eventuali variabili interne



Lifecycle dei Componenti



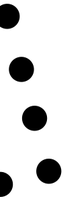
componentWillMount()

- Chiamata prima del metodo **render()**
- Può essere utilizzata per settare lo stato interno iniziale del componente
- E' raccomandato usare **constructor()** per l'inizializzazione dello stato
- Non è raccomandato per eseguire chiamate asincrone ad API esterne

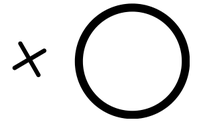
Lifecycle dei Componenti

render()

- **Metodo Obbligatorio**
- Ritorna l'output del componente
- Non deve contenere modifiche allo stato
- Riceve in input lo stato e le proprietà del componente, e ritorna un elemento DOM



Lifecycle dei Componenti



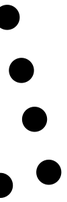
componentDidMount()

- Chiamata una sola volta, dopo il rendering iniziale del componente, definito "mounting"
- Viene solitamente utilizzato per eseguire chiamate asincrone alle API

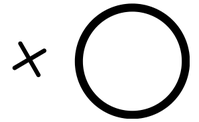
Lifecycle dei Componenti

componentWillReceiveProps(nextProps) - *Deprecated*

- Chiamato durante l'aggiornamento dati
- Riceve in input le proprietà che stanno per essere ricevute dal componente (***nextProps***)
- Viene utilizzato per modificare lo stato del componente in funzione della differenza tra le proprietà attuali e quelle che vengono ricevute (***nextProps***)

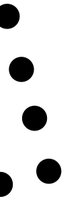


Lifecycle dei Componenti



getDerivedStateFromProps()

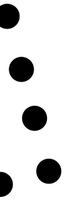
- Sostituisce **componentWillReceiveProps(nextProps)**
- Da usare in combinazione con **componentDidUpdate()**
- restituisce un update dello stato del componente



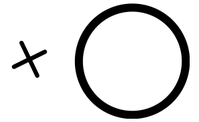
Lifecycle dei Componenti

shouldComponentUpdate(nextProps, nextState)

- Utilizzata per ottimizzare le performance
- Chiamata ad ogni update del componente
- Utilizzata per evitare che il componente si renderizzi quando non necessario



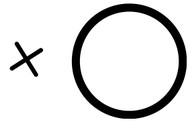
Lifecycle dei Componenti



getSnapshotBeforeUpdate()

- Viene invocata prima di ogni nuovo rendering
- Permette di catturare informazioni dal DOM (come la posizione dello scroll) prima che venga effettuato effettivamente il cambiamento
- Ogni valore restituito da questo metodo viene passato al metodo **componentDidUdpdate()**

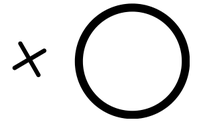
Lifecycle dei Componenti



componentDidUpdate(prevProps, prevState, snapshot)

- Chiamata immediatamente dopo render()
- Non viene invocato nel rendering iniziale
- Utilizzata per effettuare ulteriori operazioni al DOM
- Utilizzata per effettuare ulteriori operazioni asincrone

Lifecycle dei Componenti



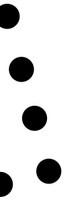
componentWillUnmount()

- Chiamata prima che il componente venga distrutto
- Utilizzato per effettuare operazioni di "pulizia"

Lifecycle dei Componenti

componentDidCatch(error, info)

- Chiamata in tutti i casi in cui il componente crea un'eccezione
- Utilizzata per segnalare messaggi di errore e memorizzare l'errore nello storage



Gestione dello stato

this.setState()

Gestione dello stato

this.setState()

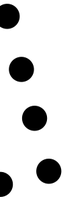
- Un componente può avere uno stato interno

```
class App extends Component {
  state = {
    field: 'value',
    anotherField: 'anotherValue'
  }
}
```

- lo stato è immutabile
- può essere variato solo tramite l'istruzione `this.setState()`
- `this.setState()` accetta come parametro un oggetto che contiene le proprietà da modificare sullo stato:

```
this.setState({field: 'sampleValue'})
```

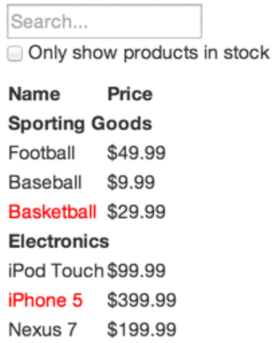
Pensare in React

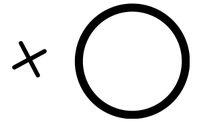


Inizia con un Mock

Immaginiamo di avere una API JSON e un mock grafico da un designer.

Il mock grafico risulta qualcosa di simile:

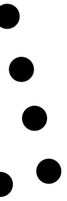




Inizia con un Mock

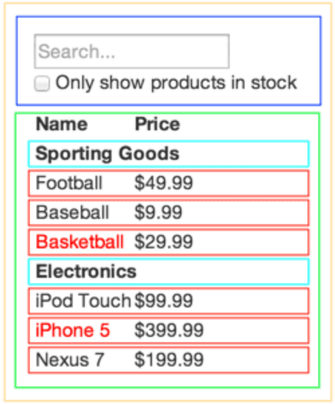
L'API ritorna un dato in JSON formattato in questo modo:

```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},
  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},
  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];
```



Pensare in React

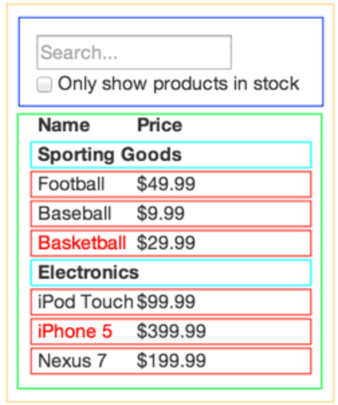
Step 1: Spezzare la UI in una gerarchia di Componenti



- identificare i componenti -> single responsibility
- verificare che la struttura rispetti il data model

Pensare in React

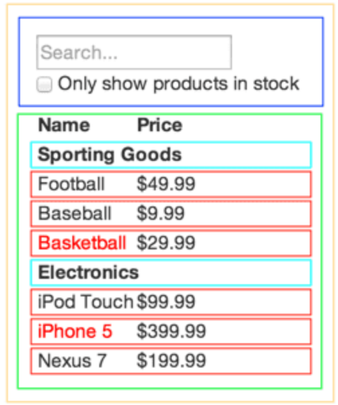
Step 1: Spezzare la UI in una gerarchia di Componenti



- **FilterableProductTable (giallo)**: contiene l'intero esempio
- **SearchBar (blue)**: riceve input dall'utente
- **ProductTable (verde)**: visualizza e filtra i dati in base agli input dell'utente
- **ProductCategoryRow (azzurro)**: visualizza l'intestazione di ogni categoria
- **ProductRow (rosso)**: visualizza una riga per ogni prodotto

Pensare in React

Step 1: Spezzare la UI in una gerarchia di Componenti

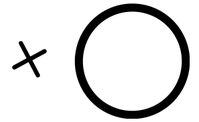


Dopo aver identificato i componenti, devono essere organizzati secondo una gerarchia. I componenti che visivamente risultano interni ad un altro, sono figli a livello gerarchico:

- FilterableProductTable
  - SearchBar
  - ProductTable
    - ProductCategoryRow
    - ProductRow

Step 2: Creare una versione statica in React

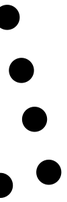
- partendo dal data model creare i componenti
- non implementare interazioni in questa fase
- tralasciare la gestione dello Stato
- passare i dati usando solo **props**
- scopo: avere una libreria di componenti riutilizzabili

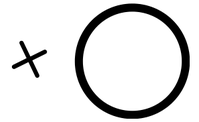


Step 3: Identificare la rappresentazione minima (ma completa) dello Stato

Per far diventare la UI interattiva, è necessario scatenare cambiamenti sul data model (one-way dataflow). E' necessario identificare ciò che è strettamente necessario mantenere nello stato, e calcolare tutto il resto on-demand.

DRY: Don’t Repeat Yourself.





Step 3: Identificare la rappresentazione minima (ma completa) dello Stato

Nell'applicazione abbiamo:

- La lista dei prodotti;
- Il testo cercato dall'utente
- Il valore del checkbox
- La lista filtrata dei prodotti

Per ognuno di questi, chiediamoci:

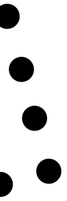
- E' un valore passato da un parent come **prop**? Se si, allora non è un valore dello state.
- Rimane invariato nel tempo? Se si, allora non è un valore dello state.
- Puoi calcolarlo in base ad altre props o elementi dello state del componente? Se si, allora non è un valore dello state.

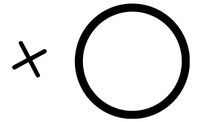
Step 3: Identificare la rappresentazione minima (ma completa) dello Stato

La lista dei prodotti è passata tramite prop, per cui non fa parte dello state; Il testo di ricerca ed il valore del checkbox cambiano nel tempo e non possono essere calcolati, per cui sono parte dello state; La lista filtrata dei prodotti non fa parte dello state perchè può essere calcolata in base al valore del testo di ricerca e del checkbox sulla lista originale dei prodotti.

Quindi, il nostro stato sarà composto da:

- il testo di ricerca
- il valore del checkbox





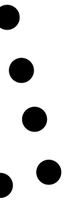
Step 4: Identificare dove lo State deve risiedere

Considerando che:

- **ProductTable** deve poter filtrare i prodotti in base al valore di ricerca
- **SearchBar** deve visualizzare il testo di ricerca ed il valore del checkbox,

Il componente in comune è **FilterableProductTable**, in cui verrà settato lo stato.

- Inizializziamo quindi `this.state = {filterText: '', inStockOnly: false}` nel constructor() di **FilterableProductTable** come stato iniziale dell'applicazione.
- Passiamo **filterText** e **inStockOnly** come props a **ProductTable** e **SearchBar**

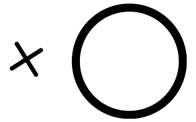


Step 5: Gestire il flusso inverso dei dati

Vogliamo essere sicuri che ogni volta che l'utente modifichi la form, aggiorniamo lo state in modo che rifletta l'input dello user (lo stato è immutabile).

FilterableProductTable deve passare a SearchBar una callback che verrà invocata ogni volta che lo stato necessita di essere modificato.

- **SearchBar** può usare il metodo onChange
- **FilterableProductTable** chiamerà **setState()** per aggiornare lo stato



GRAZIE A TUTTI!