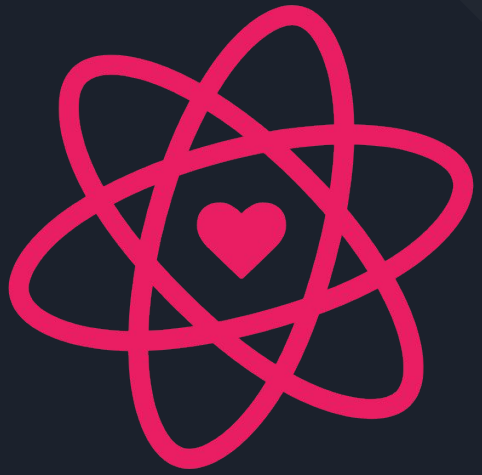
ReactJS



Facebook WebApp library

Perchè un framework/libreria

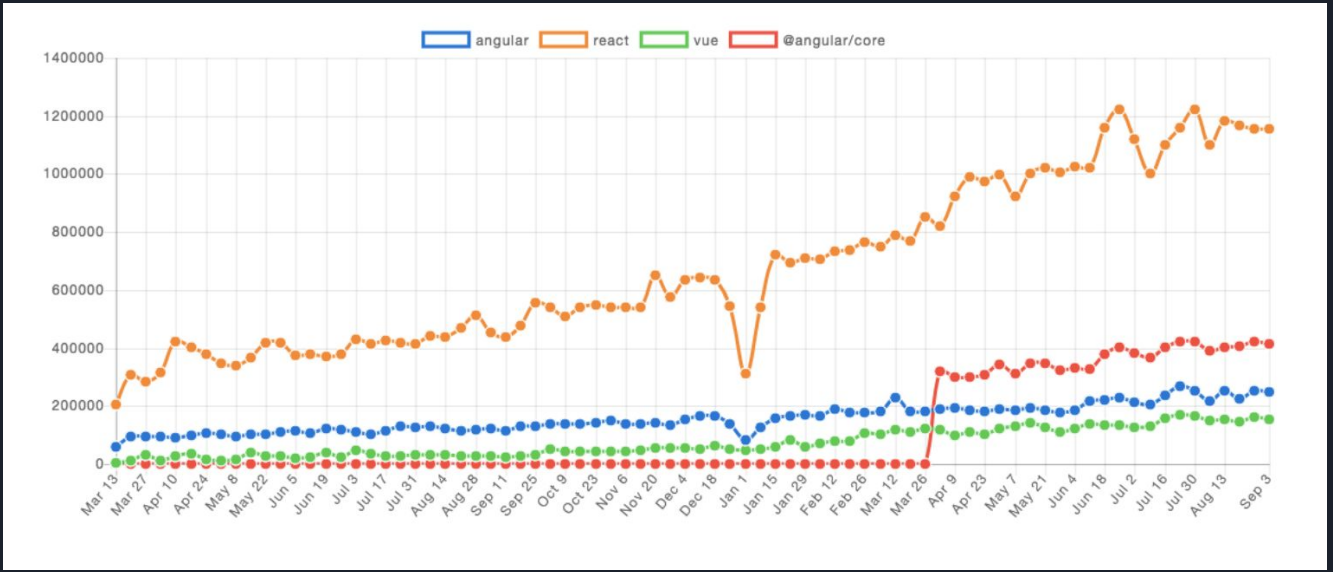
- SPA(Single Page Application-routing)
- Guidelines
- API
- Best practice

ReactJS: Framework o libreria

- Framework vs Libreria:
  - IOC (Inversion Of control): you Call a Library but a Framework Call you
  - Framework contiene una serie di linee guida da seguire oltre alla libreria di base
  - Framework meno libertà, induce all'uso di best practices, libreria più libertà di scelta
- Esempi framework-libreria
  - Framework: Angular, Spring,ecc…
  - Libreria: ReactJS, jQuery, ecc…

ReactJS

- Libreria SPA(Single-Page-Application) sviluppata da Facebook in risposta al framework AngularJS
- Declarative programming: reactJS si basa su uno stile dichiarativo e non imperativo
- Component-based: reactJS induce lo sviluppatore a organizzare l'applicazione in componenti
  - Scalabilità
  - Leggibilità e organizzazione del codice
  - Riusabilità
- Una sola libreria, più dispositivi (ReactJS, React Native, React Desktop)



facebook

TERADEK

Instagram

NETFLIX

KHANACADEMY

reddit

asana

airbnb

Prima app in ReactJS

- CLI creato da Facebook che aiuta ad avere un boilerplate per scrivere un’applicazione in ReactJS:
  - *npm install -g create-react-app || yarn global add create-react-app*
  - *create-react-app* ***<my_app>***

Keywords ReactJS

- Components
- JSX
- VirtualDOM
- Props
- State

Components

- Futuro dello sviluppo web
- Consente la riusabilità del codice
- Più controllo sui singoli moduli che compongono l’applicazione
- Fast development

Components

- *“Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.”*
- Un React component è semplicemente una funzione di JavaScript che prende in input alcuni dati (**Props**) e restituisce un’interfaccia grafica con particolari funzionalità tramite un metodo chiamato **render().**
- Il metodo render() oltre che ad altri lifecycles dei components viene esteso dalla **classe astratta** Component all’interno della libreria ReactJS.
- Più componenti possono interagire tra di loro tramite rapporti di padre-figlio
- L’insieme di tutti i componenti di un’applicazione crea l’applicazione stessa.

Example

- Basic Component project
  - Arrow component
  - Basic function component

JSX(JavaScript eXtension)

- *“Why is there html in my JavaScript? This is a terrible idea. I feel dirty.”*
- Una delle caratteristiche più interessanti di ReactJS è l’utilizzo di JSX
- JSX è un’estensione XML based di JavaScript che consente di scrivere codice HTML in javascript (tramite Babel).
- Ci permette di inserire HTML all’interno di codice JavaScript
- Se vogliamo utilizzare una variabile all’interno di un tag JSX utilizziamo le curly braces {}.

VirtualDOM

- Nato per trovare una soluzione alla lentezza dell'aggiornamento del DOM HTML.
- DOM: è un albero di tutti i componenti HTML
- In ReactJS viene creata una copia "leggera" del DOM chiamata VirtualDOM in JavaScript, che contiene tutti gli elementi che ha il classico DOM, ma non ha il potere di cambiare ciò che c'è sullo schermo
- Ogni volta che un tag JSX viene renderizzato, viene aggiornato il virtualDOM
- Una volta che il virtualDOM viene aggiornato , viene confrontata la nuova versione con uno snapshot creato all'ultimo update (**DIFFING**)
- Se React nota che un elemento del DOM è cambiato, aggiorna solamente quello e non tutto il DOM.

Props

- Dati dinamici passati da un componente padre al component figlio
- Permettono al componente di essere riutilizzabile
- Dato che JavaScript è un linguaggio non tipizzato, React mette a disposizione due funzionalità:
  - **propTypes** → stabilire il tipo delle props che arrivano al component figlio
  - **isRequired** → stabilire se una props è sempre richiesta
  - **defaultProps** → stabilire se qualche props ha un valore di default, se non viene passata dal padre

Example

Basic Props Component ReactJS

State

- Oggetto che rappresenta lo stato in un determinato istante di tempo del componente
- Immutable, per questo motivo non si può modificare re inizializzandolo, ma con un preciso metodo: **setState().**
- Quando lo stato di un componente viene cambiato, viene avviato il **DIFFING** del virtualDOM in modo tale da ri renderizzare il DOM.

Example

- example of state & setState()

Stateless vs Stateful components

- STATELESS: componenti react espressi da funzioni JavaScript. Non possono manipolare lo stato e non contengono lifecycles.
- STATEFUL: componenti espressi da classi ES6(EcmaScript2015). Possono accedere e modificare lo stato del componente e possiedono i seguenti lifecycles:
  - **constructor(special lifecycle)**
  - **componentWillMount(deprecated with React 16)**
  - **componentDidMount**
  - **componentWillReceiveProps(deprecated for static *getDerivedStateFromProps* from React 17)**
  - **componentWillUpdate(nextProps, nextState)**
  - **shouldComponentUpdate(nextProps, nextState)**
  - **componentDidUpdate(prevProps, prevState)**
  - **render**
  - **componentWillUnmount**
  - **componentDidCatch(errorString,errorInfo)**

Focus & Best Practice Lifecycles(1)

- **constructor**:
  - Richiamato quando viene creato un nuovo oggetto JS.
  - Quando richiamiamo un costruttore in React è sempre importante richiamare la keywords super() per inizializzare il nostro componente da React.Component. In questo modo possiamo richiamare this.props,ecc…
    - TO DO
      - inizializzare state e altri variabili utili
- **componentWillMount:**
  - lifecycle deprecato

Focus & Best Practice Lifecycles(2)

- **componentDidMount:**
  - lifecycle chiamato una sola volta subito dopo il render del componente
    - TO DO
      - eseguire HTTP call
      - setState()
- **componentWillReceiveProps(nextProps):**
  - sostituito da React 17 da **getDerivedStateFromProps**
  - chiamato quando il componente riceve nuove props dal componente padre che gliele passa
  - Non viene richiamato al primo render
    - TO DO:
      - controllare il valore di una certa props e quando raggiunge il valore desiderato modificare lo stato del componente in modo da causare un re render.

Focus & Best Practice Lifecycles(3)

- **componentWillUpdate(nextProps, nextState):**
  - triggerata subito prima di un re render del componente
    - DON'T
      - mai chiamare un setState durante componentWillUpdate → causa infiniti re render.
- **componentDidUpdate(prevProps, prevState):**
  - triggerato dopo il re render di un componente
    - DO:
      - aggiornare il DOM dopo una variazione di stato o props

Focus & Best Practice Lifecycles(4)

- **shouldComponentUpdate(nextProps, nextState)**
  - “should I re-render?”
  - metodo che decide se re renderizzare il componente a seconda se torna true o false.
  - Utilizzato per motivi di performance o per evitare re render inutili
    - DON’T
      - setState dentro shouldComponentUpdate
- **componentWillUnmount:**
  - subito prima che avviene l’unmount del componente posso eseguire operazioni in questo lifecycle
  - di solito utilizzato per pulire intervalli creati tramite **setInterval** o **setTimeout.**
- **componentDidCatch(errorString, errorInfo):**
  - nuova funzionalità da React 16
  - Quando si verifica un errore nel render del componente, viene catturato da questo lifecycle
    - errorString: corpo dell’errore
    - errorInfo: stack dell’errore

Routing

- Componente che permette di muoversi all'interno di una SPA, senza ricaricare la pagina
- Content Injecting

```
const BasicExample = () => (
  <Router>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr/>

      <Route exact path="/" component={Home}/>
      <Route path="/about" component={About}/>
      <Route path="/topics" component={Topics}/>
    </div>
  </Router>
)
```