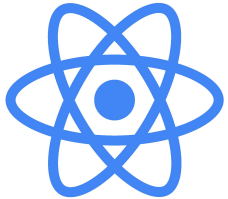
React fundamentals



Tecnologie Internet

a.a. 2022/2023

Gabriele Penzotti

Contacts

- *Gabriele Penzotti*

  Ph.D. Student in Information Technologies

  Department of Engineering and Architecture

  Email: gabriele.penzotti@unipr.it

- *Francesco Saccani*

  Ph.D. Student in Information Technologies

  Department of Engineering and Architecture

  Email: francesco.saccani@unipr.it

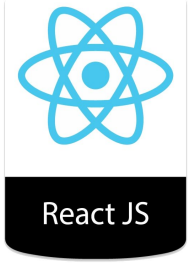
Introduction

Prerequisites

React prerequisites before start to code:

- Basic familiarity with HTML & CSS
- Basic knowledge of JavaScript and programming
- Basic understanding of the DOM
- Familiarity with ES6+ syntax and features
- Node.js and npm basic experience

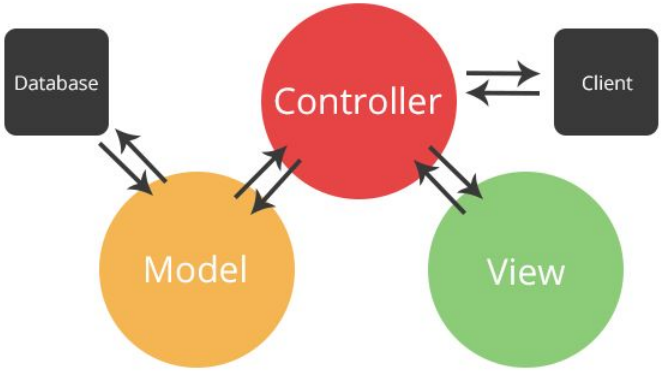
What is React



- *React* is an *open-source project* created by Facebook
- React is one of the most popular JavaScript library to build *User Interfaces* (UI). The term *library* is a bit misleading when compared with other library like *jQuery*
- React is not a framework (unlike *Angular*, or *Vue*), but sometimes is referred as a front-end framework

What is React

- React is the V of an *MVC* application (Model View Controller)
- But also the C (depends on the use made of it)



Install React

We need:

- *React*, *ReactDOM*, *Babel*
- *Nodejs*, *npm* and some other packages (Webpack, etc.)

Several possibilities:

- packages inclusion in HTML header
- online playgrounds
- local installation (Recommended)

Install React

- Packages inclusion in HTML header

  Create an HTML file and using script tag in header to download required packages. Ex: this file from React website

  NB: It does a slow runtime code transformation, so using this is only recommended for simple demos.

- Online playgrounds: CodePen, CodeSandbox, o Stackblitz

- Local installation

  Create a React App using the *Toolchain* provided by React Dev (*Nodejs and npm* required):

  https://github.com/facebook/create-react-app#create-react-app--

  Or using a predefined Docker container :)

```
$ npx create-react-app react-ti21
$ cd react-ti21
$ npm start
```

Install React

File: *package.json*

```
// package.json
{
  "name": "react-ti22",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
      "@testing-library/jest-dom": "^5.16.5",
      "@testing-library/react": "^13.4.0",
      "@testing-library/user-event": "^13.5.0",
      "react": "^18.2.0",
      "react-dom": "^18.2.0",
      "react-scripts": "5.0.1",
      "web-vitals": "^2.1.4"
  },
```

```
"scripts": {
   "start": "react-scripts start",
   "build": "react-scripts build",
   "test": "react-scripts test",
   "eject": "react-scripts eject"
 },
 "eslintConfig": {
   "extends": [
     "react-app",
     "react-app/jest"
   ]
// ...
```

Install React

File: *public/index.html*

meta tag: here you can place other library import, like Bootstrap

Body with a *root* div (class component…)

```
<!DOCTYPE html>
<html lang="en">
 <head>
   <meta charset="utf-8" />
   <link rel="icon" href="%PUBLIC_URL%/favicon.ico" />
   <meta name="viewport" content="width=device-width, initial-scale=1" />
   <meta name="theme-color" content="#000000" />
       ...
   <title>React App</title>
 </head>
 <body>
   <div id="root"></div>
   <script type="text/babel">
   class App extends React.Component{
       render(){ return <h1>hello WORLD!</h1> } }
   ReactDOM.render(<App />, document.getElementById("root"))
  </script>
 </body>
</html>
```

Key Concepts

JSX

The simplest example of React Application (look into *index.js*):

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

```
const container = document.getElementById('app');
const root = createRoot(container);
root.render(<h1>Hello, world!</h1>);
```

React 18

JSX

- JSX (JavaScript + XML) is a widely used extension of JS
- *Markup code* and *logic* are included together in components
- it is ***syntactic sugar*** that allow to insert JS in HTML code and to produce React elements
- Using JSX *is not mandatory* for writing React, but it is widely appreciated. Under the hood, it's running *createElement*, which takes tags, properties, childrens and renders the same information.

JSX translation: https://babeljs.io/repl/
HTML to JSX: https://transform.tools/html-to-jsx

JSX

- JSX is closer to JavaScript than HTML, so there are a few *key differences* to note when writing it:
  - className is used instead of *class* for adding CSS classes, as class is a *reserved keyword* in JavaScript
  - Properties and methods in JSX are camelCase - e.g., *onclick* will become *onClick*
  - Self-closing (no children or content) tags should end in a slash - e.g. <img />
  - ...

JSX

- JavaScript expressions can also be embedded inside JSX using *curly braces* { }, including *variables*, *functions*, and *properties*

```
let btnLabel = "I'm a button"

function dt() {
   date = new Date()
   return "Hello. it's " + date.toLocaleTimeString()
}

const root = ReactDOM.createRoot(
  document.getElementById('root')
);
root.render(
       <button onClick={() => console.log(dt())}>
       Hello, {btnLabel} </button>
);
```

React Element

- An Element is an object that describe what you want to see on screen, i.e. a DOM node or a component, and its desired properties
- It contains only information about type, its properties, and any child elements inside it
- It is an immutable description with two* fields:
  - type (string | ReactClass)
  - props: Object

```
const element = <button className='button button-blue'><b> OK! </b></button>
```

React Element - Rendering

- Renderization is the operation that take React Elements and return DOM tree (***render*** method)
- Usually, the DOM tree of a React app is placed under a *single root node*

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

→

```
const root = ReactDOM.createRoot( document.getElementById('root'));
const element = <h1>Hello, world</h1>;
root.render(element);
```

- React does not update all nodes of the DOM tree, but *only those that are modified*, through a mechanism called Reconciliation

React Element - Rendering

- React use the Virtual DOM (VDOM) paradigm, and use declarative API to define element changes

- Reconciliation is done using an O(n) *diff* algorithm, thanks to some Heuristic rules:

  1. Different component types are assumed to generate substantially different subtrees.
     React will not attempt to diff them, but rather replace the old tree completely.
  2. Diffing of lists is performed using keys prop, implemented by developers.

- It is a diff algorithm, so every element is checked between the previous e the new DOM

React Element - Rendering

OLD

```
<div>
  <Counter />
</div>
```

diff

NEW

```
<span>
  <Counter />
</span>
```

→ All the subtree is updated

OLD

```
<div className="A" title="stuff" />
```

diff

NEW

```
<div className="B" title="stuff" />
```

→ Only classname is updated

OLD

```
<div style={{color: 'red',
             fontWeight: 'bold'}} />
```

diff

NEW

```
<div style={{color: 'blue',
             fontWeight: 'bold'}} />
```

→ Only color is updated

React Element - Rendering

OLD

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>
```

diff

NEW

```
<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

→ Only “third” is updated

OLD

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>
```

diff

NEW

```
<ul>
  <li>third</li>
  <li>first</li>
  <li>second</li>
</ul>
```

→ All the <li> element are updated (and also their subtrees)

React Element - Rendering

- Using the *Key* prop to we could update *list* elements matching their id

OLD

```
<ul>
  <li key="1">first</li>
  <li key="2">second</li>
</ul>
```

diff

NEW

```
<ul>
  <li key="3">third</li>
  <li key="1">first</li>
  <li key="2">second</li>
</ul>
```

→ Only “third” is updated

- Keys should be "stable, predictable, and unique"

React Element - Rendering

NB: The diff algorithm is evolved with *Fiber* by React 16, but the main concepts remain the same.
Key points:

- Not every update is necessary to be applied immediately.
- Different types of updates have different priorities (e.g. animation has top priority).
- Scheduling decide both by the programmerand the the framework (React).

More: https://github.com/acdlite/react-fiber-architecture

NB: In practice, most React apps only call root.render() *once*, and updates are guided by components state changes

React Element - Rendering

Example

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function tick() {
  const element = (
      <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
      </div>
  );
  root.render(element);}

setInterval(tick, 1000);
```

- Element that show the actual time
- *ReactDOM.render* take an element made up of different element and render it
- *setInterval* will re-execute the entire function every seconds
- Only *<h2>* will be updated!

React Component

- Components are another way to create elements. They are *JavaScript functions* or *classes* that return DOM elements. *Nowadays, class are no more used*.

- Components let you split the UI into *independent*, *reusable* pieces, and think about each piece in *isolation*

```
function Greet (props) {
 return <h1>Hello, {props.nome}</h1>;
}
```

```
class Greet extends React.Component {
 render() {
   return <h1>Hello, {this.props.nome}</h1>;
 }
}
```

React Component

- Components can reference one or more other components in their output, using nesting. *Component identification* is a core activity in React development
- Components starts with a *capitalized* letter
- *export default* keywords specify the main component in the file

```
function MyButton() {
  return (
      <button>
      I'm a button
      </button>
  );
}

function MyApp() {
  return (
      <div>
      <h1>Welcome to my app</h1>
      <MyButton />
      </div>
  );
}

const root = ReactDOM.createRoot(
document.getElementById('root'));
root.render(<MyApp />);
```

React Component

- Components could use function parameters to specialize its behaviour

- This introduce PROPS

```
import ReactDOM from "react-dom"; //using CodeSandBox

function Greet(props){
  const nome = props.name;
   return ( <h1> Hello, {nome} </h1> );
}
function MyApp(){
       return (
       <div>
       <Greet name="Mario" />
       <Greet name="Sara" />
       <Greet name="Paola" />
       </div>
       );
}

const root = ReactDOM.createRoot( document.getElementById('root'));
root.render(<MyApp />);
```

Component Props

- We can pass properties to the element *during element creation.* A component can access his properties using the keyword props

- The props are read-only: a method must never modify them - so we want only method as *Pure Function*

- Props flows in one way: *from parent to children*

Component Props

Example using props

```
import ReactDOM from "react-dom"; //using CodeSandBox
const root = ReactDOM.createRoot( document.getElementById('root'));

function Clock(props) {
  return (
  <div>
       <h1>Hello, world!</h1>
       <h2>It is {props.date.toLocaleTimeString()}.</h2>
  </div>
  );
}

function tick() {
  root.render(<Clock date={new Date()} />);
}
setInterval(tick, 1000);
```

Component Props

- However, one basic requirement is missing: Clock is setting a timer and updating its UI every seconds, should be an *implementation detail of Clock*.

- Ideally, we would like to write the following code *once*, and have the Clock update itself:

```
root.render( <Clock />);
```

- To implement this, we need to add a "***state***" to the Clock component.

Component State

- The state (or status) is similar to props, but it is private and *completely controlled* by the component
- Think about state as any data that should be saved and modified *without necessarily* being added to a *database* - for example, shopping cart before confirming your purchase
- State is associated with *Rendering* and *Lifecycle* (more later)

Component State

- State was born for classes:
  - state is the keyword to access them
  - setState the method to update (outside the constructor)

…. But classes are no more used :(

<u>So how to use State with Component Functions?</u>

```
import ReactDOM from "react-dom";//using CodeSandBox
const root = ReactDOM.createRoot(
        document.getElementById('root'));
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
}
  render() {
    return (
      <div>
       <h1>Hello, world!</h1>
       <h2>It is
         {this.state.date.toLocaleTimeString()}.</h2>
      </div>
   );
}}

root.render(<Clock />);
```

never forget where you came from

React Hook

- Hooks were introduced with React version 16.8
- Briefly, they are a way to *use React functionality in any place*
- From a practical perspective, React Hooks are simple *JavaScript functions* that we can use to isolate the reusable part from a functional component.
- Hooks can be stateful and manage side-effects, so function Components could be stateful.
- Note: hooks function start with "*use*"

React Hook

- For example, with the useState Hook you can also use it in *Functions*, making the function component *stateful* as well.
- Normally variables disappear after a function end, but React will preserve this state variables
- useState accept an *argument* (initial state) and returns a *pair* (current state value and a update function)

```
import ReactDOM from "react-dom"; //using CodeSandBox
import React, { useState } from 'react';
function Counter() {
   const [contatore, setContatore] = useState(0);
   return (
   <div>
       <p>Hai cliccato {contatore} volte</p>
       <button onClick={() => setContatore(contatore + 1)}>
           Click me
       </button>
   </div>
   );
}
const root = ReactDOM.createRoot(
document.getElementById('root'));
root.render(<Counter />);
```

React Hook

State Creation

```
function Counter() {
   const [contatore, setContatore] = useState(0);
   const [contatore2, setContatore2] = useState(0);
...
```

State Read

```
<div>
       <p>Hai cliccato {contatore} volte</p>
...
```

State Update

```
<button onClick={() => setContatore(contatore + 1)}>
      Click me
</button>
...
```

React Hook

- Example, with the useEffect Hook
- This hook tells to React to do something after *render*, and will call the argument function after performing the DOM updates
- In this effect we could perform *data fetching*, call some other *imperative API*, …
- Could be also conditionally activated

```
import ReactDOM from "react-dom"; //using CodeSandBox
import React, { useState, useEffect } from 'react';
function Counter() {
   const [count, setCount] = useState(0);
   useEffect(() => {
             document.title = `You clicked ${count} times`;
             console.log(document.title)}); //debug
   return (
   <div>
       <p>Hai cliccato {contatore} volte</p>
       <button onClick={() => setContatore(contatore + 1)}>
           Click me
       </button>
   </div>
   );
}
const root = ReactDOM.createRoot( document.getElementById('root'));
root.render(<Counter />);
```

React Hook

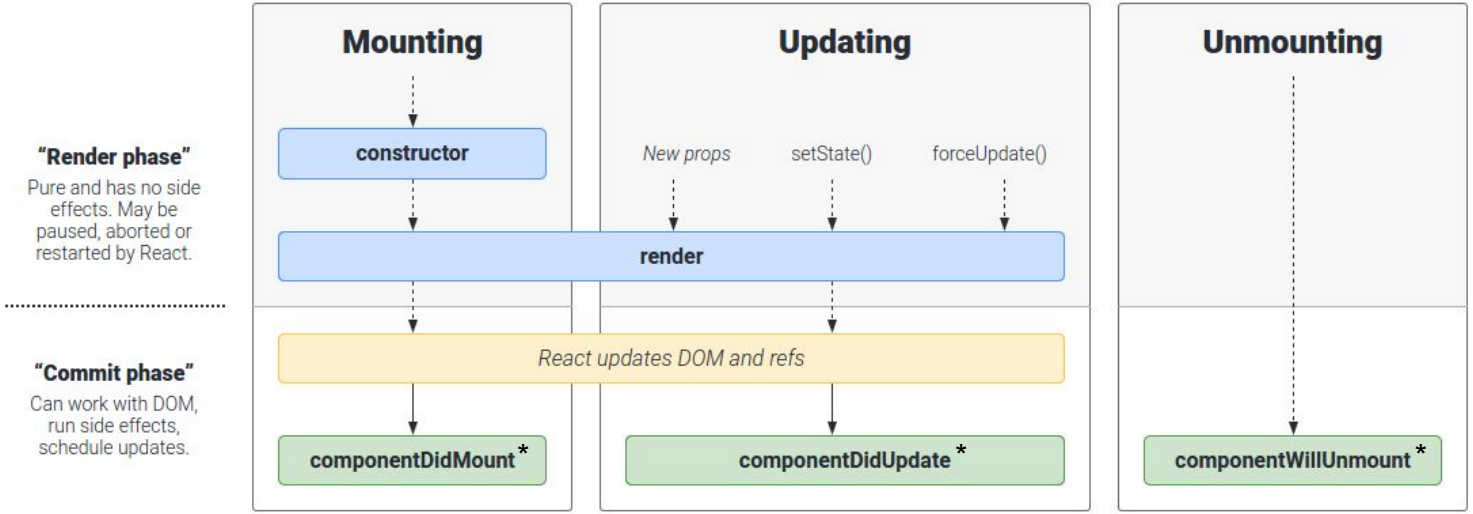
- There are different purpose Hook:
  - ***useState***: state hook, adds the state to functions
  - ***useEffect***: effect hook that adds the ability to perform side effects as componentDidMount, componentDidUpdate, and componentWillUnmount in the React classes, unified under a single API
  - ***useContext***: hook to use Context
  - ***useReducer***: adds the use of reducer function to handle states
  - …
- Possibility to make and use ***Custom Hook***

Lifecycle

- During the life of a web page, various *events* may happen to a *component*
- Each component has the same lifecycle:
  - A component *mounts* when it's added to the screen
  - A component *updates* when it receives new props or state
  - A component *unmounts* when it's removed from the screen

function Component		*class Component*
root.render( )	→	*constructor( )* and *render( )*
useState() useEffect() ...	→	*componentDidMount( ),* *componentDidUpdate( ),* *componentWllUnmout( )* ...

Lifecycle



*useEffect

Event Handlings

- Very similar to handling in DOM elements, but with syntactic differences:
  - React events are declared using *camelCase*, rather than lowercase
  - In JSX, the event handler is *passed as a function*, rather than a string
  - some others…

- Use of cross browser virtual events

```
import ReactDOM from "react-dom";  //using CodeSandBox

function MyButton() {
 function handleClick() {
   alert('You clicked me!');
 }

 return (
   <button onClick={handleClick}>
     Click me
   </button>
 );
}
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyButton />);
```

Component Manipulation

- You can use the JS Control Flow functions to generate components: *conditional* (e.g., *if-else*, ternary, …), *boolean* (!, && or ||), *loop* (e.g., *for*)
- For list generation, the key attribute is recommended for rendering speed up

```
import ReactDOM from "react-dom";  //using CodeSandBox

const products = [
  { title: 'Cabbage', isFruit: false, id: 1 },
  { title: 'Garlic', isFruit: false, id: 2 },
  { title: 'Apple', isFruit: true, id: 3 },
];

function ShoppingList() {
  const listItems = products. map(product =>
        <li
        key={product.id}
        style={{
        color: product.isFruit  ? 'magenta' : 'darkgreen'
        }}>
        {product.title}
        </li>);
  return ( <ul>{listItems}</ul>);
}
const root =
ReactDOM.createRoot(document.getElementById('root'));
root.render(<MyButton />);
```

Component Manipulation

Some tips:

- if two or more components base their output on the same *shared state*, then it must be placed in the *closest common ancestor*. This process is called lifting state up
- If a component becomes too complex, it is a good idea to *individuate and divide* it into distinct components that are more easy to maintain and manipulate
- In React, Component Composition is fast. More "specific" components render the more "generic" version that can then be configured using props and state

Style

- You can use and import CSS stylesheet, like in normal HTML
- It is also possible to use inline CSS using (double) *curly braces* { }, or single if passing through a variable

```
<header>
   <h1 style = {{ color: 'red',
backgroundColor:'black'}}>Hello</h1>
   <h1 style = { headingStyle }>World</h1>
</header>
//...
const headingStyle = {
   color: 'red',
   backgroundColor:'black'
}
```

Context

- Context provides a way to pass data through the component tree without having to pass props down manually at every level.
- It is especially useful for *certain types of props* (e.g. locale preference, UI theme) that are *required by many components*
- Actually, *useContext Hook* is widely used to consume contexts in function

```
import { createContext } from 'react';

const AuthContext = createContext(null);
const ThemeContext = React.createContext('light');

function Button() {
 const theme = useContext(ThemeContext);
 // ...
}

function Profile() {
 const currentUser = useContext(AuthContext);
 // ...
}
```

Suggestions and References

- Read the Official documentation
- Thinking in React: https://reactjs.org/docs/thinking-in-react.html
- JSX in Depth: https://reactjs.org/docs/jsx-in-depth.html
- A lot of video content is available online (pay attention to the version of react it is based on)

Some libraries, React based, are widely used by the dev community:

- https://redux.js.org/
- https://react-query.tanstack.com/

This lesson is mostly based on the official documentation and tutorial:

- https://reactjs.org/
- https://reactjs.org/tutorial/tutorial.html