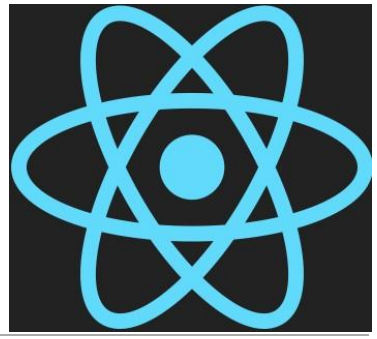
ReactJS



A JS LIBRARY TO BUILD UI

Introduction

- React is a JavaScript **library**, NOT a framework
- React is **DECLARATIVE**: you define the final desired state of your app, you can operate on that and React will manage to render the page components **updating only what is changed**
- React is **COMPONENT-BASED**: components manage their own state, you can combine multiple components to create reach UIs
- We can write React components in plain **JavaScript**, in **JSX** or even in **TypeScript**

What's JSX

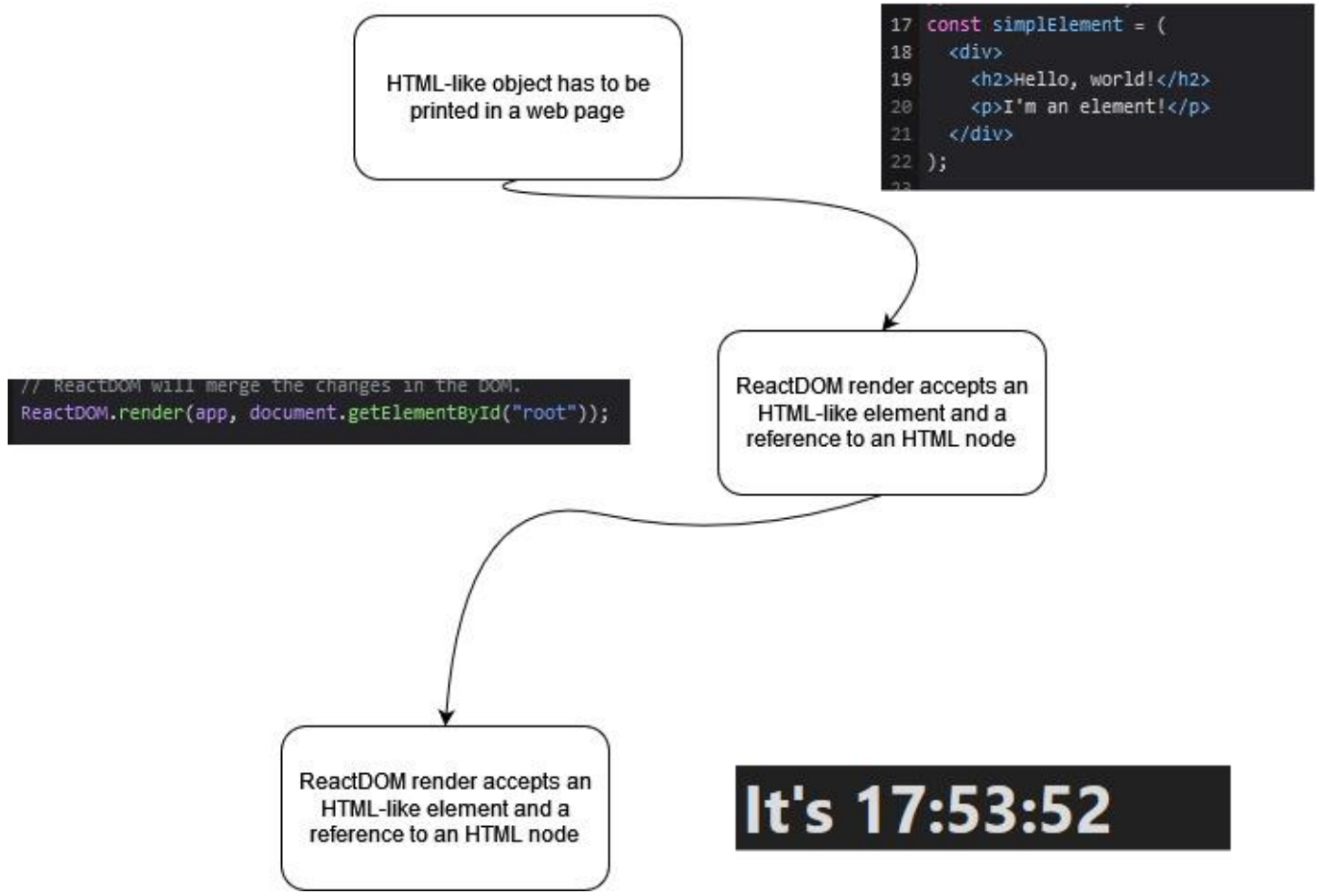
- JSX is an **extension of JavaScript**
- It combines **HTML + JavaScript** to better describe what the UI will look like
- JSX **need to be compiled** in order to be used in a browser, once compiled it becomes plain JavaScript
- JSX is **secure**: React DOM, the engine that renders the UI, escapes any values embedded in JSX before render.
- **JSX represent objects**

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

How it works?

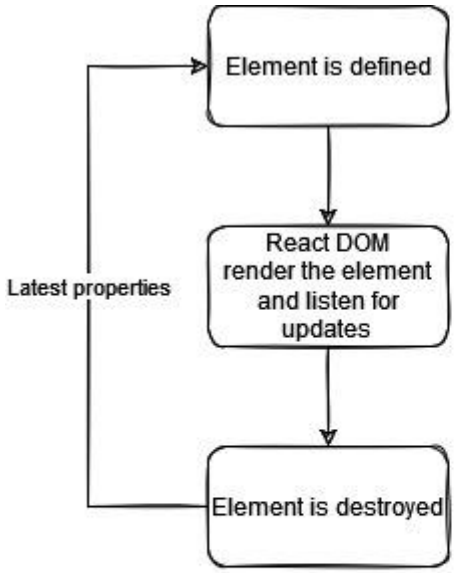
React needs elements to work, those elements are a result of class/functional business logic meant to display useful information for the user in our webapp.

ReactDOM is the main library involved in updating/merging this business logic results in the HTML element associated.

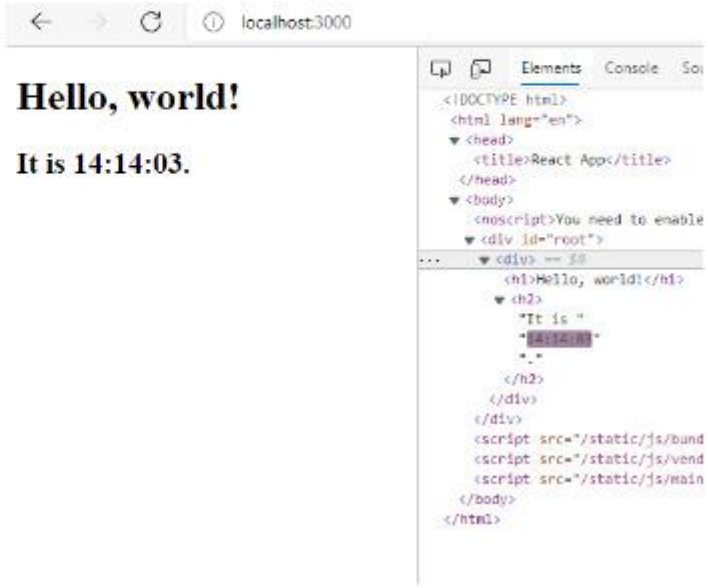


Elements & ReactDOM render

- ReactDOM's render method **controls the content** of a container node <div id="root"...
- **Render accepts: an element and an HTML node**
- **Elements are** the smallest building blocks in React and are **IMMUTABLE**.
- **Elements are** not part of the DOM, they are **objects**
- The **render method** takes care of **updating the DOM**

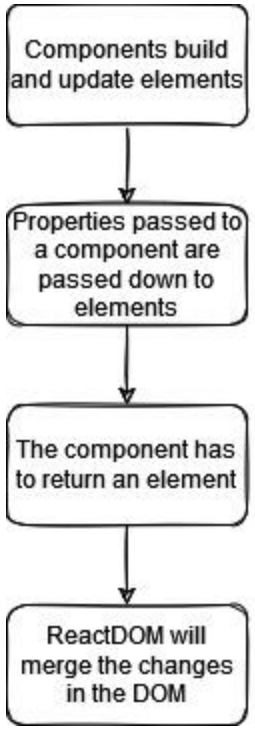



```
const element = <h1>Hello, world</h1>;
```



Components

- Components allow developers to split the UI into **reusable, independent pieces**. Each component is **isolated from the others**.
- Component are **like JavaScript functions**, they accepts arbitrary inputs, called ***Props***, and **return React elements**
- There’re 2 type of components: **function and class components**
- **Props are READ-ONLY**, once you set a value you cannot change it




```jsx
// Function component
function WelcomeComponent(props) {
  return <h1>Hello, {props.name}</h1>;
}

// Class component
class WelcomeClassComponent extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

Stateful

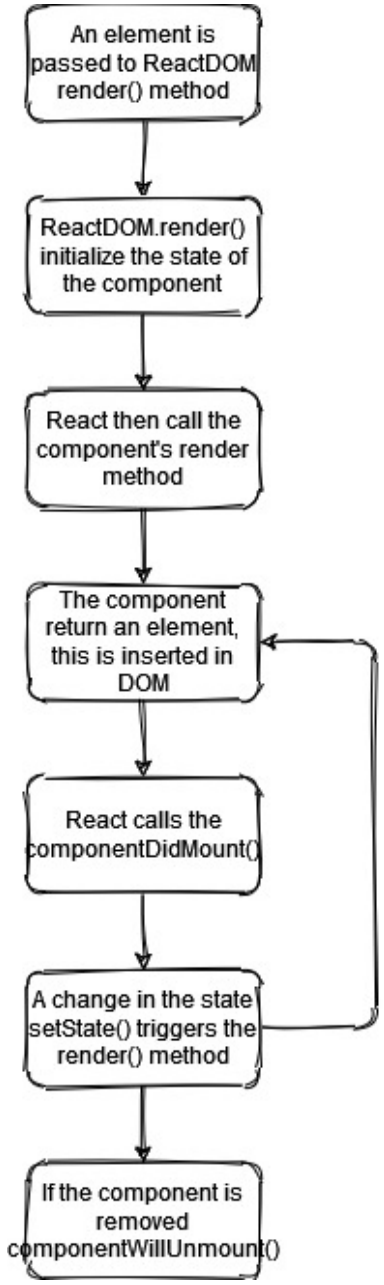
- Lifecycle management, each component's life phase trigger a lifecycle method, you can control each phase
- Store dynamic information in the state
- Are expressed with ES6 class syntax
- Because is a class it can contain other methods

Stateless

- These components are purely presentational: they just return an element to be rendered in the page
- They takes optionally properties (arguments)
- They don't hold any information nor a lifecycle (…see you again in hooks!)
- Are preferred than Stateful components for different reason such as code testing, code maintenance and predictability (a function will return always the same result)

State & Lifecycle of a component

- State is used to store **information that vary during the component lifecycle**. State is used for interactivity.
- Components have methods to manage their lifecycle like **componentDidMount** and **componentWillUnmount**
- **Every change in the state triggers the render method**, this returns an element with the latest information in the state



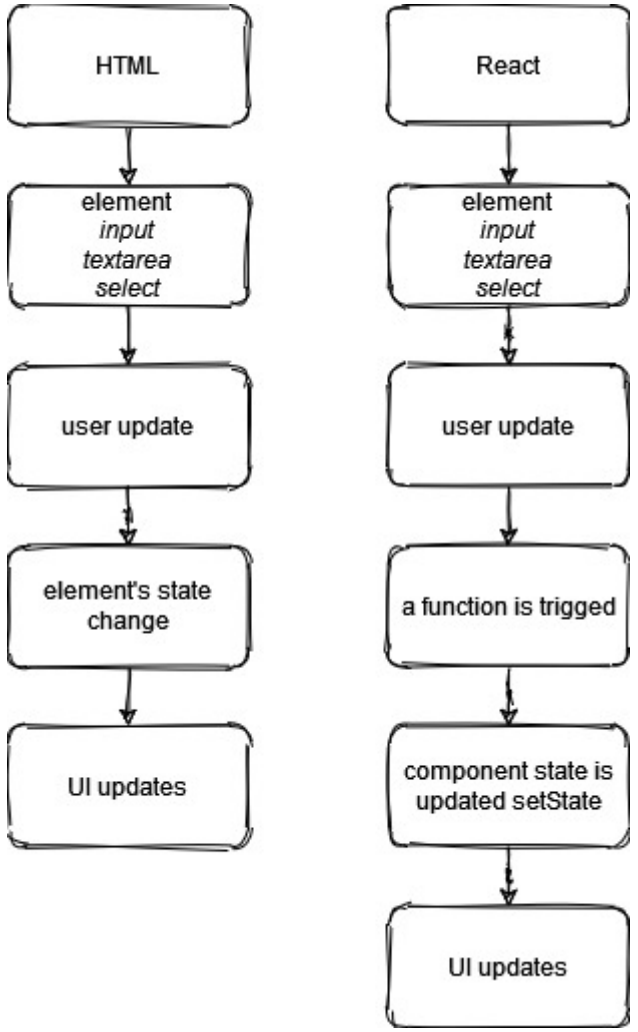
Events

- Events can be triggered by user action, React is able to intercept and handle them
- Main differences with JS/HTML
  - **Event is passed as the pointer to a function, rather than a string**
    - <button onClick="eventListener()">…
    - <button onClick={eventListener}>…
  - You need to **explicitly call preventDefault** if you intend to stop default behavior
- You need to **bind the class scope to the method**, because by default class methods are not bound

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);
```

Controlled components

- Form elements such as *input*, *textarea* and *select* have their own state and update it based on user input
- In React **mutable information are kept in component state** and updated via the setState function
- Therefore elements cited above should update by React following the component's state
- The component's state has to be the **single source of truth**
- An input element whose value is controller by React (component's state) is called a **Controlled component**



Conditional Render

- Like JavaScript if/else statements, conditional render can help us in **displaying UI** components **depending on the component state**
- Conditional rendering can change, hide, show part of the UI
- We can use if/else statements, inline if statements with && operator or inline if-else statement with conditional operator

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

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

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

Lists, Key and map

- The **map(…)** function is broadly used in React, this function return a list of element starting from an array
- When you render a list each element within the list needs a **key** attribute
- Keys helps react in **identifying which element of the list has changed**. Key should be **unique** among siblings
- The key has to be specified each time you have a map/for/or loop that return a list of elements

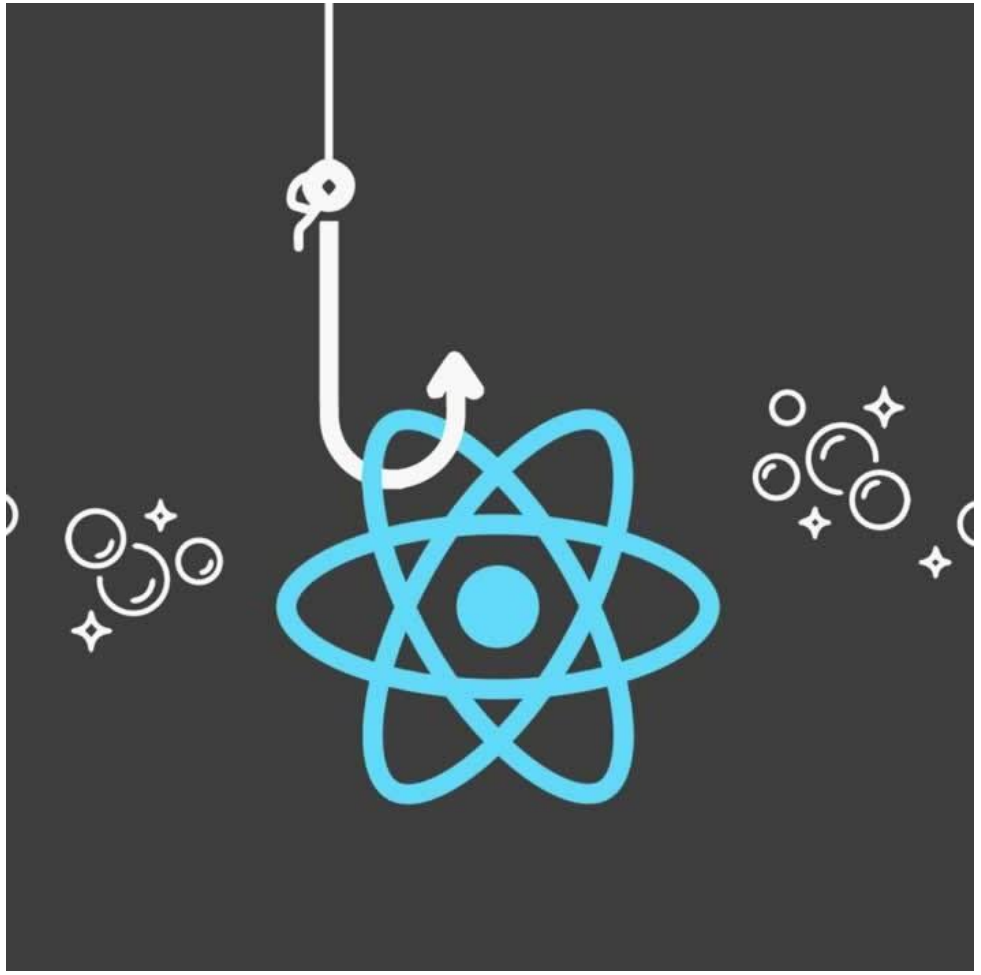
```
const productList = this.products.map((product) => {
  <li key={product.name}>
    {product.name} - {product.price}€
  </li>;
});
```

```
return (
  // Wrong! There is no need to specify the key here:
  <li key={value.toString()}>
    {value}
  </li>
);
```

```
const listItems = numbers.map((number) =>
  // Wrong! The key should have been specified here:
  <ListItem value={number} />
);
```

Hooks

- Hooks allow to **reuse stateful logic** without changing the component hierarchy
- With hooks you can act on components state **without writing a class**, they let you ***hook into*** **react state and lifecycle**
- Important to remember
  - **useState** → create a new state
  - **useEffect** → manage lifecycle, is similar to componentDidMount / componentDidUpdate
- Please note hooks are useful to **share stateful logic not the state**, each component has its own state



Context

- The main idea of context is to **allow your components to access global data** and re-render when it change
- Global data can hold
  - A global state
  - Theme
  - App config
  - User information
  - Collection of services
- **Adding a context increase complexity**, specifically in unit testing where you'll need to wrap your components in a mock context

```
export function ContextProviderHello(props) {
  const helloHook = useHello();
  return (
    <helloContext.Provider value={helloHook}>
      {props.children}
    </helloContext.Provider>
  );
```

Effect Hook

- You can think about effect hook as a **lifecycle management**: componentDidUpdate and componentWillUnmount combined
- As per class methods mentioned above, useEffect is useful to **load API data, setting up subscription or manually changing the DOM**
- There're 2 types of effects
  - Require clean-up → your effect returns a function. Useful in setting up a subscription
  - Don't require clean-up → your effect doesn't return. Useful in API Load, manual DOM changes

```
useEffect(() => {
  document.title = `Hello ${hello.sayHello}`;
});
```

```
useEffect(() => {
  const tick = setInterval(() => {
    setClock(new Date());
  }, 1000);
  return () => {
    clearInterval(tick);
  };
});
```

Create a react app

Open the terminal and run

- *npx create-react-app my-app*
- *npm install react react-dom react-scripts* (before init your app with *npm init*)

Start you app by running *npm run start*

Final points

- Lifting state up is a technique to share the state among different components
- Compositions win over Inheritance
- Use the **single responsibility principle** breaking down the UI in components, each of them having a unique, precise function, rather than a component doing everything
- Think of the minimal set of mutable state your project needs. DRY: Don’t Repeat Yourself.
- Find the right place for your state
- Prefer functional component rather than Class components
- Test your components!