

# Web services

## React

# Contents

## 1. Introduction

## 2. Environment and development tools

### 2.1. Create a React app project

## 3. Main concepts

### 3.1 Hello World Example

### 3.2 JavaScript XML (JSX)

### 3.3 Rendering Elements

### 3.4 Components and Props

#### 3.4.1 Rendering a Component

#### 3.4.2 Composing Component

### 3.5 State and Lifecycle

### 3.6 Handling Events

#### 3.6.1 Passing Arguments to Event Handlers

### 3.7 Conditional Rendering

### 3.8 Lists and Keys

### 3.9 Forms

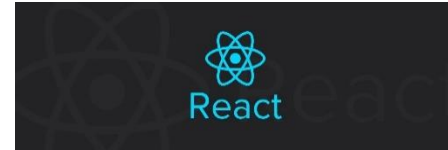
### 3.10 Lifting State Up - Composition vs Inheritance

## 4. Exercise: make your React app

# #1 Introduction

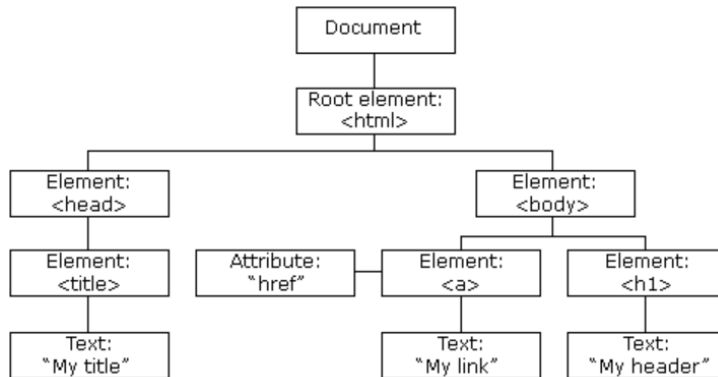
# 1. Introduction

- ❑ React is a JavaScript library for building User Interfaces (UI). It encourages the creation of reusable UI components which present data that changes over time.
- ❑ Traditionally, web application UIs are built using (complex) templates or HTML directives.
- ❑ React approaches building user interfaces differently by breaking these templates of HTML into components.
- ❑ React uses a real, full featured programming language to render views.
- ❑ React is more interesting when your data changes over time.



- ❑ In a traditional JavaScript application, you need to look at what data changed and imperatively make changes to the DOM (Document Object Model) to keep it up-to-date.
- ❑ When the **component** is first initialized, a **render** method is called and generates a lightweight representation of your view.
- ❑ From that representation, **a string of markup** is produced, and injected into the document.
- ❑ When the data changes, the render method is called again.
- ❑ In order to perform updates as efficiently as possible, we diff the return value from the previous call to render with the new one, **and generate a minimal set of changes to be applied to the DOM → This where comes the name of React !**
- ❑ The data returned from render is neither a string nor a DOM node — it's a lightweight description of what the DOM should look like.

HTML DOM Tree of Objects



**HTML DOM** is a standard for how to get, change, add, or delete HTML elements

# #2 Environment and development tools

## 2. Environment and development tools (1/2)

- ❑ Add **React Developer Tools** extensions to your preferred browser.
  - **Firefox:** [https://addons.mozilla.org/fr/firefox/addon/react-devtools/?utm\\_source=addons.mozilla.org&utm\\_medium=referral&utm\\_content=search](https://addons.mozilla.org/fr/firefox/addon/react-devtools/?utm_source=addons.mozilla.org&utm_medium=referral&utm_content=search)
  - **Chrome:** <https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi>
- ❑ Install Node.js (<https://nodejs.org/en/download/>) → runtime environment for executing Javascript code outside the browser. (**node -version**).
- ❑ We will use the Node.js'built-in tool which is Node Package Manager (npm) to install third party libraries by running **npm i -g create-react-app** command in your terminal.
- ❑ Use your preferred code editor. If you choose Visual Studio Code you can install it from here: <https://code.visualstudio.com/>
- ❑ Add VS code to the path to ease open it from the terminal (press Shift command +P for Mac users or Shift + CTRL + P) → **install code command in path**.
- ❑ Install two extensions in VS code to make easier built React applications with this editor.
  - **Simple React Snippet** (dev. by Burke Holland).
  - **Prettier - Code formatter** (Code formatter using prettier). Need to enable Formatting On Save. Whenever you save the changes this extension will automatically reformat your code → Go to VS Code Settings == > Search bar > “**editor: format on save**”. Make this feature ON.

```
c:\>npm i -g create-react-app
C:\Users\> npm install -g create-react-app
npm WARN deprecated create-react-app@4.0.1
+ create-react-app@4.0.1
added 67 packages from 25 contributors in 3.262s
```

```
New patch version of npm available! 6.14.8 -> 6.14.10
Changelog: https://github.com/npm/cli/releases/tag/v6.14.10
Run npm install -g npm to update!
```

## 2. Environment and development tools (2/2)

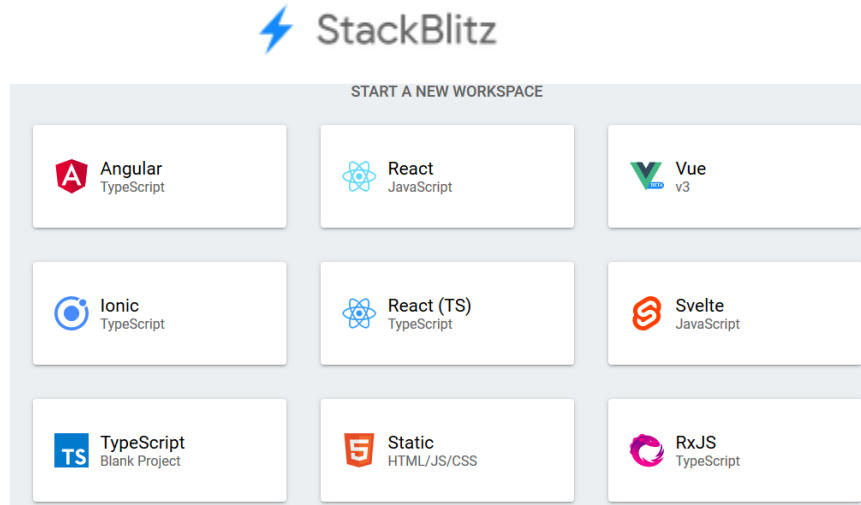
❑ You can also use online code editing : [StackBlitz](#), [CodePen](#) or [CodeSandbox](#).


❑ **Ex. StackBlitz** (<https://stackblitz.com/>) provides another way to code your application online and from your browser. It provides the ability to create, edit and deploy fullstack applications.

❑ Need to sign in to GitHub to use StackBlitz

❑ Can be used to code with a large panel of web development frameworks (Angular, React, Vue.js, etc).

❑ Sponsored by the world known organizations & individuals (Google, Firbase, Ignite UI, etc).





Sign in to **GitHub**  
to continue to **Stackblitz**

Username or email address

Afficher les identifiants enregistrés

.....

Sign in

New to GitHub? [Create an account.](#)

## 2.1 Create a React app project (1/2)

- ❑ Create React app package to build a new React application → In terminal windows write : **create-react-app react-app**
- ❑ This command will install React as well as all the third party libraries (like : Lightweight Development Server, Webpack, Babel for compiling JavaScript code, etc.).

```
c:\> create-react-app react-app

Creating a new React app in c:\react-app.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...
```

- ❑ from the folder of the React app just been created (cd react-app) run > **npm start**
- ❑ From the directory of your project write in the Terminal “**code .**” → this will open VS code pointing to your project directory.

```
c:\>cd react-app

c:\react-app>npm start

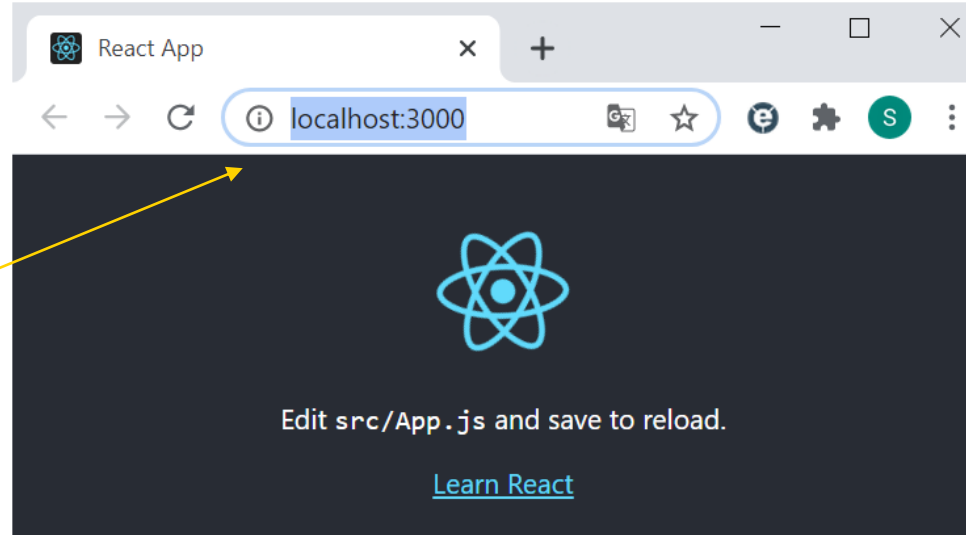
> react-app@0.1.0 start c:\react-app
> react-scripts start

i Browserslist: Project is running at http://10.171.234.128/
i Browserslist: webpack output is served from
i Browserslist: Content not from webpack is served from c:\react-app\public
i Browserslist: 404s will fallback to /
Starting the development server...
Compiled successfully!

You can now view react-app in the browser.

Local:      http://localhost:3000
On Your Network: http://10.171.234.128:3000

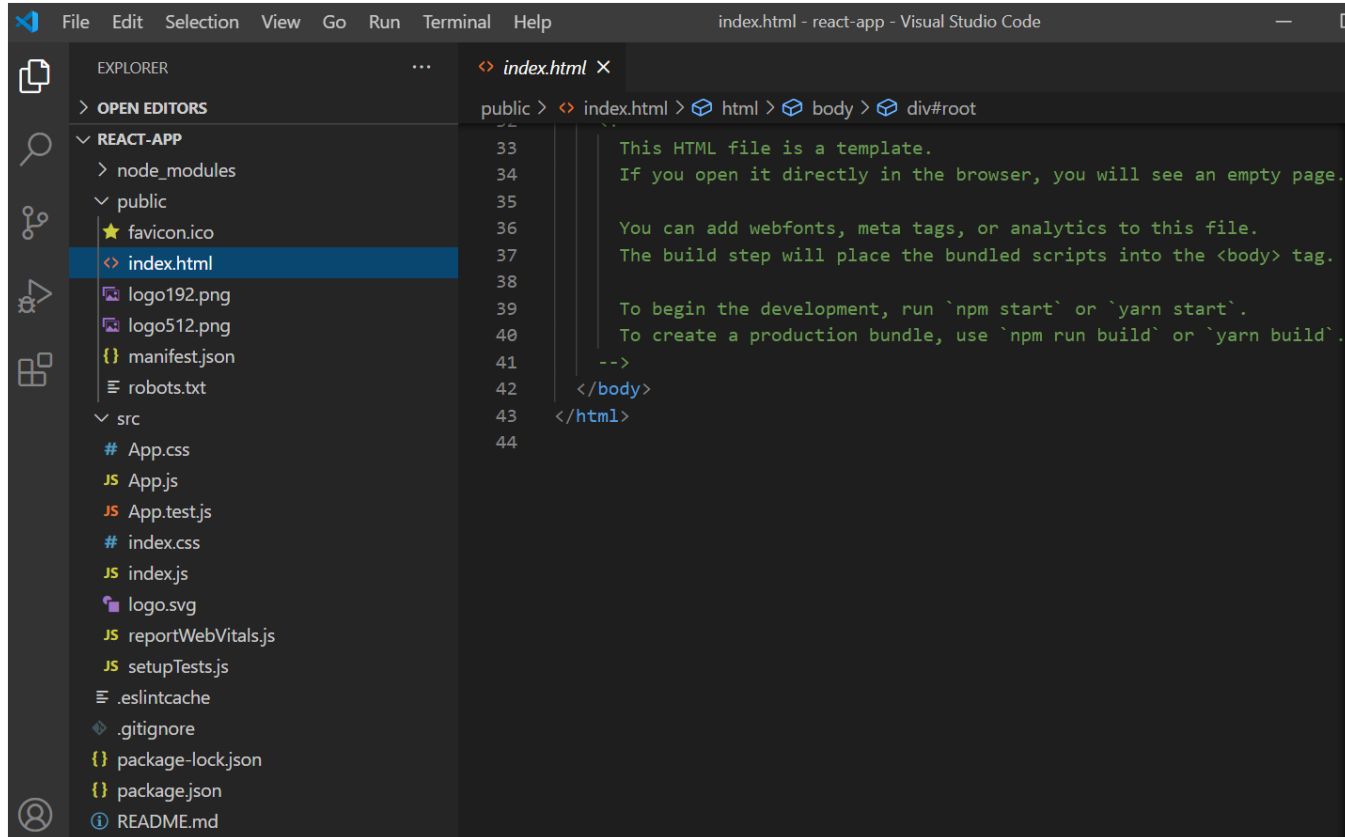
8 Note that the development build is not optimized.
To create a production build, use npm run build.
```





## 2.1 Create a React app project (2/2)

- ❑ `<div id="root"></div>` marks the container or the place where the React Application is going to operate within the index.html (in VS code react-app project folder/Public/index.html).



# #3 Main concepts

## 3.1 Hello World Example

- ❑ The smallest React example looks like the following code.
- ❑ It displays a heading saying “Hello, world!” on the page.

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

## 3.2 JavaScript XML (JSX)

- ❑ JSX stands for JavaScript and XML. It's an extension of JavaScript that brings HTML-like syntax to a JavaScript environment.
- ❑ JSX allows to write HTML elements in JavaScript and place them in the DOM without any createElement() and/or appendChild() methods.
- ❑ It's recommended to add Babel JavaScript extension (developed by Michael McDermott) to VS code, so that both ES6 and JSX code will be properly highlighted.

//The following shows a simple JSX example:

```
const subject = "World";
const header = (
  <header>
    <h1>Hello, {subject}</h1>
  </header>
);
```



//When used with React, the JSX from the previous  
//snippet would be compiled into this

```
var subject = "World";
var header = React.createElement("header", null,
  React.createElement("h1", null, "Hello, ", subject, "!")
);
```



//When ultimately rendered by the browser, the above  
// snippet will produce HTML that looks like this:

```
<header>
  <h1>Hello, World!</h1>
</header>
```

## 3.3 Rendering Elements

- ❑ An **element** describes what you want to see on the screen: example : `const element = <h1>Hello, world</h1>;`
- ❑ Unlike browser DOM elements, React elements are plain objects, and are cheap to create. React DOM takes care of updating the DOM to match the React elements.
- ❑ “root” DOM node → `<div id="root"></div>` .Everything inside this root DOM will be managed by React DOM.
- ❑ Applications built with just React **usually have a single root DOM node**.
- ❑ To **render** a React element into a root DOM node, pass both to **ReactDOM.render()**:

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

- ❑ React elements are **immutable**. Once you create an element, you can't change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.
- ❑ The only way to update the UI is to create a new element, and pass it to ReactDOM.render().
- ❑ In practice, most React apps only call ReactDOM.render() once
- ❑ **React Only Updates What's Necessary !** React DOM compares the element and its children to the previous one, **and only applies update on only the changed parties**.

```
// The function Tick calls ReactDOM.render()
// every second from a setInterval() callback.
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'));
}

setInterval(tick, 1000);
```

## 3.4 Components and Props

- ❑ **Components** let split the UI into independent, reusable pieces, and think about each piece in isolation.
- ❑ Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “**props**”) and return React elements describing what should appear on the screen.

- ❑ The simplest way to define a component is to write a JavaScript function which accepts a single “props” (i.e. properties) object argument with data and returns a React element.
- ❑ Such components are called “**Function Components**” because they are literally JavaScript functions.



```
// A valid react Component
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- ❑ A component can also be provided with an ES6 (JS2015) Class → a “**Class Component**”.
- ❑ Both Class and Function Components are equivalent from React’s point of view.



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

## 3.4.1 Components and Props – Rendering a Component

```
//Rendering "Hello, Sara" on the page

function Welcome(props) { return <h1>Hello, {props.name}</h1>;
}

const element = <Welcome name="Sara" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Let's recap what happens in this example:

1. **ReactDOM.render()** is called with the **<Welcome name="Sara" />** element.
2. React calls the **Welcome** component with **{name: 'Sara'}** as the props.
3. The **Welcome** component returns a **<h1>Hello, Sara</h1>** element as the result.
4. React DOM efficiently updates the DOM to match **<h1>Hello, Sara</h1>**.

**Tips : always start component names with a capital letter !**

- ❑ React treats components starting with lowercase letters as DOM tags. For example, **<div />** represents an HTML div tag, but **<Welcome />** represents a component and requires **Welcome** to be in scope

## 3.4.2 Components and Props – Composing Component (1/2)

```
// App component that renders Welcome many times
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />      <Welcome name="Cahal" />
      <Welcome name="Edite" />    </div>
    );
}

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

- ❑ Components can refer to other components in their output.
  - ❑ Consequently, the same component abstraction can be reused for any level of detail (e.g. button, form, dialog, screen etc.).
- 
- ❑ Typically, new React apps have a single App component at the very top.
  - ❑ However, if React is integrated into an existing app, we might start bottom-up with a small component like Button and the progressively move up to the top of the view hierarchy.



### 3.4.2 Components and Props – Composing Component (2/2)

```
function Comment(props) {  
  return (  
    <div className="Comment">  
      <div className="UserInfo">  
        <img className="Avatar"  
          src={props.author.avatarUrl}  
          alt={props.author.name}  
        />  
        <div className="UserInfo-name">  
          {props.author.name}  
        </div>  
      </div>  
      <div className="Comment-text">  
        {props.text}  
      </div>  
      <div className="Comment-date">  
        {formatDate(props.date)}  
      </div>  
    </div>  
  );  
}
```

A tricky function component “Comment”, that should be difficult to change and reuse its individual parts → it's better to divide it and extract few components from it.

```
function Avatar(props) {  
  return (  
    <img className="Avatar"  
      src={props.user.avatarUrl}  
      alt={props.user.name}  
    />  
  );  
}
```

```
function UserInfo(props) {  
  return (  
    <div className="UserInfo">  
      <Avatar user={props.user} />  
      <div className="UserInfo-name">  
        {props.user.name}  
      </div>  
    </div>  );  
}
```

```
function Comment(props) {  
  return (  
    <div className="Comment">  
      <UserInfo user={props.author} />  
      <div className="Comment-text">  
        {props.text}  
      </div>  
      <div className="Comment-date">  
        {formatDate(props.date)}  
      </div>  
    </div>  
  );  
}
```

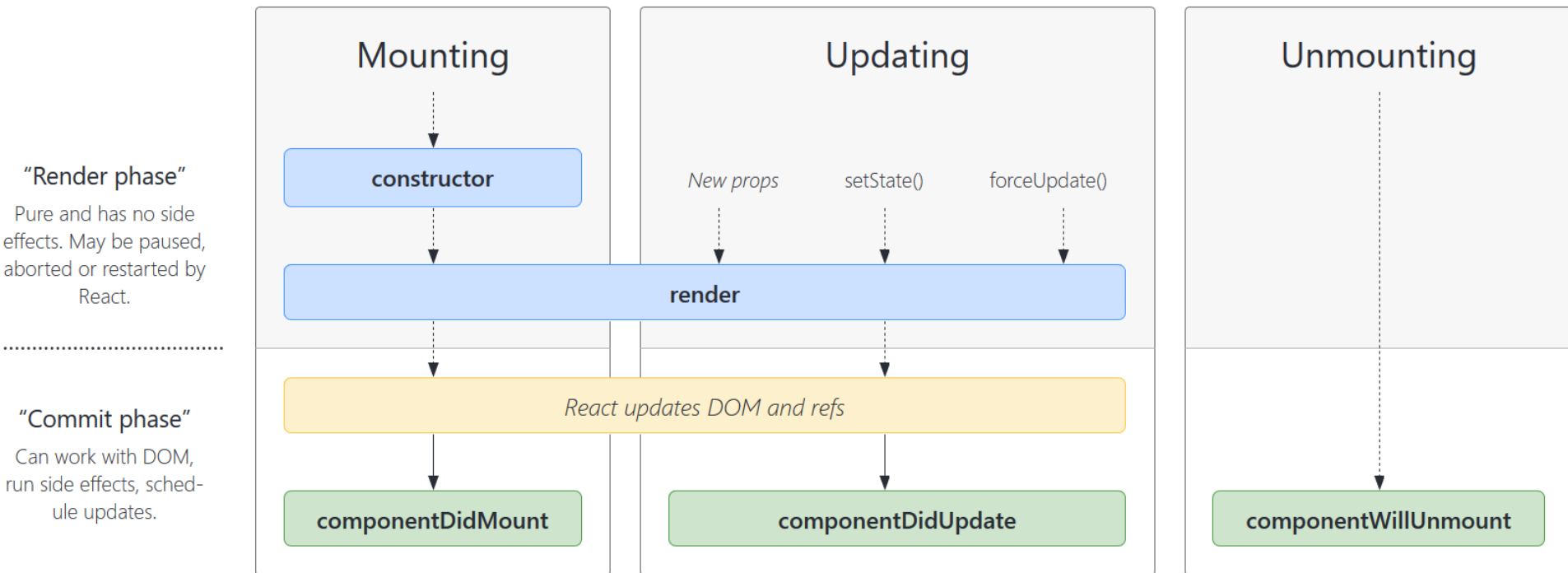
Our Comment function is now simplified. Together with the two extracted small functions provides the same job than its initial form.

Firstly, we extract Avatar as a separate small component.

Secondly, we will extract a UserInfo component that renders an Avatar next to the user's name.

## 3.5 State and Lifecycle (1/2)

React's Lifecycle diagram : <https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>



## 3.5 State and Lifecycle (2/2)

1) When `<Clock />` is passed to `ReactDOM.render()`, React calls the constructor of the Clock component. Since Clock needs to display the current time, it initializes `this.state` with an object including the current time. We will later update this state.

2) React then calls the Clock component's `render()` method. This is how React learns what should be displayed on the screen. React then updates the DOM to match the Clock's render output.

3) When the Clock output is inserted in the DOM, React calls the `componentDidMount()` lifecycle method. Inside it, the Clock component asks the browser to set up a timer to call the component's `tick()` method once a second.

4) Every second the browser calls the `tick()` method. Inside it, the Clock component schedules a UI update by calling `setState()` with an object containing the current time. Thanks to the `setState()` call, React knows the state has changed, and calls the `render()` method again to learn what should be on the screen. This time, `this.state.date` in the `render()` method will be different, and so the render output will include the updated time. React updates the DOM accordingly.

4) If the Clock component is ever removed from the DOM, React calls the `componentWillUnmount()` lifecycle method so the timer is stopped

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }
  componentWillUnmount() {clearInterval(this.timerID);}
  tick() { this.setState({ date: new Date() }); }
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

## 3.6 Handling Events (1/2)

Handling events with React elements is very similar to handling events on DOM elements. There still some syntax differences:

1. React events are named using camelCase, rather than lowercase.
2. With JSX you pass a function as the event handler, rather than a string.
3. In React do not return **false** to prevent default behavior. In place must explicitly call **preventDefault**.

❑ “e” is a **synthetic event**.

❑ synthetic events in react follow W3C specifications, so you don't need to worry about cross-browser compatibility.

❑ React events do not work exactly the same as native events.

4. When using React, we generally don't need to call **addEventListener** to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

```
//in HTML
<button onClick="activateLasers()">
  Activate Lasers
</button>
```

```
//in React
<button onClick={activateLasers}> Activate Lasers
</button>
```

```
//example, with plain HTML, to prevent the
//default link behavior of opening a new :
<a href="#" onClick="console.log('The link was clicked.');" return false">
  Click me
</a>
```

```
//In react, this could be wrote as follow:
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');"
  }
  return (
    <a href="#" onClick={handleClick}>
      Click me
    </a>
  );
}
```

## 3.6 Handling Events (2/2)

5. An event handler can be also a method of a ES6 class component like as given in this example with the function *handleClickO*.

**Note :** in this example, just be sure to bind *this.handleClick* and pass it to *onClick*, because in JavaScript ES6 generally, if we refer to a method without *()* after it, such as `onClick={this.handleClick}`, we should bind that method.

```
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(state => ({      isToggleOn: !state.isToggleOn
  });
  }

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

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

### 3.6.1 Handling Events: Passing Arguments to Event Handlers

Inside a loop, it is common to want to pass an extra parameter to an event handler. For example, if `id` is the row ID, either of the following two examples would work :



```
//Method 1: passing a parameter to event handler by using arrow function
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

//Method 2;: passing a parameter to event handler by using Function.prototype.bind"
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

- ❑ In both cases, the “**e**” argument representing the React event will be passed as a second argument after the ID.
- ❑ With an arrow function the “**e**” argument is passed explicitly, but with `bind` any further arguments are automatically forwarded.

## 3.7 Conditional Rendering

We can create distinct components that encapsulate behavior we need and then render only some of them, depending on the state of our application.

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like **if** or **the conditional operator** to create elements representing the current state, and let React update the UI to match them.

### Variable Elements:

Use variables for storing elements to conditionally render a part of a component while the rest of the output doesn't change.

→ For that use Stateful Component to render conditional outputs depending on its current state.

```
function UserGreeting(props) {  
  return <h1>Welcome back!</h1>;  
}  
  
function GuestGreeting(props) {  
  return <h1>Please sign up.</h1>;  
}  
  
if (isLoggedIn) { return <UserGreeting />; } return <GuestGreeting />;
```

See an example of this on CodePen

<https://codepen.io/gaearon/pen/QKzAgB?editors=0010>

While declaring a variable and using an if statement is a fine way to conditionally render a component, sometimes you might want to use a shorter syntax. Need ways to inline conditions in JSX :

- Inline If with Logical && Operator
- Inline If-Else with Conditional Operator ***condition ? true : false***

## 3.8 Lists and Keys (1/2)

- Loop through the numbers array using the JavaScript **map()** function.
- Return a **<li> element** for each item and assign the resulting array of elements to listItems.
- include the entire listItems array inside a **<ul> element**, and render it to the DOM

### Rendering lists inside a component

Include the special “key” string attribute when creating lists of elements → assign a **key** to our list items inside **numbers.map()**

### Keys

- Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity.
- The best way to pick a key is to use a string that uniquely identifies a list item among its siblings. Most often you would use IDs from your data as keys:

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}> {todo.text}
</li>
);
```

24

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
ReactDOM.render(
  <ul>{listItems}</ul>,
  document.getElementById('root')
);
```

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```



## 3.8 Lists and Keys (2/2)

### Extracting Components with Keys:

- Keys only make sense in the context of the surrounding array.
- For example, if we extract a *ListItem component*, we should keep the key on the `<ListItem />` **elements in the array** rather than *on the `<li>` element in the ListItem itself*.

### Rules to remember:

- elements inside the *map()* call need keys
- Keys used within arrays should be unique among their siblings. However they don't need to be globally unique. We can use the same keys when we produce two different arrays

## 3.9 Forms (1/2)

→ Since the value attribute is set on our form element, the displayed value will always be **this.state.value**, making the React state the source of truth.

→ Since **handleChange** runs on every keystroke to update the React state, the displayed value will update as the user types

### Forms as Controlled Components:

With a controlled component, the input's value (form) is always driven by the React state. While this means you have to type a bit more code, you can now pass the value to other UI elements too, or reset it from other event handlers.

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {    this.setState({value: event.target.value});  }
  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

## 3.9 Forms (2/2)

### Textarea Tag:

In React, a `<textarea>` uses a `value` attribute. By this way, a form using a `<textarea>` can be written very similarly to a form that uses a single-line input.

### Select Tag:

React, instead of using ***selected attribute***, uses a ***value attribute*** on the ***root select tag*** which is more convenient in a controlled component as we only need to update it in one place.

It's possible to pass an array into the value attribute. This allows to select multiple options in a select tag: `<select multiple={true} value={['B', 'C']>`

### Handling Multiple Inputs

When handling multiple controlled input elements is needed, we can add a `name` attribute to each element and let the handler function choose what to do based on the value of ***event.target.name***.

### Fully-Fledged Solutions :

Formik is one of the popular complete solution for form that allows include validation, keep track of the visited fields, and handle form submission. It's built on the same principles of controlled components and managing state.

<https://formik.org/>

## 3.10 Lifting State Up - Composition vs Inheritance

### Lifting State Up:

Often, several components need to reflect the same changing data. It's recommended to lift the shared state up to their closest common ancestor.

### Composition vs Inheritance:

React has a powerful composition model, and it's recommended to use composition instead of inheritance to reuse code between components.

# #4

## Exercise

## Make your React app

# References

- ❑ React official web page: <https://fr.reactjs.org/>



# Thanks !

# Questions ?

Samir Bellahsene

[samir.bellahsene@orange.com](mailto:samir.bellahsene@orange.com)