

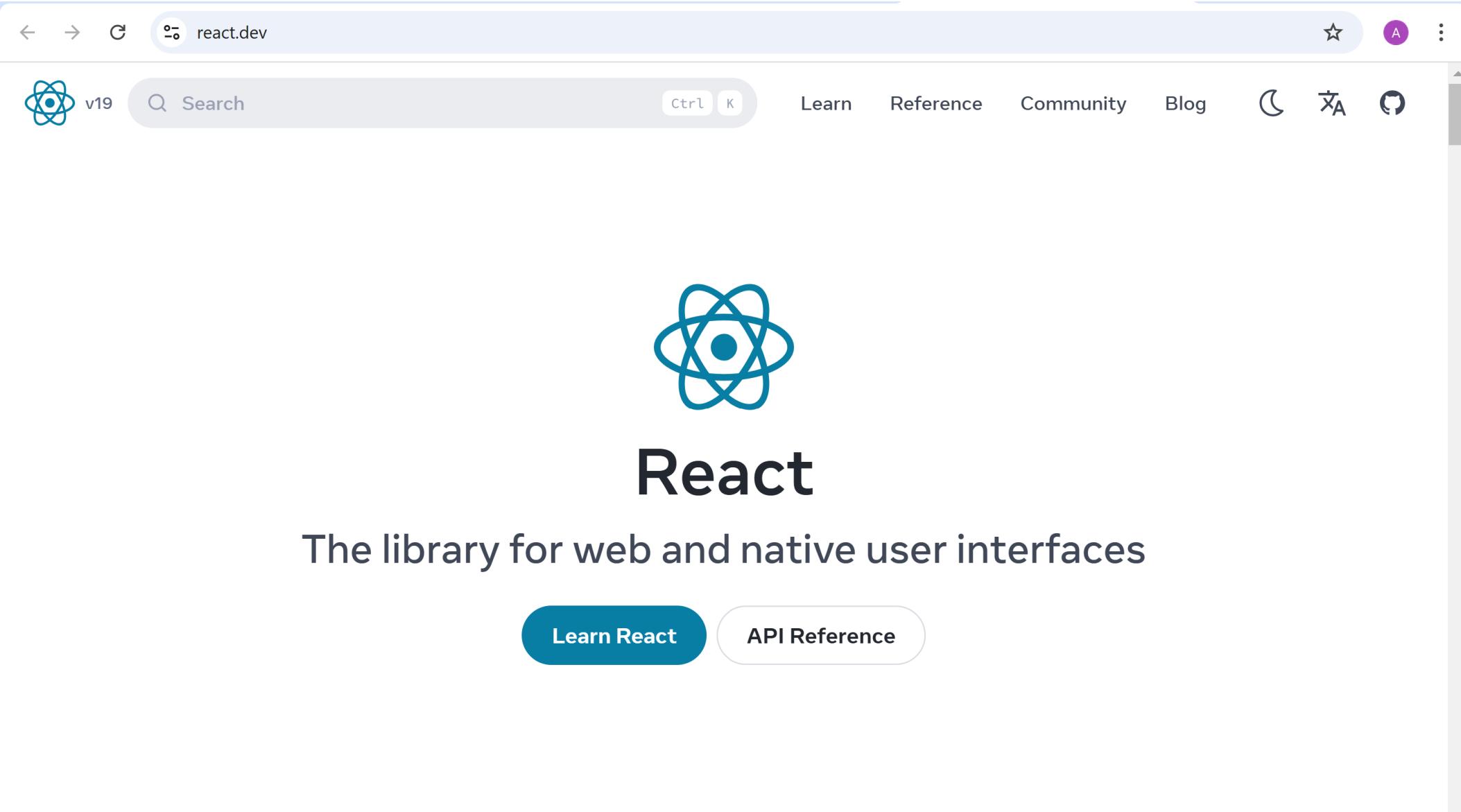
# Introduction to React, for Angular Developpers

Course Esilv 2024  
arnaud.nauwynck@gmail.com

this document:

<https://github.com/Arnaud-Nauwynck/Presentation/web/>

# https://react.dev



The screenshot shows the React.dev homepage. At the top, there's a browser header with back/forward buttons, a search bar containing "react.dev", and a star icon. Below the header is a navigation bar with links for "Learn", "Reference", "Community", and "Blog". There are also icons for a moon (dark mode), a keyboard (ctrl+k), and a refresh symbol. On the left side of the main content area, there's a small React logo icon and the text "v19". A search bar with the placeholder "Search" is also present. The main content features the large React logo, followed by the word "React" in a large serif font, and the subtitle "The library for web and native user interfaces" in a smaller sans-serif font. At the bottom, there are two buttons: a teal rounded rectangle labeled "Learn React" and a white rounded rectangle labeled "API Reference".

react.dev

v19

Search

Ctrl K

Learn Reference Community Blog

React

The library for web and native user interfaces

Learn React API Reference

# https://react.dev/learn

The screenshot shows the 'Learn' section of the React.dev website. At the top, there's a navigation bar with icons for back, forward, search, and user profile. Below it is a header with the React logo, version v19, a search bar, and links for Learn, Reference, Community, Blog, and GitHub.

**GET STARTED**

**Quick Start**

- Tutorial: Tic-Tac-Toe
- Thinking in React

**Installation**

**LEARN REACT**

**Describing the UI**

**Adding Interactivity**

**Managing State**

**Escape Hatches**

React components are JavaScript functions that return markup:

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

Now that you've declared `MyButton`, you can nest it into another component:

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

Is this page useful?

Notice that `<MyButton />` starts with a capital letter. That's how you know it's a React

# JSX, TSX ?

ChatGPT ▾

explain tsx langage



TSX (TypeScript JSX) is a syntax extension for TypeScript that allows you to write JSX code with TypeScript's type-checking and additional features. It is commonly used in React projects where developers benefit from TypeScript's static typing and JSX's syntax for creating user interfaces.

## What is JSX?

JSX (JavaScript XML) is a syntax extension for JavaScript used with React. It allows you to write HTML-like code within JavaScript, making it easier to describe what the UI should look like. For example:

jsx

Copy code

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

## What makes TSX different?

TSX is JSX combined with TypeScript, so you can:

1. Use TypeScript's static typing to enforce and check types.
2. Leverage TypeScript's interfaces, enums, and types for better type safety in your React components and props.
3. Write cleaner, safer, and more maintainable React code.

# TSX -> TS + React runtime Virtual DOM

how to transform jsx to js ?



Transforming JSX to plain JavaScript involves converting the JSX syntax into equivalent `React.createElement` function calls, which are how JSX is interpreted by React under the hood. Here's how you can do it:

## 1. Understand the Equivalent JavaScript Code

JSX is syntactic sugar for `React.createElement`. For example:

JSX:

jsx

Copy code

```
const element = <div className="container">Hello, world!</div>;
```

JavaScript:

js

Copy code

```
const element = React.createElement(
  "div",
  { className: "container" },
  "Hello, world!"
);
```

<https://babeljs.io/docs/babel-plugin-transform-react-jsx>

BABEL

Docs

- Guides >
- Config Reference >
- Presets >▼
  - @babel/preset-env >
  - @babel/preset-react >▼
    - transform-react-jsx
  - transform-react-jsx-development
- @babel/preset-typescript >
- @babel/preset-flow >
- Misc >
- Integration Packages >
- Tooling Packages >
- Helper Packages >

JavaScript

```
const profile = (
  <div>
    
    <h3>{[user.firstName, user.lastName].join(" ")}</h3>
  </div>
);
```

Out

JavaScript

```
import { jsx as _jsx } from "react/jsx-runtime";
import { jsxs as _jsxs } from "react/jsx-runtime";

const profile = _jsxs("div", {
  children: [
    _jsx("img", {
      src: "avatar.png",
      className: "profile",
    }),
    _jsx("h3", {
      children: [user.firstName, user.lastName].join(" "),
    }),
  ],
});
```

# 2 new languages mix of JS + HTML

TSX = extension of TS (TypeScript) containing HTML

JSX = extension to JS (JavaScript) containing HTML

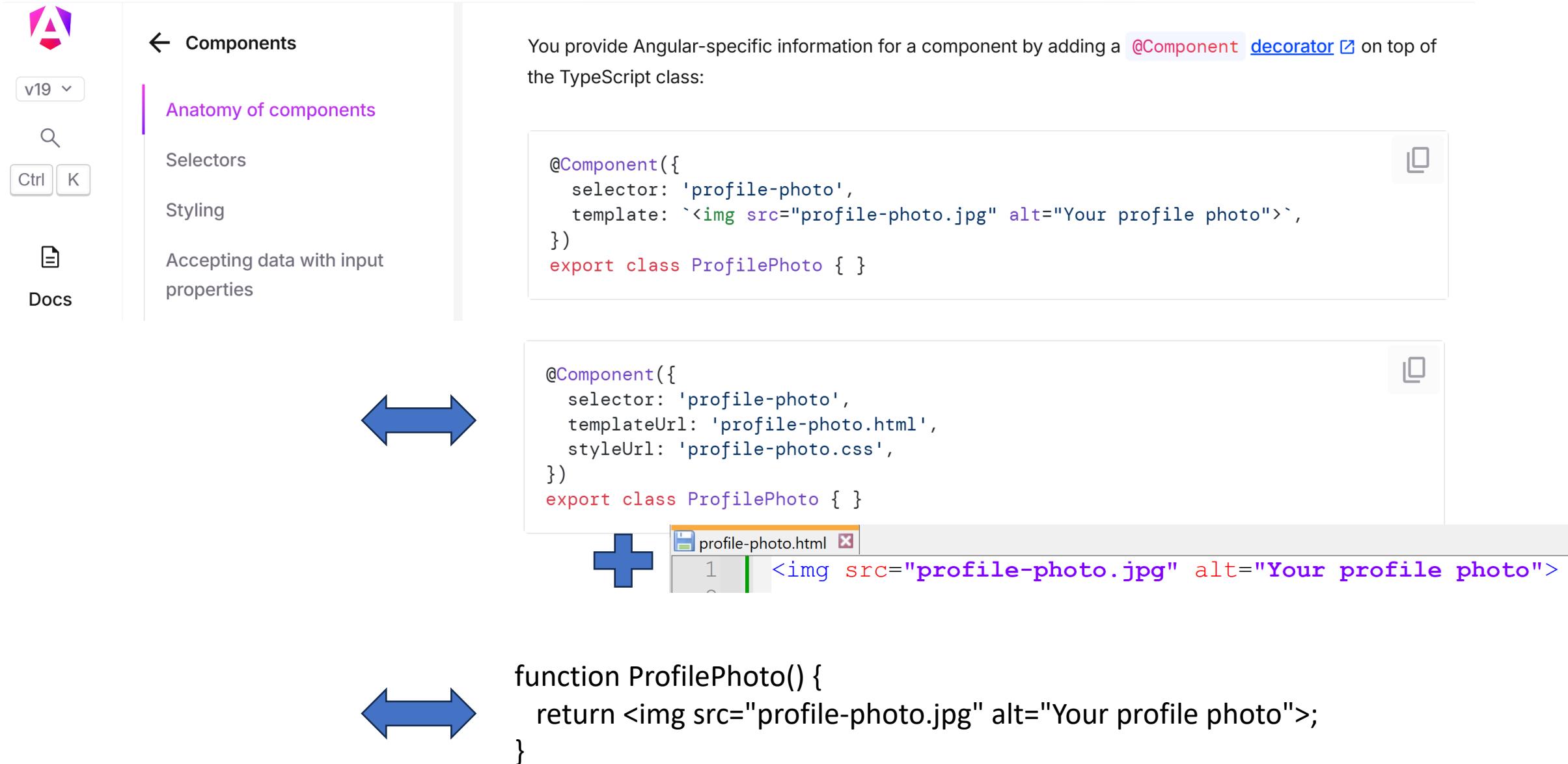
JSX is javascript that contains HTML <> ... </>

and nested HTML can contains <>{ javascript}<> !!

...which can contains sub-nested HTML {<> .. </>}

... which can contain Js: <>{ js.. <> { js } </> ...} </>

# 1 TSX File vs Angular 1 ts or 1 ts+1 html Files



The screenshot shows the Angular documentation for Components. On the left, there's a sidebar with a search icon, a dropdown for version (v19), and keyboard shortcut keys (Ctrl + K). Below the sidebar are sections: Anatomy of components, Selectors, Styling, and Accepting data with input properties. A double-headed arrow points from the 'Anatomy of components' section to the first code sample.

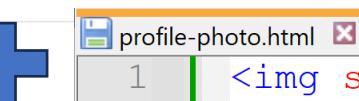
You provide Angular-specific information for a component by adding a `@Component` [decorator](#) on top of the TypeScript class:

```
@Component({
  selector: 'profile-photo',
  template: ``,
})
export class ProfilePhoto {}
```

A second double-headed arrow points from the 'Accepting data with input properties' section to the second code sample. This sample uses templateUrl and styleUrls instead of template and styleUrl.

```
@Component({
  selector: 'profile-photo',
  templateUrl: 'profile-photo.html',
  styleUrls: 'profile-photo.css',
})
export class ProfilePhoto {}
```

Below this, there's a preview of a code editor showing the contents of profile-photo.html:



```

```

At the bottom, another double-headed arrow points from the 'Accepting data with input properties' section to the third code sample, which shows a function-based component implementation:

```
function ProfilePhoto() {
  return ;
}
```

# Pros/Cons of 2 Files for Angular ?

Angular Html Templates are REAL Html syntaxically correct

(the \*ngIf or @if() { ..} are just attribute / text in plain old html)

++ having 2 files with extension is better supported by all IDE

- It is overkill for component with few line(s)
- templateHtml='<div>..</div>' is not supported by all IDE as Html
- TSX / TSX : neither valid Typescript, nor valid HTML.  
merge of both langages, require special IDE support

# Create a New React Project

try 1 : tested official doc React + Next.JS Framework

**npx create-next-app**

next dev ... FAILED

try 2 : tested

**npx create-react-app**

npm install ... FAILED

try 3 : tested **IDEA > New Project**

Run ... FAILED

new project using React + Next.Js

# New React Project + choose a Framework

The screenshot shows a web browser displaying the [react.dev/learn/start-a-new-react-project](https://react.dev/learn/start-a-new-react-project) page. The page is part of the 'Learn' section, specifically under 'INSTALLATION > Start a New React Project'. The left sidebar lists several 'GET STARTED' topics, with 'Start a New React Project' highlighted. The main content area contains text about starting a new React project using frameworks like Create React App or Next.js.

react.dev/learn/start-a-new-react-project

GET STARTED

Quick Start >

Installation ▾

Start a New React Project

Add React to an Existing Project

Editor Setup

Using TypeScript

React Developer Tools

React Compiler

LEARN REACT > INSTALLATION >

## Start a New React Project

If you want to build a new app or a new website fully with React, we recommend picking one of the React-powered frameworks popular in the community.

You can use React without a framework, however we've found that most apps and sites eventually build solutions to common problems such as code-splitting, routing, data fetching, and generating HTML. These problems are common to all UI libraries, not just React.

By starting with a framework, you can get started with React quickly, and avoid essentially building your own framework later.



React is NOT enough,  
You need additional Framework(s)

example: for Router,  
for state : Redux,  
better bindings : MOBX,..

# Next.js

react.dev/react/learn/start-a-new-react-project

GET STARTED

- Quick Start >
- Installation <▼
- Start a New React Project**
- Add React to an Existing Project
- Editor Setup
- Using TypeScript
- React Developer Tools
- React Compiler

LEARN REACT

Interested in being included on this list, [please let us know](#).

## Next.js

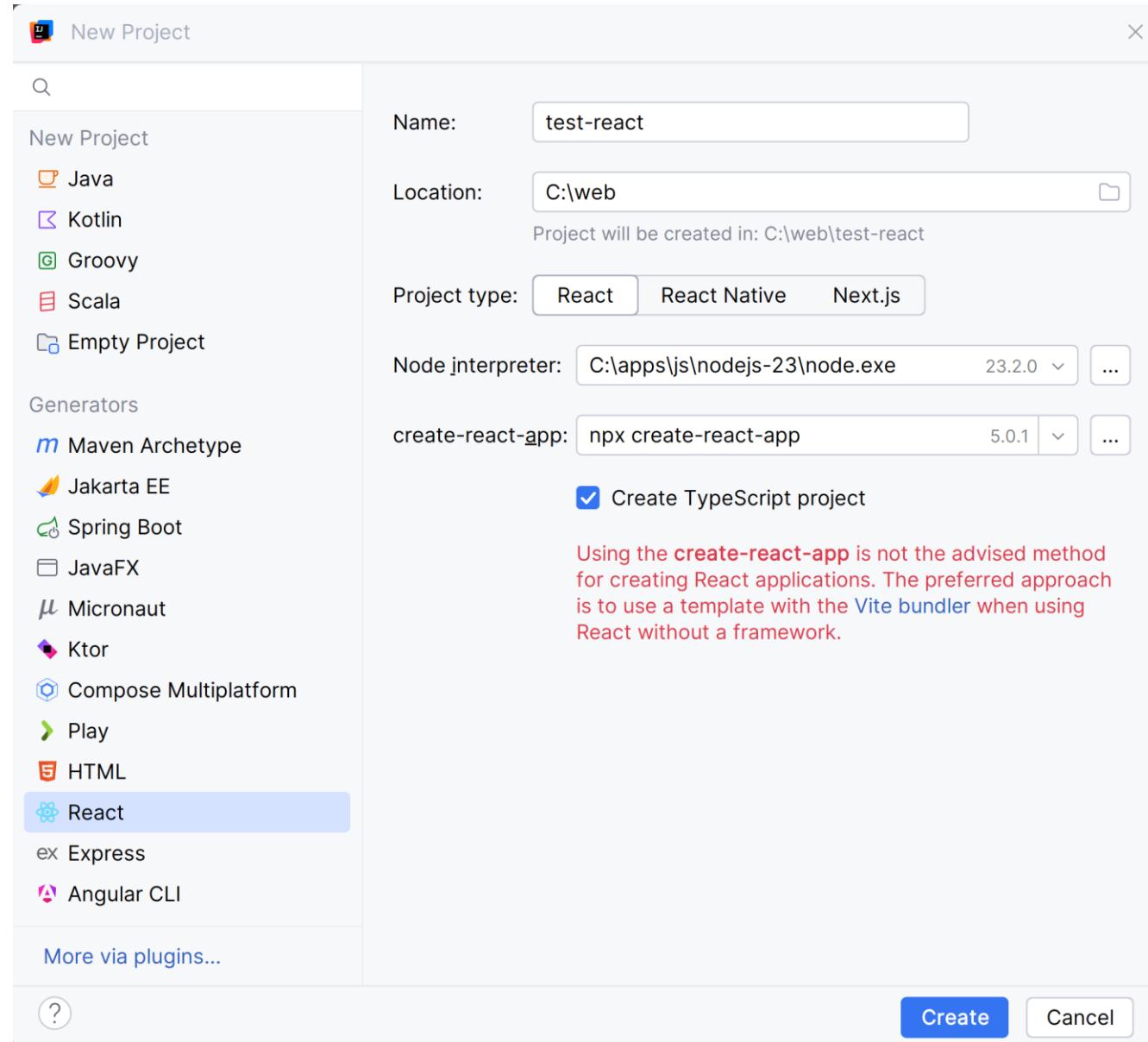
**Next.js' Pages Router** is a full-stack React framework. It's versatile and lets you create React apps of any size—from a mostly static blog to a complex dynamic application. To create a new Next.js project, run in your terminal:

```
Terminal npx create-next-app@latest
```

If you're new to Next.js, check out the [learn Next.js course](#).

Next.js is maintained by [Vercel](#). You can [deploy a Next.js app](#) to any Node.js or serverless hosting, or to your own server. Next.js also supports a [static export](#) which doesn't require a server.

# From Idea: New Project > "React"



# Exploring a Minimal React App

# React Project

The screenshot shows a development environment with the following details:

- Project Structure:** The left sidebar displays the project structure under "test-react". Key files shown include `App.css`, `App.test.tsx`, `App.tsx`, `index.css`, `index.tsx`, `logo.svg`, `reportWebVitals.ts`, `setupTests.ts`, `.gitignore`, `package.json` (selected), `package-lock.json` (highlighted), and `README.md`.
- Code Editor:** The main editor pane shows the `package.json` file content. The file defines the project's name, version, private status, dependencies (including `cra-template-typescript`, `react`, `react-dom`, and `react-scripts`), scripts (with commands for start, build, test, and eject), and an eslintConfig section.
- Terminal:** The bottom pane shows the terminal output of an npm install command. It includes experimental warnings about CommonJS modules loading ES modules and support for require(). The command completed successfully, adding 1315 packages in 1m.
- Toolbar:** The top right features a toolbar with icons for npm start, search, settings, and other project management functions.

# Index.html : Main Container for your App

The screenshot shows a code editor interface with the following details:

- Project Explorer:** On the left, the project structure is shown under the "test-react" workspace. The "node\_modules" folder is highlighted in yellow, and the "index.html" file is selected.
- Code Editor:** The main area displays the content of the "index.html" file. The file starts with an HTML declaration and a head section, followed by a title, body, and noscript tags. A specific line containing a div with id="root" is highlighted with a green box.
- Toolbars and Status:** The top bar includes icons for file operations, a search bar, and status indicators like "npm start". The bottom right corner shows a bell icon with a red dot.

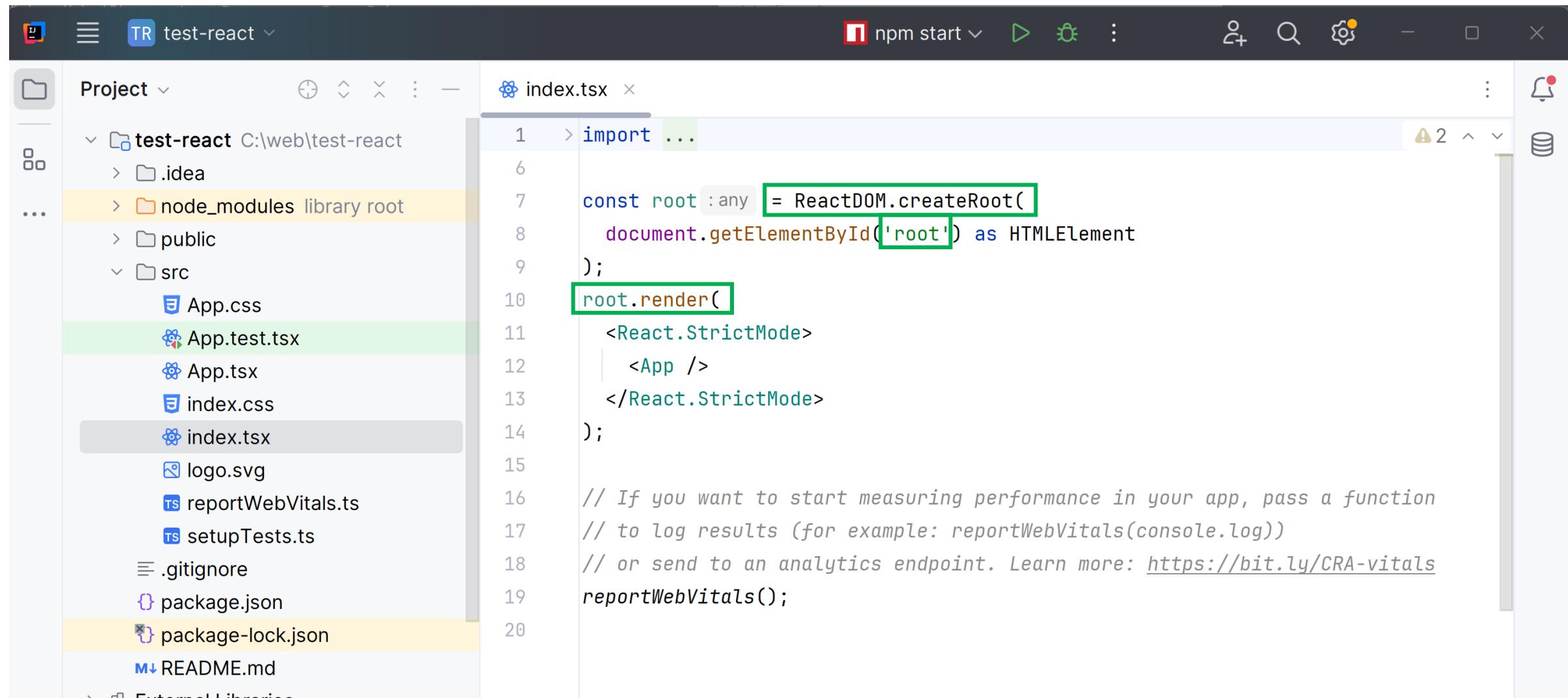
```
<html lang="en">
<head>
<title>React App</title>
</head>
<body>
<noscript>You need to enable JavaScript to run this app.</noscript>
<div id="root"></div>
<!--
This HTML file is a template.
If you open it directly in the browser, you will see an empty page.

You can add webfonts, meta tags, or analytics to this file.
The build step will place the bundled scripts into the <body> tag.

To begin the development, run `npm start` or `yarn start`.
To create a production bundle, use `npm run build` or `yarn build`.

-->
</body>
</html>
```

# index.tsx : Bootstrapping ReactDOM.render()



The screenshot shows the VS Code interface with the following details:

- Project Explorer:** Shows the project structure under "test-react". Files listed include .idea, node\_modules (library root), public, src (containing App.css, App.test.tsx, App.tsx, index.css, index.tsx, logo.svg, reportWebVitals.ts, setupTests.ts), .gitignore, package.json, package-lock.json, and README.md.
- Editor:** The file "index.tsx" is open. The code is as follows:

```
1 > import ...
6
7 const root : any = ReactDOM.createRoot(
8   document.getElementById('root') as HTMLElement
9 );
10 root.render(
11   <React.StrictMode>
12     <App />
13   </React.StrictMode>
14 );
15
16 // If you want to start measuring performance in your app, pass a function
17 // to log results (for example: reportWebVitals(console.log))
18 // or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
19 reportWebVitals();
20
```

- Status Bar:** Shows "npm start" and other standard VS Code icons.

# App.tsx

The screenshot shows the IntelliJ IDEA interface with the following details:

- Project Bar:** Shows the project name "test-react" and the file "App.tsx" is currently open.
- Toolbars:** Includes standard icons for file operations, search, and navigation.
- Project Explorer:** Displays the project structure:
  - test-react**: Contains .idea, node\_modules (library root), public (with favicon.ico, index.html, logo192.png, logo512.png, manifest.json, robots.txt), src (with App.css, App.test.tsx, App.tsx, index.css, index.tsx, logo.svg, reportWebVitals.ts, setupTests.ts), .gitignore, package.json, package-lock.json (highlighted in yellow), README.md, External Libraries, and Scratches and Consoles.
- Code Editor:** The "App.tsx" file is open, showing the following code:

```
import React from 'react';
import logo from './logo.svg';
import './App.css';

function App(): any {
  return (
    <div className="App">
      <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <p>
          Edit <code>src/App.tsx</code> and save to reload.
        </p>
        <a
          className="App-link"
          href="https://reactjs.org"
          target="_blank"
          rel="noopener noreferrer"
        >
          Learn React
        </a>
      </header>
    </div>
  );
}

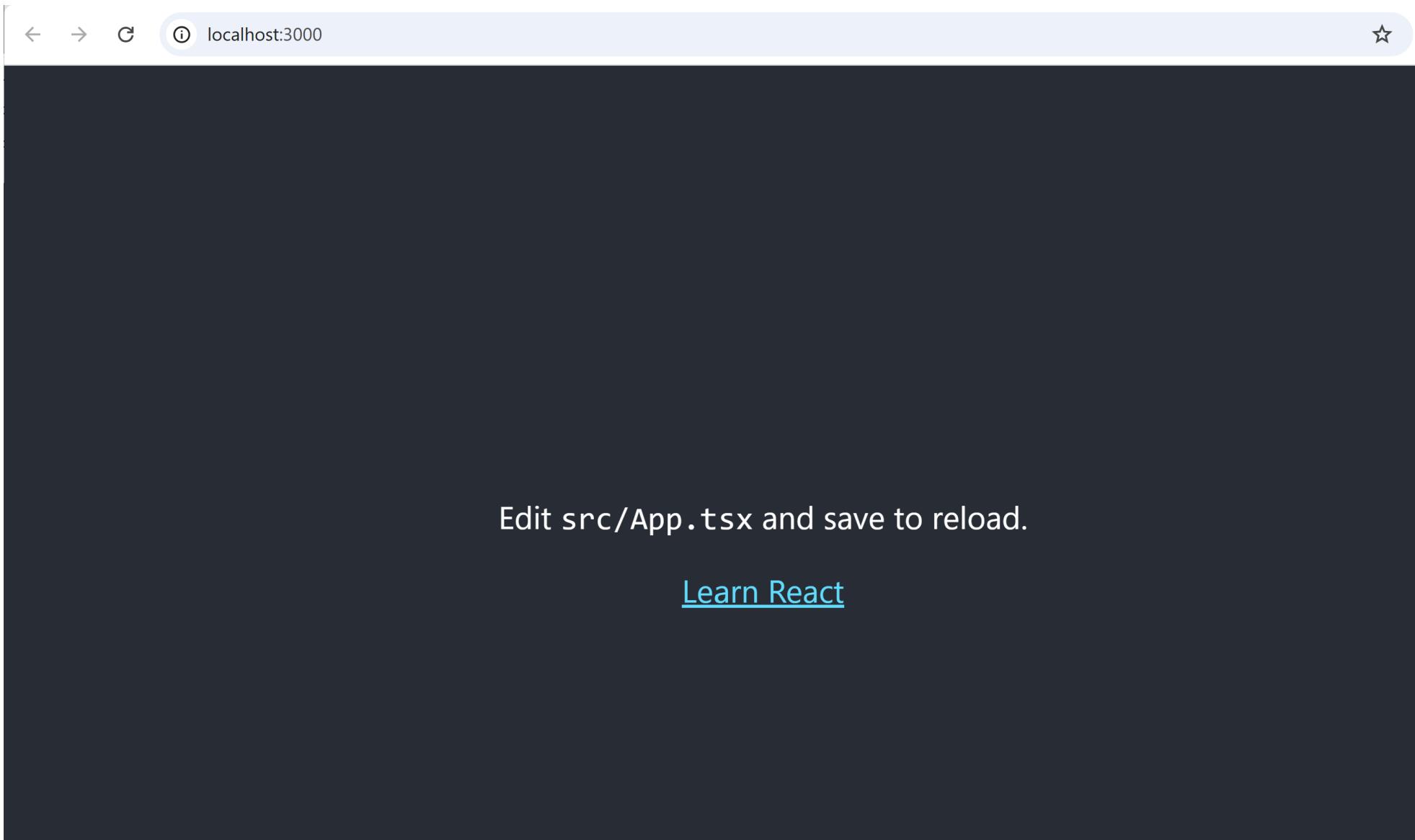
export default App;
```
- Status Bar:** Shows 1 warning and 1 error in the status bar.

# Idea Run / npm run start

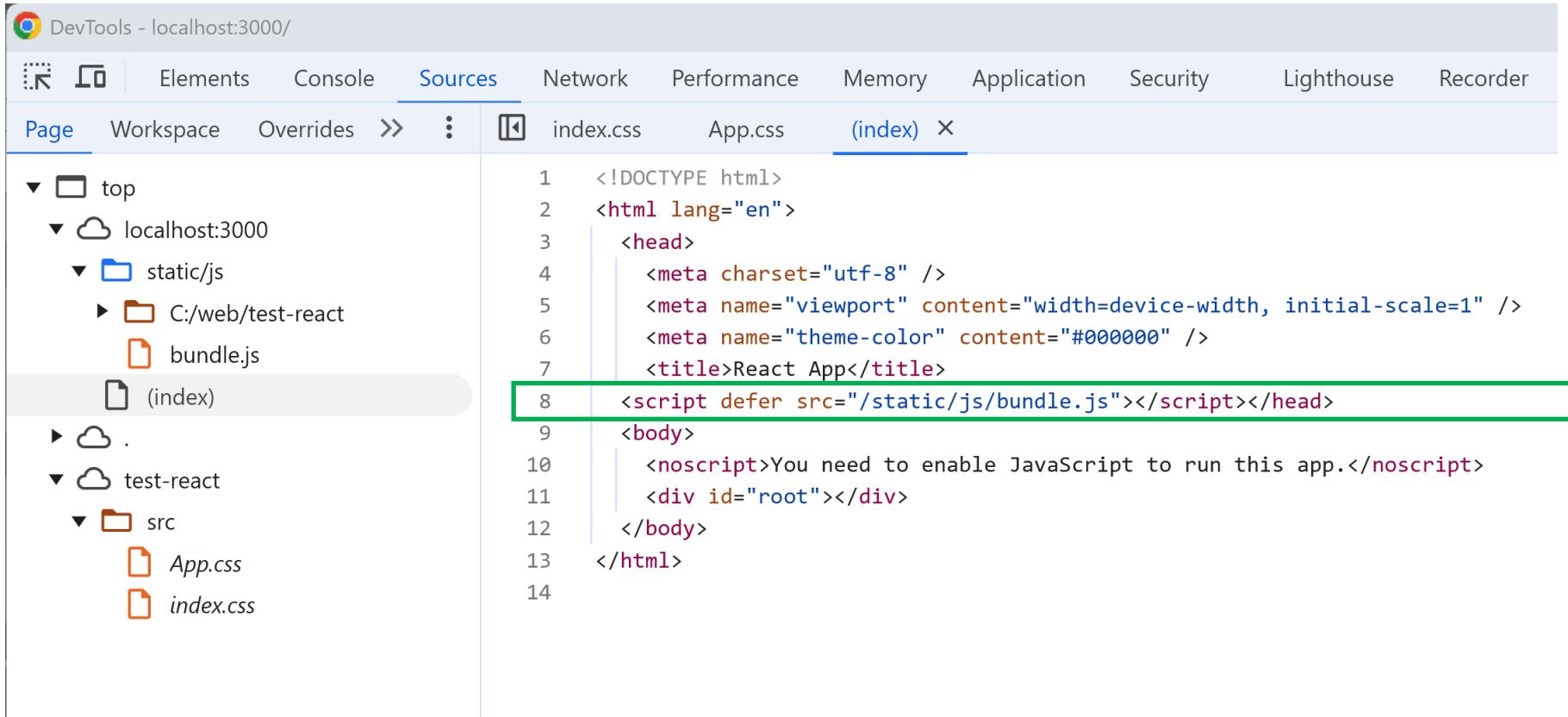
The screenshot shows the IntelliJ IDEA interface with the following details:

- Top Bar:** Shows the project name "test-react" and the "npm start" run configuration, which is highlighted with a green box.
- Project View:** Displays the project structure with files like index.tsx, App.tsx, and index.css open in the editor.
- Editor:** Shows the code for the App component, which includes a comment instructing the user to edit the file and save to reload.
- Debug Tab:** Shows the "npm start" configuration and its output.
- Output Terminal:** Displays the following text:
  - Compiled successfully!
  - You can now view **test-react** in the browser.
  - Local:** <http://localhost:3000>
  - On Your Network:** <http://172.24.224.1:3000>
  - Note that the development build is not optimized.
  - To create a production build, use `npm run build`.
  - webpack compiled **successfully**
  - No issues found.
- Bottom Status Bar:** Shows the file path "test-react > package-lock.json" and the status "6:1 TS ~ ⚡ LF UTF-8 2 spaces\*".

# http://localhost:3000



# DevTools ... viewing generated "/js/bundle"



The screenshot shows the Google DevTools interface with the "Sources" tab selected. The left sidebar displays a file tree for the project structure:

- top
- localhost:3000
  - static/js
    - C:/web/test-react
    - bundle.js
  - (index)
- .
- test-react
  - src
    - App.css
    - index.css

The right pane shows the generated HTML code for the index page. A green box highlights the `<script defer src="/static/js/bundle.js"></script>` line at line 8.

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4      <meta charset="utf-8" />
5      <meta name="viewport" content="width=device-width, initial-scale=1" />
6      <meta name="theme-color" content="#000000" />
7      <title>React App</title>
8      <script defer src="/static/js/bundle.js"></script>
9  <body>
10     <noscript>You need to enable JavaScript to run this app.</noscript>
11     <div id="root"></div>
12  </body>
13 </html>
14
```

bundle ... Transpiled TSX -> to JS  
+ react runtime on Virtual DOM

# Exploring React Features comparison with Angular

# Basic Features

Html Rendering template in Component

Calling a child component from a Component

templating with "If" and "For"

binding properties to a Component

binding Callback

Bi-Directional Binding ?

Routing ?

More ?

# Basic Features

## Comparison with Angular

Html Rendering template in Component	→ {{ expression }}	expression : field in class component
Calling a child component from a Component	→ <app-child />	selector="app-child" on child class
templating with "If" and "For"	→ @if(..) { } @else { .. } @for() { .. }	cf also legacy syntaxes *ngIf="cond" *ngFor="let i of items"
binding properties to a Component	→ field = input(); <app-child [field]="expr">	cf also legacy syntax @Input()
binding Callback	→ onEvent = signal(); <app-child (onEvent)="cb()">	cf also legacy syntax @Output()
Bi-Directional Binding ?	→ <app-child [(field)] ="parentField">	...magic & simple
Routing ?	→ <a routerLink="/compA"> or router.navigate(['compA'])	
More ?	→ form, i18n, animation, etc etc	

# Html Rendering template in Component



v19

Search

Ctrl K

Learn

Reference

## GET STARTED

Quick Start >

Installation >

## LEARN REACT

Describing the UI <

Your First Component

Importing and Exporting Components

Writing Markup with JSX

JavaScript in JSX with Curly Braces

Passing Props to a Component

Conditional Rendering

Rendering Lists

LEARN REACT > DESCRIBING THE UI >

## JavaScript in JSX with Curly Braces

JSX lets you write HTML-like markup inside a JavaScript file, keeping rendering logic and content in the same place. Sometimes you will want to add a little JavaScript logic or reference a dynamic property inside that markup. In this situation, you can use curly braces in your JSX to open a window to JavaScript.

### You will learn

- How to pass strings with quotes
- How to reference a JavaScript variable inside JSX with curly braces
- How to call a JavaScript function inside JSX with curly braces
- How to use a JavaScript object inside JSX with curly braces

Passing strings with quotes

```
return (<App attribute={JS}> </App>);
```

But what if you want to dynamically specify the `src` or `alt` text? You could **use a value from JavaScript by replacing " and " with { and }**:

App.js

Download Reset Fork

```
1 export default function Avatar() {
2   const avatar = 'https://i.imgur.com/7vQD0fPs.jpg';
3   const description = 'Gregorio Y. Zara';
4   return (
5     <img
6       className="avatar"
7       src={avatar}
8       alt={description}
9     />
10    );
11  }
12
```

```
return (<App> text {JS} </App>);
```

## Using curly braces: A window into the JavaScript world

JSX is a special way of writing JavaScript. That means it's possible to use JavaScript inside it—with curly braces `{ }`. The example below first declares a name for the scientist, `name`, then embeds it with curly braces inside the `<h1>`:

App.js

Download Reset Fork

```
1  export default function TodoList() {
2    const name = 'Gregorio Y. Zara';
3    return (
4      <h1>{name}'s To Do List</h1>
5    );
6  }
7
```

# Exploring React Features

## Comparison with Angular

<input checked="" type="checkbox"/> Html Rendering template in Component	return (<> text { expr } </>)	→ {{ expr }}
Calling a child component from a Component		→ <app-child />
templating with "If" and "For"		→ @if(..) { } @else { .. } @for() { .. }
binding properties to a Component		→ field = input(); <app-child [field]="expr">
binding Callback		→ onEvent = signal(); <app-child (onEvent)="cb()">
Bi-Directional Binding ?		→ <app-child [(field)] ="parentField">
Routing ?		→ <a routerLink="/compA"> or router.navigate(['compA'])
More ?		→ form, i18n, animation, etc etc

# Class are Statefull != pure Function are stateless



```
@Component( ..)
export class MyComp {

    stateField: string = 'Hey?';

}

<app-MyComp>
    {{ stateField }}
</app-MyComp>
```



```
function MyComp() {
    return (
        <>
            { stateField }
        </>
    );
}
```

??????????  
**not a parameter**  
**not a global variable**  
**can not be in pure fonction**

Class are Statefull  
!= pure Function are stateless  
BUT React use "Hook" useState() !!



```
@Component( ..)
export class MyComp {

    stateField: string = 'Hey?';

}

<app-MyComp>
    {{ stateField }}
</app-MyComp>
```



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

function MyComp () {
    const [ stateField, setStateField ] = useState<string>('Hey?')
    return (
        <>
            { stateField }
        </>
    );
}
```

# useState is a React Hook for state variable

The screenshot shows the React documentation interface. At the top, there's a navigation bar with a React logo, 'v19', a search bar ('Search'), keyboard shortcuts ('Ctrl K'), and links for 'Learn', 'Reference' (which is highlighted in blue), 'Community', and 'Blog'. Below the navigation, a sidebar lists various hooks: 'useImperativeHandle', 'useInsertEffect', 'useLayoutEffect', 'useMemo', 'useOptimistic', 'useReducer', 'useRef', 'useState' (which is also highlighted in blue), 'useSyncExternalStore', 'useTransition', and 'Components'. The main content area has a breadcrumb navigation ('API REFERENCE > HOOKS >') and a large title 'useState'. A descriptive paragraph explains that 'useState' is a React Hook that lets you add a state variable to your component. Below this, a code snippet shows the usage: `const [state, setState] = useState(initialState)`. The page also includes sections for 'Reference' (with links to 'useState(initialState)' and 'set functions, like setSomething(nextState)') and 'Usage'.

useImperativeHandle

useInsertEffect

useLayoutEffect

useMemo

useOptimistic

useReducer

useRef

useState

useSyncExternalStore

useTransition

Components >

API REFERENCE > HOOKS >

# useState

useState is a React Hook that lets you add a state variable to your component.

```
const [state, setState] = useState(initialState)
```

- Reference
  - useState(initialState)
  - set functions, like setSomething(nextState)
- Usage

# Notice ... state "Variable" is "const" !! but there is a setter

```
const [state, setState] = useState(initialState)
```

In TypeScript, this is equivalent of declaring 2 constants

```
const state: Type = initialState;
```

```
const setState: (value: Type) => void = function(value) { magic update that trigger re-rendering ... }
```

This does not compile "state = newValue", but ok with "setState(newValue)"

# Setter then Automatically re-rendering



In angular, you just do  
**this.state = newValue;**

=> it automatically re-renders

May use explicit "PushMode" for performance

May also use angular 17 "signal()" and  
"computed()"



In React, you just do  
**setState(newValue);**

=> it automatically re-renders

May also use "effect( () => { ... } )" to do outside of method call

# Why React Functions ? it used to be "Class" (now legacy)

The screenshot shows the React API Reference page for the `Component` class. The URL in the browser bar is `react.dev/reference/react/Component`. The page title is `Component`. On the left sidebar, under `Legacy React APIs`, the `Component` item is highlighted with a blue background. The main content area contains a **Pitfall** section with the text: "We recommend defining components as functions instead of classes. See how to migrate." Below this, a sidebar notes: "Component is the base class for the React components defined as JavaScript classes. Class components are still supported by React, but we don't recommend using them in new code." At the bottom, there is a code snippet for a class-based component:

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

At the very bottom left, there is a "Is this page useful?" button with a thumbs-up icon.

# React moving away from W3C Standard Spec " CustomElement are classes"



# HTML

**Living Standard — Last Updated 6 December 2024**

[← 4.12.5 The canvas element](#) — [Table of Contents](#) — [4.14 Common idioms without dedicated elements →](#)

[4.13 Custom elements](#)

[4.13.1 Introduction](#)

[4.13.1.1 Creating an autonomous custom element](#)

# CustomElement = class + register it

html.spec.whatwg.org/multipage/custom-elements.html#custom-elements-autonomous-example

## § 4.13.1.1 Creating an autonomous custom element

This section is non-normative.

For the purposes of illustrating how to create an [autonomous custom element](#), let's define a custom element that we can use like so:

```
<flag-icon country="nl"></flag-icon>
```

To do this, we first declare a class for the custom element, extending `HTMLElement`:

```
class FlagIcon extends HTMLElement {  
    constructor() {  
        super();  
        this._countryCode = null;  
    }  
  
    static observedAttributes = ["country"];  
  
    attributeChangedCallback(name, oldValue, newValue) {  
        // name will always be "country" due to observedAttributes  
        this._countryCode = newValue;  
        this._updateRendering();  
    }  
    connectedCallback() {  
        this._updateRendering();  
    }  
  
    get country() {  
        return this._countryCode;  
    }  
    set country(v) {  
        this.setAttribute("country", v);  
    }  
  
    _updateRendering() {  
        // Left as an exercise for the reader. But, you'll probably want to
```

class  
lifecycle  
methods

attr,  
getter,  
setter

We then need to use this class to define the element:

```
customElements.define("flag-icon", FlagIcon);
```

At this point, our above code will work! The parser, whenever it sees the `flag-icon` tag, will construct a `FlagIcon` object, which we then use to set the element's internal state and update its rendering (when it's inserted).

You can also create `flag-icon` elements using the DOM API:

```
const flagIcon = document.createElement("flag-icon")  
flagIcon.country = "jp"  
document.body.appendChild(flagIcon)
```

Finally, we can also use the [custom element constructor](#) itself. That is, the above code is equivalent to:

```
const flagIcon = new FlagIcon()  
flagIcon.country = "jp"  
document.body.appendChild(flagIcon)
```

# Exploring React Features

## Comparison with Angular

Html Rendering template in Component

```
const [x, setX] = useState()  
return (<> text { expr } </>)
```

→ {{ expr }}

Calling a child component from a Component

→ <app-child />

templating with "If" and "For"

→ @if(..) {} @else {} ..  
@for() {} ..

binding properties to a Component

→ field = input();  
<app-child [field]="expr">

binding Callback

→ onEvent = signal();  
<app-child (onEvent)="cb()">

Bi-Directional Binding ?

→ <app-child  
[(field)] ="parentField">

Routing ?

→ <a routerLink="/compA">  
or router.navigate(['compA'])

More ?

→ form, i18n, animation, etc etc

# Calling a child component from a Component



v19

Search

Ctrl K

Learn

Reference

[Your First Component](#)[Importing and Exporting Components](#)[Writing Markup with JSX](#)[JavaScript in JSX with Curly Braces](#)[Passing Props to a Component](#)[Conditional Rendering](#)[Rendering Lists](#)[Keeping Components Pure](#)[Your UI as a Tree](#)[Adding Interactivity](#) >[Managing State](#) >[Escape Hatches](#) >

A render tree represents a single render pass of a React application. With [conditional rendering](#), a parent component may render different children depending on the data passed.

We can update the app to conditionally render either an inspirational quote or color.

App.js ▾

↻ Reset ⌂ Fork

```
1 import FancyText from './FancyText';
2 import InspirationGenerator from './InspirationGenerator';
3 import Copyright from './Copyright';
4
5 export default function App() {
6   return (
7     <>
8       <FancyText title text="Get Inspired App" />
9       <InspirationGenerator>
10         <Copyright year={2004} />
11       </InspirationGenerator>
12     </>
13   );
14 }
```

# Exploring React Features

## Comparison with Angular

Html Rendering template in Component

```
const [x, setX] = useState()  
return (<> text { expr } </>)
```

→ {{ expr }}

Calling a child component from a Component

```
return (<> <AppChild /> </>)
```

→ <app-child />

templating with "If" and "For"

```
@if(..) { } @else { .. }  
@for() { .. }
```

binding properties to a Component

```
field = input();  
<app-child [field]="expr">
```

binding Callback

```
onEvent = signal();  
<app-child (onEvent)="cb()">
```

Bi-Directional Binding ?

```
<app-child  
[(field)] ="parentField">
```

Routing ?

```
<a routerLink="/compA">  
or router.navigate(['compA'])
```

More ?

form, i18n, animation, etc etc

# Equivalent of NG @if() { cond && <Elt/> }

## LEARN REACT

### Describing the UI

Your First Component

Importing and Exporting Components

Writing Markup with JSX

JavaScript in JSX with Curly Braces

Passing Props to a Component

Conditional Rendering

### Logical AND operator (&&)

Another common shortcut you'll encounter is the [JavaScript logical AND \(&&\) operator](#). Inside React components, it often comes up when you want to render some JSX when the condition is true, **or render nothing otherwise**. With `&&`, you could conditionally render the checkmark only if `isPacked` is `true`:

```
return (
  <li className="item">
    {name} {isPacked && '✓'}
  </li>
);
```

# plain old "If()", and local variable with <HTML>

The screenshot shows a left sidebar with navigation links and a main content area. The sidebar has a tree-like structure:

- Installation
- LEARN REACT
  - Describing the UI
    - Your First Component
    - Importing and Exporting Components
    - Writing Markup with JSX
    - JavaScript in JSX with Curly Braces
    - Passing Props to a Component
  - Conditional Rendering

The 'Conditional Rendering' link is highlighted with a blue background.

The main content area is titled 'App.js' and contains the following code:

```
1 function Item({ name, isPacked }) {  
2   let itemContent = name;  
3   if (isPacked) {  
4     itemContent = (  
5       <del>  
6         {name + " ✓"}  
7       </del>  
8     );  
9   }  
10  return (  
11    <li className="item">  
12      {itemContent}  
13    </li>  
14  );  
15}  
16
```

At the top right of the content area, there are three buttons: 'Download', 'Reset', and 'Fork'.

# equivalent of NG @if() { } @else { }

## LEARN REACT

### Describing the UI

Your First Component

Importing and Exporting Components

Writing Markup with JSX

JavaScript in JSX with Curly Braces

Passing Props to a Component

### Conditional Rendering

Rendering Lists

If you're not familiar with JavaScript, this variety of styles might seem overwhelming at first. However, learning them will help you read and write any JavaScript code — and not just React components! Pick the one you prefer for a start, and then consult this reference again if you forget how the other ones work.

## Recap

- In React, you control branching logic with JavaScript.
- You can return a JSX expression conditionally with an `if` statement.
- You can conditionally save some JSX to a variable and then include it inside other JSX by using the curly braces.
- In JSX, `{cond ? <A /> : <B />}` means “*if cond, render <A />, otherwise <B />*”.
- In JSX, `{cond && <A />}` means “*if cond, render <A />, otherwise nothing*”.
- The shortcuts are common, but you don't have to use them if you prefer plain `if`.

# Equivalent of NG @for() using JS .map(x => (<> {x} </>) ) function!

Installation

LEARN REACT

Describing the UI

Your First Component

Importing and Exporting Components

Writing Markup with JSX

JavaScript in JSX with Curly Braces

Passing Props to a Component

Conditional Rendering

Rendering Lists

Keeping Components Pure

Your UI as a Tree

1. Move the data into an array:

```
const people = [
  'Creola Katherine Johnson: mathematician',
  'Mario José Molina-Pasquel Henríquez: chemist',
  'Mohammad Abdus Salam: physicist',
  'Percy Lavon Julian: chemist',
  'Subrahmanyan Chandrasekhar: astrophysicist'
];
```

2. Map the `people` members into a new array of JSX nodes, `listItems`:

```
const listItems = people.map(person => <li>{person}</li>);
```

3. Return `listItems` from your component wrapped in a `<ul>`:

```
return <ul>{listItems}</ul>;
```

# Exploring React Features

## Comparison with Angular

Html Rendering template in Component

```
const [x, setX] = useState()  
return (<> text { expr } </>)
```

→ {{ expr }}

Calling a child component from a Component

```
return (<> <AppChild /> </>)
```

→ <app-child />

 templating with "If" and "For"

```
{cond && (<> </>) }  
{items.map(i => (<> {i} </>) ) }
```

→ @if(..) { } @else { .. }  
@for() { .. }

binding properties to a Component

→ field = input();  
<app-child [field]="expr">

binding Callback

→ onEvent = signal();  
<app-child (onEvent)="cb()">

Bi-Directional Binding ?

→ <app-child  
[(field)] ="parentField">

Routing ?

→ <a routerLink="/compA">  
or router.navigate(['compA'])

More ?

→ form, i18n, animation, etc etc

# Passing Properties equivalent to NG input() or "@Input"

## Describing the UI

Your First Component

Importing and Exporting Components

Writing Markup with JSX

JavaScript in JSX with Curly Braces

## Passing Props to a Component

Conditional Rendering

Rendering Lists

Now you can read these props inside the `Avatar` component.

## Step 2: Read props inside the child component

You can read these props by listing their names `person`, `size` separated by the commas inside `({ } and })` directly after `function Avatar`. This lets you use them inside the `Avatar` code, like you would with a variable.

```
function Avatar({ person, size }) {  
  // person and size are available here  
}
```

# Reminder... TypeScript Object Parameter De-Structuring

In Typescript (nothing specific to React)

```
myFunc ( { a: number, b: string, ...others } ) { a ... }
```

is equivalent to ..

```
interface TProps {  
    a: number;  
    b: string;  
}  
myFunc ( props : TProps ) { props.a ... }
```

... NOTICE: props can contains others members

# Exploring React Features

## Comparison with Angular

Html Rendering template in Component

```
const [x, setX] = useState()  
return (<> text { expr } </>)
```

→ {{ expr }}

Calling a child component from a Component

```
return (<> <AppChild /> </>)
```

→ <app-child />

templating with "If" and "For"

```
{cond && (<> </>) }  
{items.map(i => (<> {i} </>) ) }
```

→ @if(..) {} @else {} ..  
@for() {} ..

binding properties to a Component

```
function AppChild({ field }) {..}  
<AppChild field={expr} />
```

→ field = input();  
<app-child [field]="expr">

binding Callback

→ onEvent = signal();  
<app-child (onEvent)="cb()">

Bi-Directional Binding ?

→ <app-child  
[(field)] ="parentField" />

Routing ?

→ <a routerLink="/compA">  
or router.navigate(['compA'])

More ?

→ form, i18n, animation, etc etc

# Responding to Event



v19

Search

Ctrl K

Learn

Reference

## GET STARTED

[Quick Start](#)[Installation](#)

## LEARN REACT

[Describing the UI](#)[Adding Interactivity](#)[Responding to Events](#)[State: A Component's Memory](#)

### App.js

[Download](#) [Reset](#) [Fork](#)

```
1  export default function Button() {  
2      function handleClick() {  
3          alert('You clicked me!');  
4      }  
5  
6      return (  
7          <button onClick={handleClick}>  
8              Click me  
9          </button>  
10     );  
11  }  
12
```

# binding callback Outside of Component

.. same as properties: pass function callback

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

const ShippingForm = memo(function ShippingForm({ onSubmit }) {
  // ...
});
```

# !! Re-Rendering Performance !!

Overview

Hooks

useActionState

useCallback

useContext

useDebugValue

useDeferredValue

useEffect

useId

useImperativeHandle

useInsertionEffect

With this change, `ShippingForm` will skip re-rendering if all of its props are the same as on the last render. This is when caching a function becomes important! Let's say you defined `handleSubmit` without `useCallback`:

```
function ProductPage({ productId, referrer, theme }) {
  // Every time the theme changes, this will be a different function...
  function handleSubmit(orderDetails) {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }

  return (
    <div className={theme}>
      {/* ... so ShippingForm's props will never be the same, and it will re-render every time */}
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

# pass Function pointer, or recreate: function() { .. } or () => { .. }

## GET STARTED

Quick Start >

Installation >

## LEARN REACT

Describing the UI >

Adding Interactivity ▾

  Responding to Events

  State: A Component's Memory

  Render and Commit

  State as a Snapshot

  Queueing a Series of State Updates

  Updating Objects in State

  Updating Arrays in State

Managing State >

Escape Hatches >

## Pitfall

Functions passed to event handlers must be passed, not called. For example:

passing a function (correct)	calling a function (incorrect)
<button onClick={handleClick}>	<button onClick={handleClick()}>

The difference is subtle. In the first example, the `handleClick` function is passed as an `onClick` event handler. This tells React to remember it and only call your function when the user clicks the button.

In the second example, the `()` at the end of `handleClick()` fires the function *immediately* during `rendering`, without any clicks. This is because JavaScript inside the `JSX { and }` executes right away.

When you write code inline, the same pitfall presents itself in a different way:

passing a function (correct)	calling a function (incorrect)
<button onClick={() => alert('...')}>	<button onClick={alert('...')}>

# binding callback Outside of Component wrap with Hook "useCallback(() => ...)"

react@19

Overview

Hooks

useActionState

useCallback

useContext

useDebugValue

useDeferredValue

useEffect

useId

useImperativeHandle

useInsertionEffect

## Skipping re-rendering of components

When you optimize rendering performance, you will sometimes need to cache the functions that you pass to child components. Let's first look at the syntax for how to do this, and then see in which cases it's useful.

To cache a function between re-renders of your component, wrap its definition into the `useCallback` Hook:

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

function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);
  // ...
}
```

<App onXX={useCallback((x) => f(x,y), [y])}>

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

function ProductPage({ productId, referrer, theme }) {
  const handleSubmit = useCallback((orderDetails) => {
    post('/product/' + productId + '/buy', {
      referrer,
      orderDetails,
    });
  }, [productId, referrer]);

  // ...
  return (
    <div className={theme}>
      <ShippingForm onSubmit={handleSubmit} />
    </div>
  );
}
```

# Exploring React Features

## Comparison with Angular

Html Rendering template in Component

```
const [x, setX] = useState()  
return (<> text { expr } </>)
```

→ {{ expr }}

Calling a child component from a Component

```
return (<> <AppChild /> </>)
```

→ <app-child />

templating with "If" and "For"

```
{cond && (<> </>) }  
{items.map(i => (<> {i} </>) ) }
```

→ @if(..) {} @else {} ..  
@for() {} ..

binding properties to a Component

```
function AppChild({ field }) {..}  
<AppChild field={expr} />
```

→ field = input();  
<app-child [field]="expr">

binding Callback

```
<AppChild onX="{ () => f() }"  
on={useCallback((x)=>f(x,y), [y])}>
```

→ onEvent = signal();  
<app-child (onEvent)="cb()">

Bi-Directional Binding ?

→ <app-child  
[(field)] ="parentField">

Routing ?

→ <a routerLink="/compA">  
or router.navigate(['compA'])

More ?

→ form, i18n, animation, etc etc

# NO Bi-Directional Binding in React !!??

in react, is there bi-directional binding ?



In React, there is no native support for **bi-directional binding** like in frameworks such as Angular. Instead, React follows a **unidirectional data flow**, where:

1. **State** flows from parent to child components via props.
2. **Events** flow from child to parent via callback functions.

That said, you can implement functionality similar to bi-directional binding by combining **state management** and **event handlers**. This is commonly used in form elements.

# useState + explicit {render} + callback setter

tsx

```
import React, { useState } from "react";

const BiDirectionalBinding: React.FC = () => {
  const [value, setValue] = useState<string>("");

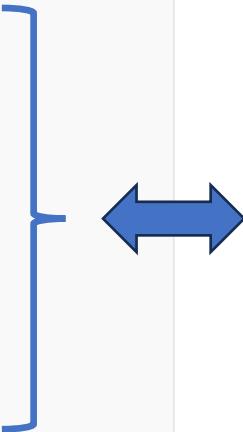
  const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setValue(event.target.value); // Updates state when input changes
  };

  return (
    <div>
      <input type="text" value={value} onChange={handleChange} />
      <p>Current Value: {value}</p>
    </div>
  );
}

export default BiDirectionalBinding;
```



Copy code



value = "";

<input type="text" [(ngModel)]="value">

## How it Works:

1. The `value` prop of the input element is controlled by the `value` state.
2. The `onChange` event updates the state when the user types in the input field.
3. The state (`value`) flows back into the input, creating a loop resembling bi-directional binding.



# Bi-directional Bindings Frameworks, or Redux ?

frameworks for react bi-directional bindings ?



Here are some frameworks and libraries that facilitate **bi-directional bindings** in React applications. These tools simplify synchronizing state and user input, ensuring updates propagate seamlessly between the UI and the underlying data model.

## 1. React Hook Form

- **Description:** Lightweight library for managing form state in React.
- **Features:**
  - Minimizes re-renders for better performance.
  - Simple integration with validation libraries (e.g., Yup, Zod).
  - Supports complex forms with nested fields.

## 4. Redux Form (Deprecated but still used)

- **Description:** Extends Redux for form state management.
- **Features:**
  - Form state is stored in the Redux store.
  - Supports validation and field-level state.

## 7. Zustand

- **Description:** Lightweight state management library.
- **Features:**
  - Easy-to-use and small footprint.
  - Great for local and global state in forms.

## 2. Formik

- **Description:** Form state management library, ideal for handling forms and validation.
- **Features:**
  - Declarative form management.
  - Built-in support for validation with Yup.
  - Suitable for larger, complex forms.

## 5. React Final Form

- **Description:** Lightweight library for managing form state, similar to Redux Form.
- **Features:**
  - Declarative form state management.
  - Built-in validation support.

## 8. Valtio

- **Description:** A proxy-based state management library.
- **Features:**
  - Seamless bi-directional bindings with React.
  - Reactive state updates without actions or boilerplate.

## 3. MobX

- **Description:** State management library emphasizing reactivity.
- **Features:**
  - Automatic UI updates for observable state changes.
  - Simple, scalable, and great for bi-directional binding.

## 6. Recoil

- **Description:** State management library for React, designed for fine-grained state updates.
- **Features:**
  - Atom-based state management for easy form state handling.
  - Reactivity with minimal boilerplate.

# One Framework a Day, The Doctor Away

## Summary of Frameworks:

Framework/Library	Key Features	When to Use
React Hook Form	Lightweight, minimal re-renders	Simple to medium forms with validation.
Formik	Declarative, Yup integration	Large, complex forms with validation needs.
MobX	Reactive state management	Apps needing global state with form support.
Redux Form	Form state in Redux store	Legacy projects already using Redux.
React Final Form	Declarative and simple	Alternative to Redux Form.
Recoil	Atom-based state	Fine-grained state management.
Zustand	Lightweight, flexible	Simple apps or lightweight state needs.
Valtio	Proxy-based reactive state	Modern apps with minimal boilerplate.

Choose based on your app's complexity, performance requirements, and familiarity with the library.

# Using MobX

sample react bi-directional binding with mobx



Using MobX for bi-directional data binding in a React application involves setting up observable state and binding it to input elements. Here's a step-by-step example:

---

## 1. Install MobX and MobX-React

bash

Copy code

```
npm install mobx mobx-react-lite
```

# MobX Store (=Model, ~Model-View pattern)

## 2. Create an Observable Store

Define a MobX store with an observable property and an action to update it.

typescript

```
// store.ts
import { makeAutoObservable } from "mobx";

class FormStore {
  name: string = "";

  constructor() {
    makeAutoObservable(this);
  }

  setName(value: string) {
    this.name = value;
  }
}

export const formStore = new FormStore();
```

Copy code

## 3. React Component with Bi-Directional Binding

Use `observer` from `mobx-react-lite` to make the React component reactive.

tsx

```
// App.tsx
import React from "react";
import { observer } from "mobx-react-lite";
import { formStore } from "./store";

const App = observer(() => {
  return (
    <div>
      <label>
        Name:
        <input
          type="text"
          value={formStore.name}
          onChange={(e) => formStore.setName(e.target.value)}
        />
      </label>
      <p>Typed Name: {formStore.name}</p>
    </div>
  );
});
```

Copy code

# Exploring React Features

## Comparison with Angular

Html Rendering template in Component

```
const [x, setX] = useState()  
return (<> text { expr } </>)
```

→ {{ expr }}

Calling a child component from a Component

```
return (<> <AppChild /> </>)
```

→ <app-child />

templating with "If" and "For"

```
{cond && (<> </>) }  
{items.map(i => (<> {i} </>) ) }
```

→ @if(..) {} @else {} ..  
@for() {} ..

binding properties to a Component

```
function AppChild({ field }) {..}  
<AppChild field={expr} />
```

→ field = input();  
<app-child [field]="expr">

binding Callback

```
<AppChild onX="{ () => f() }"  
on={useCallback((x)=>f(x,y), [y])}>
```

→ onEvent = signal();  
<app-child (onEvent)="cb()">

Bi-Directional Binding ?

NOT NATIVE.. use Mobx?

→ <app-child  
[(field)] ="parentField">

Routing ?

→ <a routerLink="/compA">  
or router.navigate(['compA'])

More ?

→ form, i18n, animation, etc etc

# NO native Routing in React !!

is there routing native in react ?



No, **routing is not natively built into React**. React is a JavaScript library focused solely on building user interfaces, and it does not include routing or other advanced features like state management or data fetching. These functionalities can be added through third-party libraries, with **React Router** being the most commonly used library for routing in React applications.

## Why Isn't Routing Native to React?

React adheres to a "library" philosophy rather than being a full-fledged framework like Angular. This means:

1. It focuses on **UI rendering and component composition**.
2. Other functionalities, such as routing, state management, and side effects, are left to external libraries.  


# npm install react-router-dom

```
npm install react-router-dom
```

## Basic Example

### Setting up Routes:

1. Wrap your app with `<BrowserRouter>` : This enables routing in your application.
2. Define routes using `<Routes>` and `<Route>` : Specify the paths and their associated components.

tsx

 Copy code

```
import React from "react";
import ReactDOM from "react-dom";
import { BrowserRouter, Routes, Route } from "react-router-dom";

const Home = () => <h1>Home Page</h1>;
const About = () => <h1>About Page</h1>;

const App = () => {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </BrowserRouter>
  );
};

ReactDOM.render(<App />, document.get↓lementById("root"));
```

# Exploring React Features

## Comparison with Angular

Html Rendering template in Component

```
const [x, setX] = useState()  
return (<> text { expr } </>)
```

→ {{ expr }}

Calling a child component from a Component

```
return (<> <AppChild /> </>)
```

→ <app-child />

templating with "If" and "For"

```
{cond && (<> </>) }  
{items.map(i => (<> {i} </>) ) }
```

→ @if(..) {} @else {} ..  
@for() {} ..

binding properties to a Component

```
function AppChild({ field }) {..}  
<AppChild field={expr} />
```

→ field = input();  
<app-child [field]="expr">

binding Callback

```
<AppChild onX="{ () => f() }"  
on={useCallback((x)=>f(x,y), [y])}>
```

→ onEvent = signal();  
<app-child (onEvent)="cb()">

Bi-Directional Binding ?

NOT NATIVE.. use Mobx?

→ <app-child  
[(field)] ="parentField">

✓ Routing ?

NOT NATIVE.. use react-router?

→ <a routerLink="/compA">  
or router.navigate(['compA'])

More ?

→ form, i18n, animation, etc etc

# No Form in native React ?!!

how to have form with validation in native react?



Creating a form with validation in **native React** involves managing the form state and implementing validation logic yourself. React doesn't have built-in form handling or validation utilities, but you can achieve this using **state** and **event handlers**.

Here's a step-by-step guide to creating a form with validation in native React:

# Exploring React Features

## Comparison with Angular

Html Rendering template in Component

```
const [x, setX] = useState()  
return (<> text { expr } </>)
```

→ {{ expr }}

Calling a child component from a Component

```
return (<> <AppChild /> </>)
```

→ <app-child />

templating with "If" and "For"

```
{cond && (<> </>) }  
{items.map(i => (<> {i} </>) ) }
```

→ @if(..) {} @else {} ..  
@for() {} ..

binding properties to a Component

```
function AppChild({ field }) {..}  
<AppChild field={expr} />
```

→ field = input();  
<app-child [field]="expr">

binding Callback

```
<AppChild onX="{ () => f() }"  
on={useCallback((x)=>f(x,y), [y])}>
```

→ onEvent = signal();  
<app-child (onEvent)="cb()">

Bi-Directional Binding ?

NOT NATIVE.. use Mobx?

→ <app-child  
[(field)] ="parentField">

Routing ?

NOT NATIVE.. use react-router?

→ <a routerLink="/compA">  
or router.navigate(['compA'])

More ?

NOT NATIVE.. do it yourself !

→ form, i18n, animation, etc etc

# React PROs

React is very fashion since ~4 years

Used in lot of companies and Open-Source Projects

becoming a de-facto standard, among the top 3 frameworks: React, Angular, Vue.Js

# React PROS

Simplistic things (no binding, no state) are possible in a simple way in React

Very easy to call stateless sub components, as Pure functions.

React forces you to use "pure" function, with exceptions to "purity"

React favors a functional approach, and encourage creating many small function components

TSX is compacting both the HTML and the TS/JS in the same location (is it a PROS or a CONS?)

# React PROS (or CONS ?)

React is Minimalist : only Rendering DOM

React "looks" SIMPLER than Angular, because there is almost NOTHING in it !

React is just a DOM rendering template engine

React features is a SMALL subset of Angular: "{{ expr }}", [field]="", (event)=""

major features relies on TSX/JSX for html <-> {JavaScript} interleaving  
and passing parameters in JS/TS functions

# React CONS : no class going away from W3c CustomElement Standard

The Hook "functionalities" in React are very tricky / difficult to understand.

Example: useState(), useEffect(), useCallBack() ...

They are counter intuitives, and need HUGE documentation/training to master strange effects compared to the very SMALL features they provide

React used to have classes, but now legacy !?

... going away from Standard W3C Specification of "Custom Element" as classes

# React CONS : Rendering with Virtual DOM

React = Only a "rendering" engine, using a Virtual DOM  
very bad for memory performances : everything x2 for memory

Once upon a time, It used to be "good" to have Virtual DOM  
because Microsoft Internet Explorer DOM was so bad. now useless.

Example of lighter/modern alternatives: Solid.JS, Svelte, Vue.js  
Solid.JS = very similar to React TSX, but NO Virtual DOM... only real fast DOM

# React CONS

React looks **SIMPLER** than Angular, because there is almost **NOTHING** in it !

For all features, you need additional frameworks : MOBX, Router, etc..

There is

- No Router
- No Bi-Directional Binding
- tons of frameworks, difficult to choose
- Redux is a headache, overkill
- No Form

# Final Words

Google search results for "top web framework trends". The snippet reads: "React, jQuery, Angular, Vue.js, Svelte, Express.js, ASP.NET Core, Django, Ruby on Rails, Laravel are the 10 most popular web development frameworks in 2025. Sep 6, 2024"

React is #1 in 2024

The Web is Absurd ... why jQuery still in 2024? rails? ... confusing SPA technologies and server-side



# Questions ?

arnaud.nauwynck@gmail.com