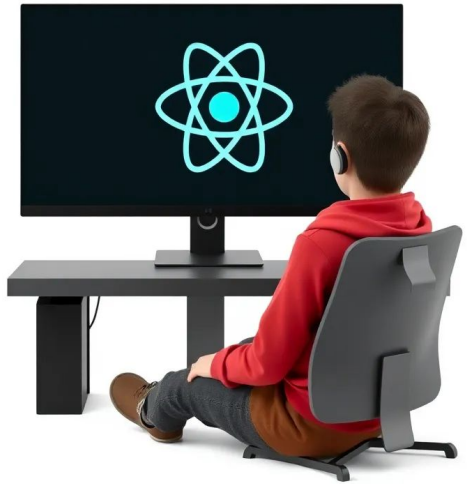
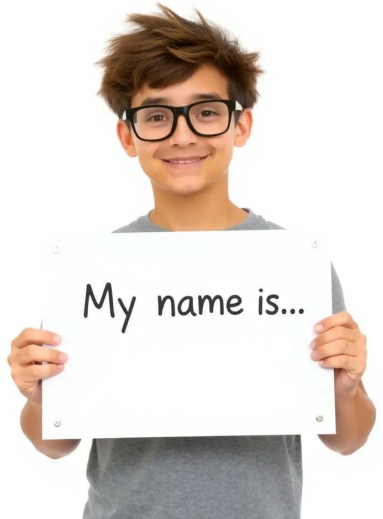
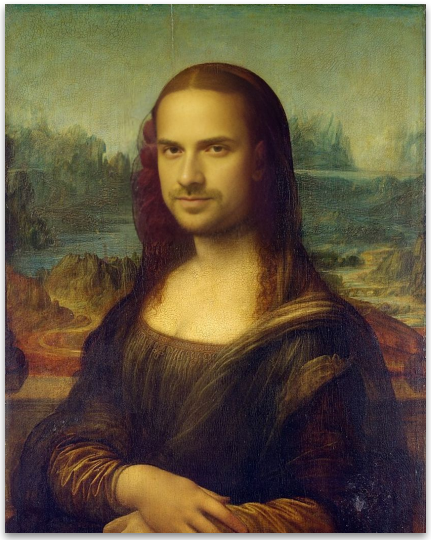
# Introduction to React



# Introduction Round

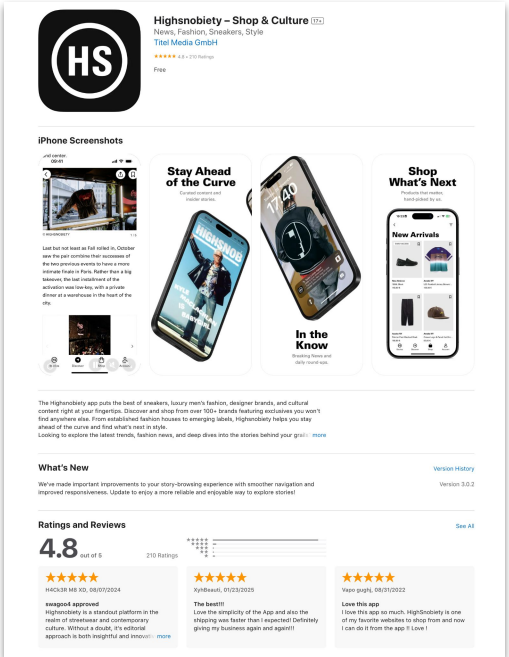
- Name
- Current area of responsibility in the company
- Prior knowledge in Web Development
- An interesting fact about yourself (optional)

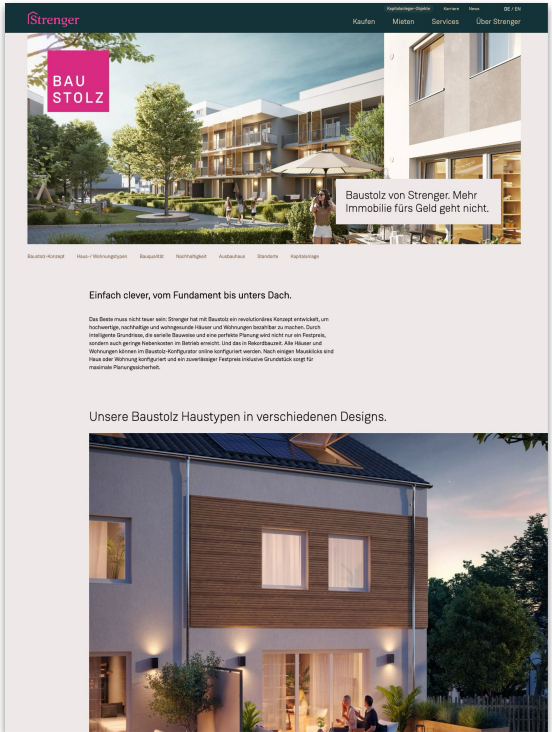


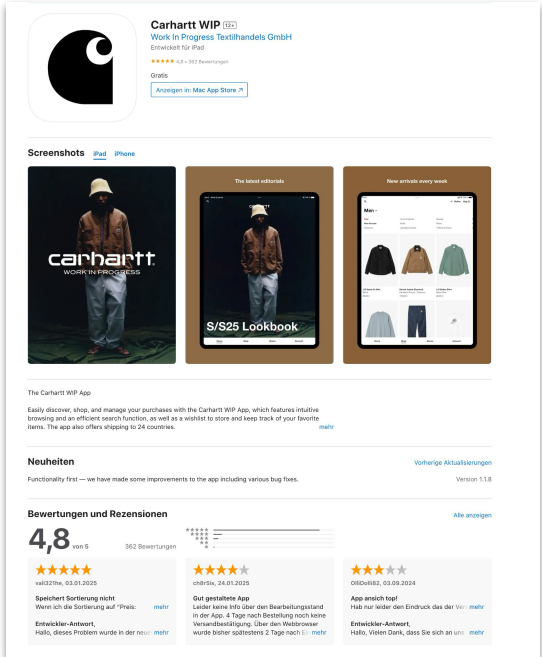


- **Entwickler seit 1590 (geschätzt)** – Schon in den 90er Jahren Webseiten mit HTML & JavaScript gebaut.
- **Von Entwicklung zu KI & Sicherheit** – Technik-Enthusiast, aber JavaScript war meine erste “Entwicklerliebe”.
- **Informatik Master & MBA** – Eine Mischung aus technischer Expertise und Unternehmertum.
- **Theorie? Klar. Praxis? Noch mehr.** – Vom Entwickler zum CTO – ich habe Einiges gesehen.
- https://www.youtube.com/**@ai-for-devs**

# Vergangene React Projekte



# Expectations

- Grundsätzliches Verständnis
- Wie arbeite ich mit React
- States
- TypeScript Grundlagen
- Grundwissen auf dem man weiter aufbauen kann
- React Umfeld (Anbindung an System, Kafka
- APIs (REST, GraphQL)
- Umgang mit Tabellen (Nutzereingabe, Relationale Datenbank simulieren)
- Debugging

- Microservices abwegen
- Node (Basics)
- AWS (wird inHouse genutzt), Hosting-Möglichkeiten (Scaling)
- Dynamo (key, value)
- ...

# Day 1: React Fundamentals and Initial Steps

- JavaScript Basics
- Overview of React and Its Ecosystem
- Insights into Virtual DOM Functionality
- Fundamental Principles of React
- Understanding Components and Properties
- Comprehensive Analysis of JSX
- Initiating Your First Project with Next.js
- Methods of Conditional Rendering in React
- Approaches to Render Lists in React

# Day 2: Advanced React Development

- Building a Portfolio Page Using React
- Handling Events and Propagation in React
- State Management in React: Techniques and Practices
- Integrating React with External Systems
- Dynamic Routing Implementation in React
- Data Retrieval from External APIs in React
- Employing React's Context API for Efficient State Management

# Day 3: Leveraging TypeScript and React Advanced Topics

- Introduction to TypeScript for React
- Utilizing Type Aliases in TypeScript
- Interface Implementation in TypeScript
- TypeScript Functions: Definition and Usage
- Generic Types in TypeScript: An Overview
- Applying TypeScript in React Projects
- Comparing Server vs Client Rendering Techniques
- Exploring React in the Context of React Native
- Basics of IT Security in Web Development

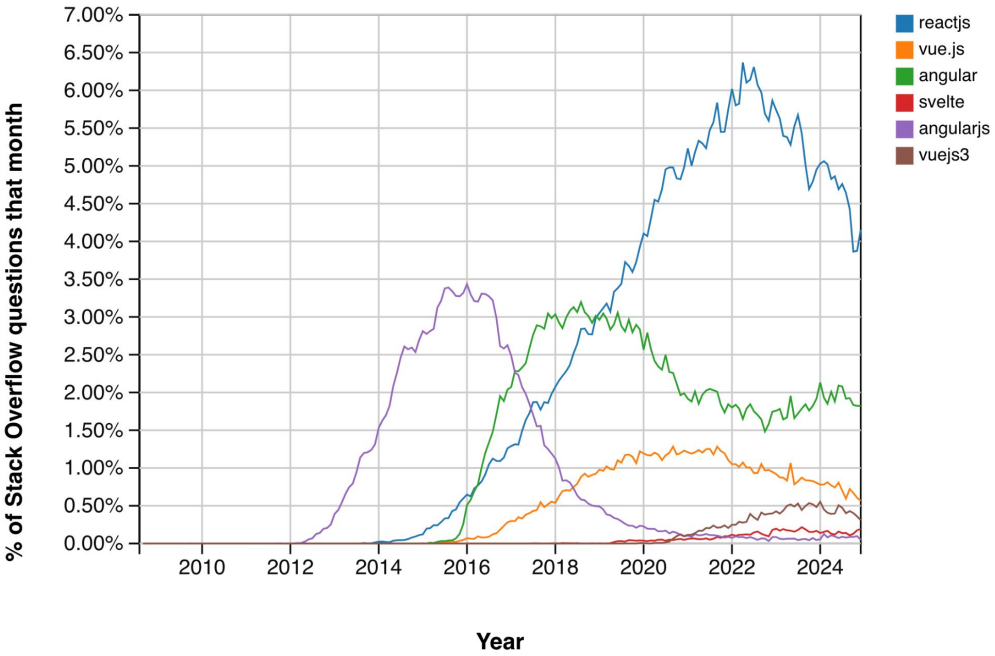
# JavaScript Popularity

# JavaScript Popularity



| Feb 2025 | Feb 2024 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 1 | | Python | 23.88% | +8.72% |
| 2 | 3 | ↑ | C++ | 11.37% | +0.84% |
| 3 | 4 | ↑ | Java | 10.66% | +1.79% |
| 4 | 2 | ↓ | C | 9.84% | -1.14% |
| 5 | 5 | | C# | 4.12% | -3.41% |
| 6 | 6 | | JavaScript | 3.78% | +0.61% |
| 7 | 7 | | SQL | 2.87% | +1.04% |
| 8 | 8 | | Go | 2.26% | +0.53% |
| 9 | 12 | ↑ | Delphi/Object Pascal | 2.18% | +0.78% |
| 10 | 9 | ↓ | Visual Basic | 2.04% | +0.52% |
| 11 | 11 | | Fortran | 1.75% | +0.35% |
| 12 | 15 | ↑ | Scratch | 1.54% | +0.36% |
| 13 | 18 | ⇈ | Rust | 1.47% | +0.42% |
| 14 | 10 | ⇊ | PHP | 1.14% | -0.37% |
| 15 | 21 | ⇈ | R | 1.06% | +0.07% |
| 16 | 13 | ↓ | MATLAB | 0.98% | -0.28% |
| 17 | 14 | ↓ | Assembly language | 0.95% | -0.24% |
| 18 | 19 | ↑ | COBOL | 0.82% | -0.18% |
| 19 | 20 | ↑ | Ruby | 0.82% | -0.17% |
| 20 | 24 | ⇈ | Prolog | 0.80% | +0.03% |

# React Popularity



# Role of JS Developer in 2025?


# JavaScript Essentials

# JavaScript Essentials

# JavaScript Essentials

Core JavaScript knowledge is crucial for React.

- Variables & Scope: let, const, understanding function vs. block scope.
- Data Structures: Working with objects, arrays, and JSON.
- Functions: Defining and using functions, arrow functions, and scope.
- Asynchronous JS: Promises, async/await, handling asynchronous operations.
- ES6 Features: Destructuring, spread operator, template literals.
- DOM vs HTML
- Module System (Import & Export)

# Variables & Scope - *var*

- If you declare a variable with var inside a function, it’s scoped to that function.
- Outside of a function, it becomes a global variable.
- Declarations made using var are hoisted to the top of their scope.

```js
function exampleVar() {
  console.log(x); // undefined (due to hoisting)
  var x = 10;
  console.log(x); // 10
}
exampleVar();
```

# Variables & Scope - *let*

- A variable declared with let is limited to the nearest enclosing block.
- If you declare a variable with let, you cannot redeclare it again in the same scope.

```js
function exampleLet() {
  // console.log(y); // ReferenceError if uncommented (TDZ)
  let y = 10;
  console.log(y); // 10
  // let y = 20; // SyntaxError: Identifier 'y' has already been declared
}
exampleLet();
```

# Variables & Scope - *global*

- Creates a global variable x on the window (in browsers) or global object (in Node.js).
- This is generally considered bad practice as it can lead to accidental global variables.
- In strict mode, this will throw a ReferenceError

```
js                                                    Copy  Edit

x = 10;
```

# Variables & Scope - *const*

- Like let, a const-declared variable is scoped to the nearest enclosing block.
- A variable declared with const must be assigned an initial value.
- Once a const variable has been assigned, you cannot reassign it

```js
const x = 10;
x = 20; // TypeError: Assignment to constant variable
```

# Essentials: Data Structures

Objects, arrays, and JSON are foundational structures in JS.

```
// Objects store key-value pairs
const person = {
  name: 'Alice',
  age: 25,
};

// Arrays hold ordered collections
const fruits = ['apple', 'banana', 'cherry'];

// JSON - Stringified object
const jsonData = JSON.stringify(person);
```

# Essentials: Functions

Functions are blocks of reusable code, and arrow functions offer a concise syntax.

```
// Function declaration
function greet(name) {
  return `Hello, ${name}!`;
}

// Arrow function with implicit return
const greet = (name) => `Hello, ${name}!`;

// Arrow function with explicit return
const greet = (name) => {
  return `Hello, ${name}!`;
};
```

# Essentials: Asynchronous JS

Handle asynchronous operations with Promises and async/await

```
// Promise usage
fetch('https://api.example.com/data')
  .then(response => response.json())
  .then(data => console.log(data));

// Async/await syntax
async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  console.log(data);
}
```

# Laufzeitumgebungen

1. **Browser**: Webbrowser bieten eine Umgebung, in der JavaScript-Code direkt auf Webseiten ausgeführt wird.

2. **Node.js**: Eine serverseitige Plattform, die es ermöglicht, JavaScript außerhalb des Browsers auszuführen.

# ECMAScript und JavaScript

| ECMAScript | JavaScript |
| --- | --- |
| Ein Standard für Skriptsprachen. | Die populärste Implementierung des ECMAScript-Standards. |
| Sprachen wie JS basieren auf dem ECMAScript-Standard. | Die Kernfunktionen von JavaScript basieren auf ECMAScript. |
| Basierend auf mehreren Technologien, darunter JavaScript (Netscape) und JScript (Microsoft). | JavaScript wurde zur Standardisierung bei ECMA eingereicht. Aufgrund von Markenrechtsproblemen mit dem Namen "JavaScript" wurde der Standard als "ECMAScript" bezeichnet. |

# Wichtigste ECMAScript 6 (ES6) Features

- **Klassen:** Einführung von Klassen-basierten Objektorientierung.
- **Pfeilfunktionen (=>):** Kompaktere Syntax für anonyme Funktionen und kein Binden von this.
- **Template Strings (` `):** Einfache String-Interpolation und Mehrzeilige Strings.
- **Destrukturierung:** Schnelles Extrahieren von Werten aus Arrays oder Objekten
- **let & const:** Block-Scoped-Variablen, let für veränderbare und const für unveränderbare Werte.
- **Iteratoren & Generatoren:** Neue Wege, um durch Daten zu iterieren

*Hinweis: Alle diese ECMAScript-Features werden zu 100% von TypeScript unterstützt.*

🔗http://w3schools.com/js/js_versions.asp

# Essentials: ES6 Features

ES6 introduced a more expressive syntax for writing JS.

```
// Destructuring an object
const { name, age } = person;

// Spread operator for array
const newFruits = [...fruits, 'dragonfruit'];

// Template literals for strings
const greeting = `My name is ${name} and I am ${age} years old.`;
```

# Essentials: Module System

Reusable pieces of code can be organized into modules, which can be imported and exported to enhance modularity and reusability.

```
// export-example.js
export const add = (a, b) => a + b;
export const subtract = (a, b) => a - b;

// import-example.js
import { add, subtract } from './export-example.js';

const sum = add(2, 3);
const difference = subtract(5, 2);
```

# Exercise: Superhero Creator 🦸

- Create an object representing a superhero with properties like `name`, `power`, and `city`.
- Use const to ensure the superhero object itself cannot be reassigned.
- Use `let` to allow modifying a hero's stats dynamically.
- Use destructuring to extract and display hero properties.
- Use template literals to print a cool introduction

```js
// 1. Erstelle ein Superhelden-Objekt mit const
const superhero = {
    ...: "...",
    ...: "...",
    ...: "...",
    ...: ...
};

// 2. Nutze Destructuring, um Werte aus dem Objekt zu extrahieren
const { ..., ..., ..., ... } = superhero;

// 3. Gib die Superhelden-Infos mit Template Literals aus
console.log(`💥 Meet ..., the guardian of ...!`);
console.log(`⚡ Superpower: ...`);
console.log(`💪 Strength Level: ...`);

// 4. Verwende let, um eine Eigenschaft dynamisch zu aktualisieren
let newStrength = ...;
console.log(`🔥 After training, ...'s ... increased to ...!`);

// 5. Versuche, das gesamte Objekt neu zuzuweisen (dies sollte fehlschlagen)
// superhero = {}; // ❌ Fehler: Assignment to constant variable

console.log("💥 Superhero setup complete! Ready to save the world!");
```

# Solution: Superhero Creator 🦸

js

Copy Edit

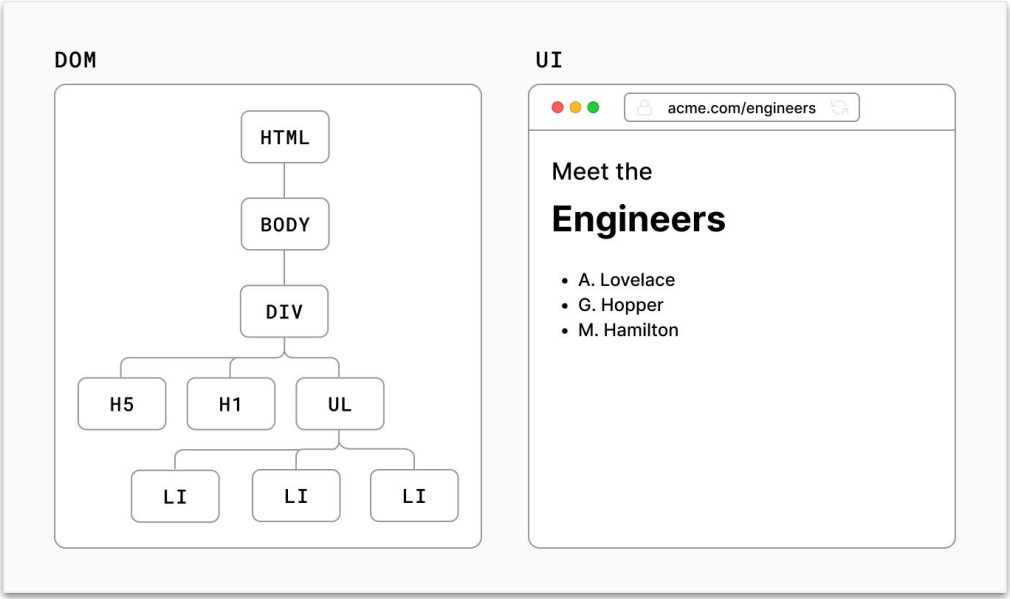
```js
// 1. Define a superhero using const and an object
const superhero = {
    name: "Lightning Bolt",
    power: "Super Speed",
    city: "Metropolis",
    strength: 80
};

// 2. Use destructuring to extract properties
const { name, power, city, strength } = superhero;

// 3. Print superhero details using template literals
console.log(`💥 Meet ${name}, the guardian of ${city}!`);
console.log(`⚡ Superpower: ${power}`);
console.log(`💪 Strength Level: ${strength}`);

// 4. Use let to modify the strength of the superhero
let newStrength = strength + 20;
console.log(`🔥 After training, ${name}'s strength increased to ${newStrength}!`);

// 5. Try to reassign the superhero object (this should fail)
// superhero = {}; // ❌ Error: Assignment to constant variable

console.log("🎇 Superhero setup complete! Ready to save the world!");
```

# DOM

# HTML vs. the DOM

# What is the DOM?

The DOM is an object representation of the HTML elements. It acts as a bridge between your code and the user interface, and has a tree-like structure with parent and child relationships.



# DOM Manipulation

**Document Access & Selection**

- Interact with the page structure and find elements using `getElementById()` or `querySelector()`.

**Event Handling**

- Use `addEventListener()` to respond to clicks, key presses, or other user actions.

**Updating the DOM**

- Change text, attributes, or styles (`innerHTML`, `setAttribute()`, etc.).

**Creating & Removing Elements**

- Generate new nodes with `createElement()` and attach them using `appendChild()`.

# DOM Manipulation

```html
<html>
    <head> </head>
    <body>
      <div id="app"></div>
        <script>
          const app = document.getElementById('app');
          const header = document.createElement('h1');
          const text = 'Develop. Preview. Ship.';
          const headerContent = document.createTextNode(text);
          header.appendChild(headerContent);
          app.appendChild(header);

          const button = document.createElement('button');
          const buttonText = 'Click me';
          const buttonContent = document.createTextNode(buttonText);
          button.appendChild(buttonContent);
          app.appendChild(button);

          button.addEventListener('click', () => {
            alert('Hello');
          });

        </script>
    </body>
</html>
```

# HTML vs. the DOM

Open the developer tools to see the live DOM, which may differ from the original HTML due to newly added or updated elements.

DOM

Elements

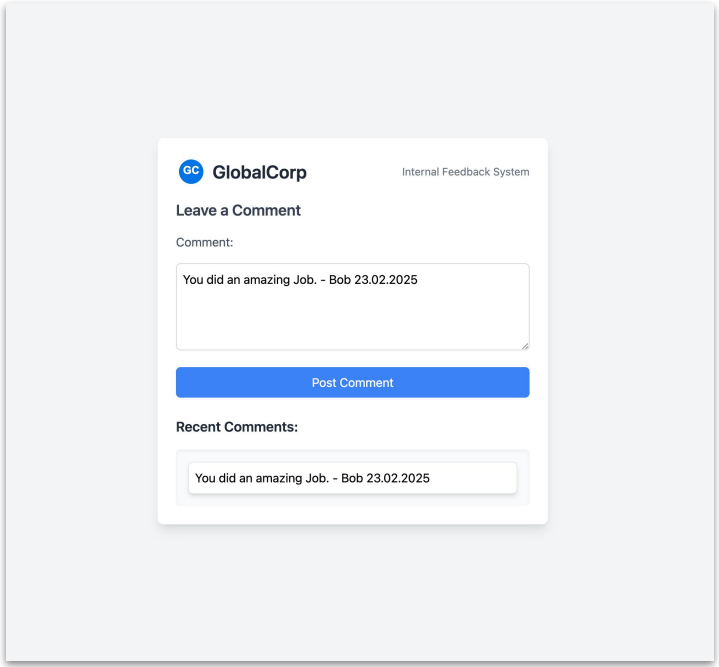
```
1  <html>
2    <head></head>
3    <body>
4      <div id="app">
5        <h1>Develop. Preview.
6  Ship. 🚀</h1>
7      </div>
8      <script type="text/
9  javascript">...</script>
10   </body>
11 </html>
```

SOURCE CODE (HTML)

index.html

```
1  <html>
2    <head></head>
3    <body>
4      <div id="app"></div>
5      <script type="text/
6  javascript">...</script>
7    </body>
8  </html>
9
10
11
```

# Sometimes Dangerous ...



# Exercise: DOM Manipulation

- Create an index.html file with an empty  that has `id="app"`.
- Create a card (with a title, a text, and a button).
- Append the card to the #app div.
- When clicking the button, change the text inside the card.

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>DOM Exercise</title>
</head>
<body>

    <div id="app"></div>

    <script>
        const app = document.getElementById("...");

        const card = document.createElement("...");
        const title = document.createElement("...");
        const text = document.createElement("...");
        const button = document.createElement("...");

        title.textContent = "...";
        text.textContent = "...";
        button.textContent = "...";

        card.appendChild(...);
        card.appendChild(...);
        card.appendChild(...);
        app.appendChild(...);

        button.addEventListener("...", () => {
            text.textContent = "...";
        });

    </script>

</body>
</html>
```

# React

# Building Blocks of a Web Application

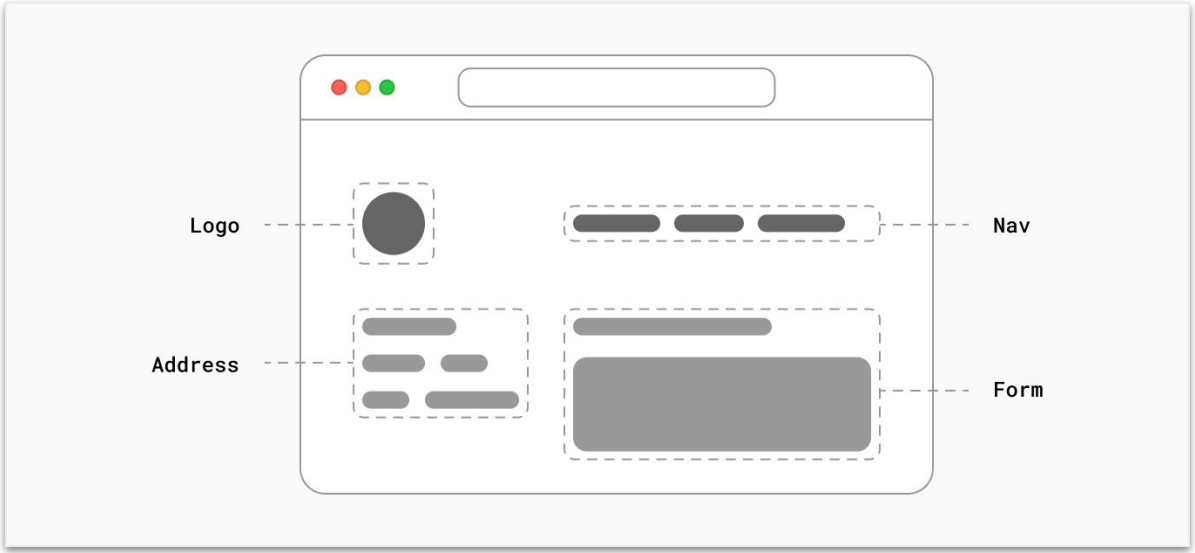
There are a few things you need to consider when building modern applications. Such as:

- **User Interfaces:** Streamlined interfaces for optimal user interaction.
- **Routing:** Efficient user flow between application segments.
- **Data Fetching:** Strategic data sourcing and retrieval methods.
- **Content Delivery:** Dynamic and static content rendering techniques.
- **Deployment Strategy:** Use of Serverless, CDN, and Edge for code execution.
- **Speed Optimization:** Enhancements for peak application performance.

For each part of your application, you will need to decide whether you will build a solution yourself or use other tools such as libraries and frameworks.

# What is React?

React is a JavaScript library for building interactive user interfaces.



# What is React?

React is a declarative, efficient, and flexible JavaScript library for building user interfaces.

- **Declarative UI:** Simplifies code readability and maintenance.
- **Component-Based:** Encapsulates code into reusable components.
- **Virtual DOM:** Optimizes rendering for performance gains.
- **Unidirectional Data Flow:** Ensures stability via controlled data.
- **Ecosystem Flexibility:** Integrates with various libraries and tools.

# Imperative Programming (tell what to do)

Updating the DOM with plain JavaScript is very powerful but verbose. You've written all this code to add an <h1> element with some text.

With this approach, developers spend a lot of time writing instructions to tell the computer **how it should do things**.

```
const app = document.getElementById('app');
const header = document.createElement('h1');
const text = 'Develop. Preview. Ship.';
const headerContent = document.createTextNode(text);
header.appendChild(headerContent);
app.appendChild(header);

const button = document.createElement('button');
const buttonText = 'Click me';
const buttonContent = document.createTextNode(buttonText);
button.appendChild(buttonContent);
app.appendChild(button);
```

# Declarative Programming (tell what you want)

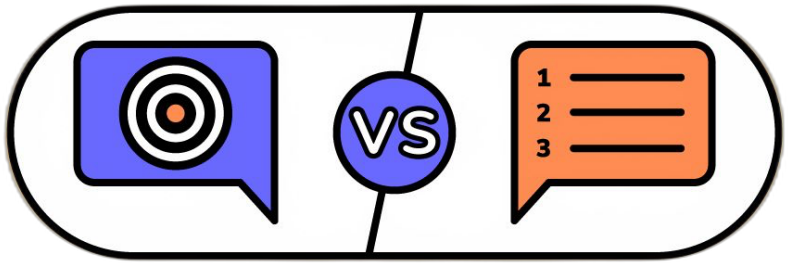
But wouldn't it be nice to describe what you want to show and let the computer figure out how to update the DOM?

```
<Header text="Develop. Preview. Ship. 🚀" />
```

# Declarative vs Imperative Programming

In other words, imperative programming is like giving a chef step-by-step instructions on how to make a pizza.

Declarative programming is like ordering a pizza without being concerned about the steps it takes to make the pizza. 🍕



# Exercise: Basic Example

Copy the code and run the example on your computer.

🔗https://github.com/verwec/02_React_Example

```
<!DOCTYPE html>
<html>
  <head>
    <title>React App</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="https://unpkg.com/react@18/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
    <script type="text/babel">
      function App() {
        return (
          <div>
            <h1>Develop. Preview. Ship.</h1>
          </div>
        );
      }

      ReactDOM.render(<App />, document.getElementById('app'));
    </script>
  </body>
</html>
```

# React Library Split

React and ReactDOM are separate to modularize client and renderer.

- **React Core**: Defines components and manages application logic.
- **ReactDOM Bindings**: Connects React to the DOM for web rendering.

✔️**Platform Agnostic:** Core concepts are universal, not tied to web or native.
✔️**Cleaner Updates:** Independent versioning and updates for each library.

# Babel Transformation

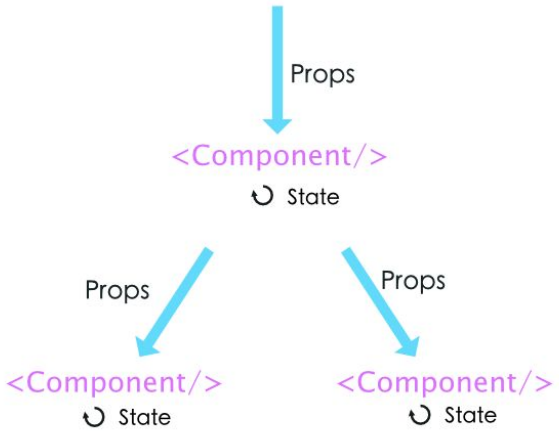
Babel transpiles modern JavaScript and JSX to browser-compatible code.

- **JSX to JavaScript:** Converts JSX syntax into plain JavaScript objects.
- **ES6+ Support:** Translates modern ES6+ code to ES5 for broader compatibility.
- **Browser Compatibility:** Ensures code runs on older web browsers.
- **Development Efficiency:** Allows developers to use the latest JS features.
- **Plugin Ecosystem:** Offers plugins for advanced code transformations and features.

# React Core Concepts

There are three core concepts of React that you'll need to be familiar with to start building React applications. These are:

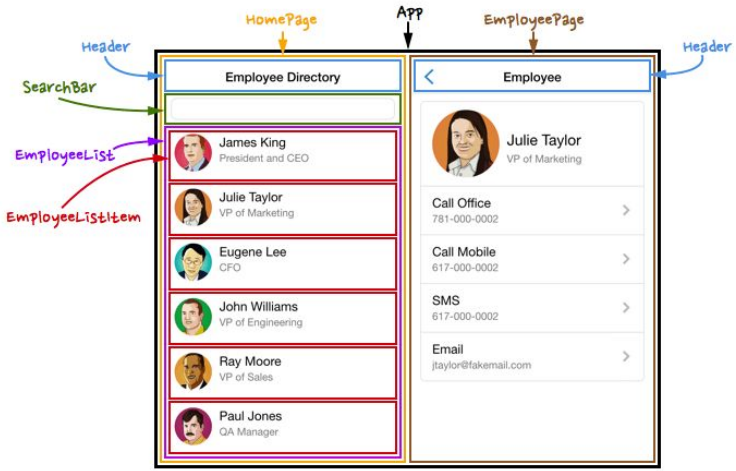
- Components
- Props
- State



# Components

# Components Hierarchy

Break up your user interface (design) into separate elements.



# Nesting Components

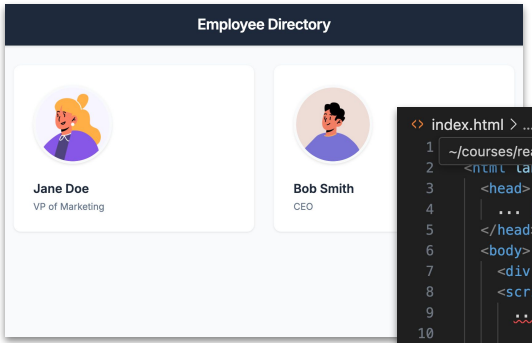
Applications usually include more content than a single component. You can nest React components inside each other like you would regular HTML elements.

```jsx
function App() {
  return (
    <div>
      <header>
        <h1>Employee Directory</h1>
      </header>
      <main>
        <EmployeeCard />
        <OtherEmployeeCard />
      </main>
    </div>
  );
}
```

```jsx
function EmployeeCard() {
  return (
    <div className="employee-card">
      <img
        src="https://avatar.iran.liara.run/public/girl"
        alt="Placeholder beard image"
      />
      <strong>Jane Doe</strong>
      <p>VP of Marketing</p>
    </div>
  );
}
```

# Exercise: Simple React “Employee Directory”

Create the component App and implement all nested Components.




index.html > ...

~/courses/react/03_COMPONENTS/index.html • 1 problem in this file

```
<html lang="en">
  <head>
    ... ⇥ tab
  </head>
  <body>
    <div id="root"></div>
    <script type="text/babel">
      ...

      function App() {
        return (
          <div>
            <header>
              <h1>Employee Directory</h1>
            </header>
            <main>
              <EmployeeCard />
              <OtherEmployeeCard />
            </main>
          </div>
        );
      }

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

```
function EmployeeCard() {
  return (
    <div className="employee-card">
      <img
        src="https://avatar.iran.liar
        alt="Placeholder beard image"
      />
      <strong>Jane Doe</strong>
      <p>VP of Marketing</p>
    </div>
  );
}
```

# Solution: Simple React “Employee Directory”

```
<script type="text/babel">
  function EmployeeCard() {
    return (
      <div className="employee-card">
        <img
          src="https://avatar.iran.liara.run/public/girl"
          alt="Placeholder beard image"
        />
        <strong>Jane Doe</strong>
        <p>VP of Marketing</p>
      </div>
    );
  }

  function OtherEmployeeCard() {
    return (
      <div className="employee-card">
        <img
          src="https://avatar.iran.liara.run/public/boy"
          alt="Placeholder beard image"
        />
        <strong>Bob Smith</strong>
        <p>CEO</p>
      </div>
    );
  }

  function App() {
    return (
      <div>
        <header>
          <h1>Employee Directory</h1>
        </header>
        <main>
          <EmployeeCard />
          <OtherEmployeeCard />
        </main>
      </div>
    );
  }

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

# Components & Props

# Displaying Data without Props

Lot of code duplication :-(

```
function EmployeeList() {
  return (
    <>
      <h1>Employee List</h1>
      <div>
        <Employee />
        <hr />
        <Employee2 />
      </div>
    </>
  );
}

function Employee() {
  return (
    <div>
      <div>Name: Bob</div>
      <div>Age: 25</div>
      <div>Salary: $100,000</div>
    </div>
  );
}

function Employee2() {
  return (
    <div>
      <div>Name: Lisa</div>
      <div>Age: 31</div>
      <div>Salary: $80,000</div>
    </div>
  );
}
```

# Exercise: Remove duplication

Remove the duplication with props.

```
function EmployeeList() {
  return (
    <>
      <h1>Employee List</h1>
      <div>
        <Employee name="Bob" age="25" salary="100,000" />
        <hr />
        <Employee name="Lisa" age="31" salary="80,000" />
      </div>
    </>
  );
}

function Employee({name, age, salary}) {
  return (
    <div>
      <div>Name: {name}</div>
      <div>Age: {age}</div>
      <div>Salary: {salary}</div>
    </div>
  );
}
```

# JSX Rules

# The Rules of JSX

- Return a single root element.
- To return multiple elements from a component, wrap them with a single parent tag.

```jsx
<>
  <h1>Employee List</h1>
  <div>
    <Employee name="Bob" age="25" salary="100,000" />
    <hr />
    <Employee name="Lisa" age="31" salary="80,000" />
  </div>
</>
```

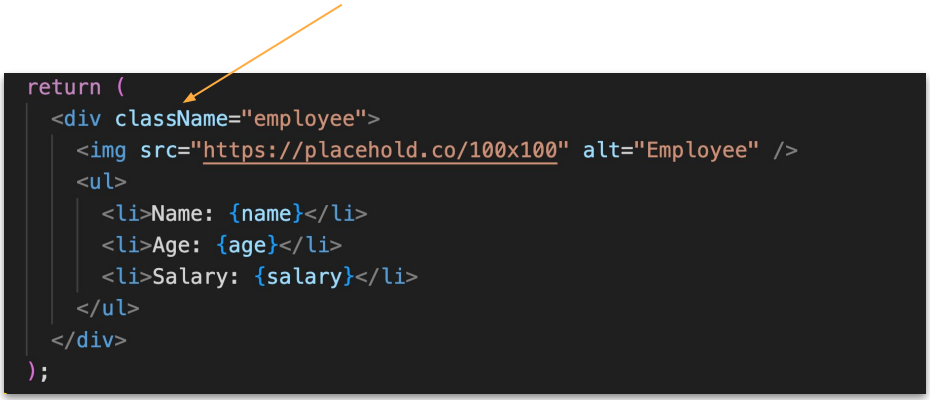
# The Rules of JSX

JSX requires tags to be explicitly closed: self-closing tags like `<img>` must become `<img />`, and wrapping tags like `<li>`oranges must be written as `<li>oranges</li>`.

```jsx
<>
  <img src="https://placehold.co/100x100" alt="Employee" />
  <ul>
    <li>Name: {name}</li>
    <li>Age: {age}</li>
    <li>Salary: {salary}</li>
  </ul>
</>
```

# The Rules of JSX

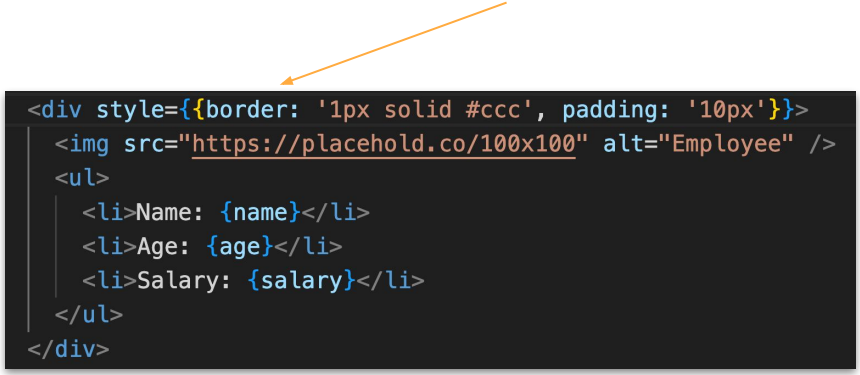
- React uses camelCase for HTML/SVG attributes (e.g. `stroke-width` → `strokeWidth`).
- `class` is reserved, so React uses `className` instead.



```
return (
  <div className="employee">
    <img src="https://placehold.co/100x100" alt="Employee" />
    <ul>
      <li>Name: {name}</li>
      <li>Age: {age}</li>
      <li>Salary: {salary}</li>
    </ul>
  </div>
);
```

# Using “double curlies“

- In addition to strings, numbers, and other JavaScript expressions, you can even pass objects in JSX.
- Objects are also denoted with curly braces. Therefore, to pass a JS object in JSX, you must wrap the object in another pair of curly braces



```
<div style={{border: '1px solid #ccc', padding: '10px'}}>
  <img src="https://placehold.co/100x100" alt="Employee" />
  <ul>
    <li>Name: {name}</li>
    <li>Age: {age}</li>
    <li>Salary: {salary}</li>
  </ul>
</div>
```

# create-react-app

# Demo: Create React App

# Exercise: Create React App

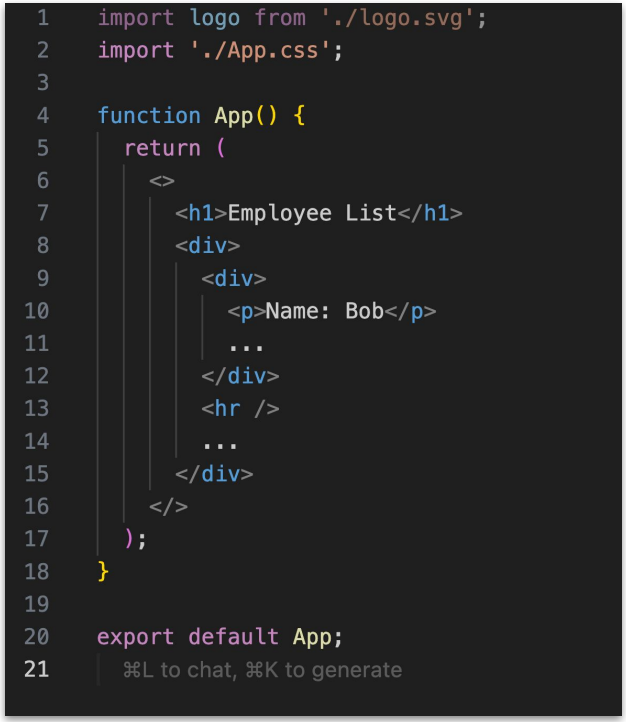
Install react globally

- `npm install -g create-react-app`

Create a new React app

- `npx create-react-app my-app`
- `cd my-app`
- `npm start`

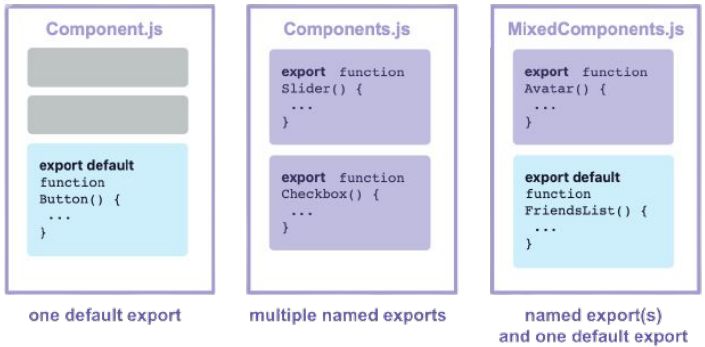
Show A List of 2-3 Employees in `App.js`



# In- & Exporting

Makes the code more modular and reusable in other files. Done in three steps:

1. Make a new JS file to put the components in.
2. Export your function component from that file (using either default or named exports).
3. Import it in the file where you'll use the component (using the corresponding technique for importing default or named exports).



# In- & Exporting

```
1   import React from 'react';
2
3   function Employee() {
4     return <div>Employee</div>;
5   }
6
7   export default Employee;
```

```
3
4   import Employee from './Employee';
5
```

# In- & Exporting

my-app > src > JS Employee.js > default

```
1   import React from 'react';
2
3   function Employee({ name, age, salary }) {
4     return (
5       <div className="employee-card">
6         <h2>{name}</h2>
7         <p>Age: {age}</p>
8         <p>Salary: ${salary}</p>
9       </div>
10    );
11  }
12
13  function OfficeDog({ breed, age }) {
14    return (
15      <div className="employee-card">
16        <h2>{breed}</h2>
17        <p>Age: {age}</p>
18      </div>
19    );
20  }
21
22  export { OfficeDog };
23
24  export default Employee;
```

# Props

# Destructuring vs. Direct Prop Access

```jsx
function OfficeDog({ breed, age }) {
  return (
    <div className="employee-card">
      <h2>{breed}</h2>
      <p>Age: {age}</p>
    </div>
  );
}

function OfficeCat(props) {
  let breed = props.breed;
  let age = props.age;

  return (
    <div className="employee-card">
      <h2>{breed}</h2>
      <p>Age: {age}</p>
    </div>
  );
}
```

Highly recommend!

# Props Default Values

```
function OfficeDog({ breed, age, isVaccinated=true}) {
  return (
    <div className="employee-card">
      <h2>{breed}</h2>
      <p>Age: {age}</p>
      <p>Vaccinated: {isVaccinated ? "Yes" : "No"}</p>
    </div>
  );
}
```

# Props Forwarding

```
function AnimalOfTheWeek(props) {
  return (
    <>
      <h1> Animal of the Week</h1>
      <OfficeDog {...props} />
    </>
  );
}

function OfficeDog({ breed, age, isVaccinated=true}) {
  return (
    <div className="animal-card">
      <h2>{breed}</h2>
      <p>Age: {age}</p>
      <p>Vaccinated: {isVaccinated ? "Yes" : "No"}</p>
    </div>
  );
}
```

# Passing JSX as children

```jsx
function Card({children}) {
  return (
    <div className="p-6 max-w-sm mx-auto rounded-xl shadow-lg">
      {children}
    </div>
  );
}

function AnimalOfTheWeek(props) {
  return (
    <Card>
      <h1 className="text-2xl font-bold mb-2"> Animal of the Week</h1>
      <OfficeDog {...props} />
    </Card>
  );
}
```

# Exercise: Refactoring & Styling

Remove the code duplication from your App.js.

- Use `Employee` components
- Integrate `<script src="https://cdn.tailwindcss.com"></script>` in your index.html
- Checkout https://tailwindcss.com/docs/styling-with-utility-classes
- Improve Styling by using the `{children}` syntax
  (e.g. a wrapping Card Component)

**John Doe**
Age: 30
Salary: $100000

**Jane Doe**
Age: 25
Salary: $80000

**Jim Doe**
Age: 35
Salary: $120000

# Conditional Rendering

# Conditional Rendering

Simple Solution:

App.js

Download Reset Fork

```
function Item({ name, isPacked }) {
  if (isPacked) {
    return <li className="item">{name} ✔</li>;
  }
  return <li className="item">{name}</li>;
}

export default function PackingList() {
  return (
    <section>
      <h1>Sally Ride's Packing List</h1>
      <ul>
        <Item
          isPacked={true}
          name="Space suit"
        />
```

Show more

**Sally Ride's Packing List**

- Space suit ✔
- Helmet with a golden leaf ✔
- Photo of Tam

# Conditional Rendering

JavaScript has a compact syntax for writing a conditional expression — the conditional operator or “ternary operator”.

```
return (
  <li className="item">
    {isPacked ? name + ' ✔' : name}
  </li>
);
```

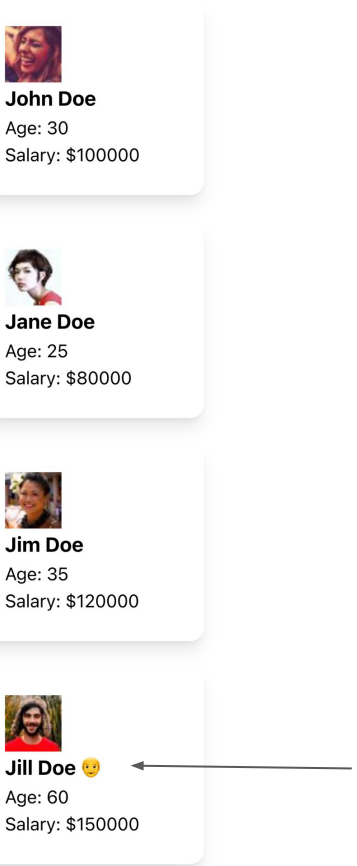
# Logical AND operator (&&)

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.

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

# Exercise: Conditional Rendering

Add a ‘Grandpa` Emoji for all Employees that are older 50 years 👴



# Rendering Lists

# Rendering Lists

You will often want to display multiple similar components from a collection of data.

1. **Map Functionality:** Render array items as components with map().
2. **Filter Usage:** Display only certain items using filter().
3. **React Keys:** Assign unique keys for stable identities.
4. **Component Arrays:** Transform data to component arrays for UI.
5. **Selective Rendering:** Control which elements to render and when.

# Rendering data from arrays

App.js

Download Reset Fork

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

export default function List() {
  const listItems = people.map(person =>
    <li>{person}</li>
  );
  return <ul>{listItems}</ul>;
}
```

- Creola Katherine Johnson: mathematician
- Mario José Molina-Pasquel Henríquez: chemist
- Mohammad Abdus Salam: physicist
- Percy Lavon Julian: chemist
- Subrahmanyan Chandrasekhar: astrophysicist

Console (1)

```
Warning: Each child in a list should have a unique "key" prop.

Check the render method of `List`. See https://reactjs.org/link/warning-keys for more information.
    at li
    at List
```

# Filtering arrays of items

App.js data.js utils.js

Reset Fork

```
1  import { people } from './data.js';
2  import { getImageUrl } from './utils.js';
3
4  export default function List() {
5    const chemists = people.filter(person =>
6      person.profession === 'chemist'
7    );
8    const listItems = chemists.map(person =>
9      <li>
10       <img
11         src={getImageUrl(person)}
12         alt={person.name}
13       />
14       <p>
15         <b>{person.name}:</b>
16         {' ' + person.profession + ' '}
17         known for {person.accomplishment}
18       </p>
19     </li>
20   );
21   return <ul>{listItems}</ul>;
22 }
23
```

**Mario José Molina-Pasquel Henríquez:** chemist known for discovery of Arctic ozone hole

**Percy Lavon Julian:** chemist known for pioneering cortisone drugs, steroids and birth control pills

Console (1)

```
Warning: Each child in a list should have a unique "key" prop.

Check the render method of `List`. See https://reactjs.org/link/warning-keys
for more information.
    at li
    at List
```

Show less

# Keeping list items in order with key

1. You need to give each array item a key — a string or a number that uniquely identifies it among other items in that array. A well-chosen key helps React infer what exactly has happened, and make the correct updates to the DOM tree.

```
<li key={person.name}>...</li>
```

# Exercise 1: List Rendering

- Replace the hardcoded employee components with a map function based on the list below.
- Create an Employee component for each item in the array.

```
const employees = [
  { name: "John Doe", age: 30, salary: 100000 },
  { name: "Jane Doe", age: 25, salary: 80000 },
  { name: "Jim Doe", age: 35, salary: 120000 },
  { name: "Jill Doe", age: 60, salary: 150000 }
];
```

# Exercise 2: Group the list (filter)

- Add a new property `active`
- Group by `active` and `Inactive` (using `filter`)

**Active Employees**

**John Doe**
Age: 30
Salary: $100000

**Jane Doe**
Age: 25
Salary: $80000

**Inactive Employees**

**Jim Doe**
Age: 35
Salary: $120000

**Jill Doe** 🙄
Age: 60
Salary: $150000

# State

# State: A Component's Memory

App.js data.js Reset Fork

```
import { sculptureList } from './data.js';

export default function Gallery() {
  let index = 0;

  function handleClick() {
    index = index + 1;
  }

  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleClick}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
        by {sculpture.artist}
      </h2>
      <h3>
        ({index + 1} of {sculptureList.length})
      </h3>
      <img
        src={sculpture.url}
        alt={sculpture.alt}
      />
      <p>
        {sculpture.description}
      </p>
    </>
  );
}
```

Next

***Homenaje a la Neurocirugía* by Marta Colvin Andrade**

(1 of 12)

Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.

Show less

Here's a component that renders a sculpture image. Clicking the "Next" button should show the next sculpture by changing the index to 1, then 2, and so on. However, this won't work.

# State: A Component's Memory

```
const employees = [
  { name: "John Doe", age: 30, salary: 100000, active: true },
  { name: "Jane Doe", age: 25, salary: 80000, active: true },
  { name: "Jim Doe", age: 35, salary: 120000, active: false },
  { name: "Jill Doe", age: 60, salary: 150000, active: false }
];

function App() {

  let index = 0;

  function handleClick() {
    console.log('index', index);
    index++;
  }

  const employee = employees[index];

  return (
    <div className="container p-6 max-w-7xl">
      <div className="mb-12" onClick={handleClick}>
        <h2 className="text-2xl font-bold mb-6">Active Employees</h2>
        <Employee
          name="John Doe"
          age={employee.age}
          salary={employee.salary}
          active={employee.active} />
      </div>
    </div>
  );
}
```

# State: A Component's Memory

The *handleClick* event handler is updating a local variable, index. But two things prevent that change from being visible:

1. **Local variables don’t persist between renders**. When React renders this component a second time, it renders it from scratch—it doesn’t consider any changes to the local variables.
2. **Changes to local variables won’t trigger renders.** React doesn’t realize it needs to render the component again with the new data.

To update a component with new data, two things need to happen:

1. Retain the data between renders.
2. Trigger React to render the component with new data (re-rendering).

# State: A Component's Memory

App.js data.js Reset Fork

```
import { useState } from 'react';
import { sculptureList } from './data.js';

export default function Gallery() {
  const [index, setIndex] = useState(0);

  function handleClick() {
    setIndex(index + 1);
  }

  let sculpture = sculptureList[index];
  return (
    <>
      <button onClick={handleClick}>
        Next
      </button>
      <h2>
        <i>{sculpture.name} </i>
        by {sculpture.artist}
      </h2>
      <h3>
        ({index + 1} of {sculptureList.length})
      </h3>
      <img
        src={sculpture.url}
        alt={sculpture.alt}
      />
      <p>
        {sculpture.description}
      </p>
    </>
  );
}
```

Next

***Homenaje a la Neurocirugía* by Marta Colvin Andrade**

(1 of 12)

Although Colvin is predominantly known for abstract themes that allude to pre-Hispanic symbols, this gigantic sculpture, an homage to neurosurgery, is one of her most recognizable public art pieces.

Show less

# State: Usage

Call *useState* at the top level of your component to declare one or more state variables.

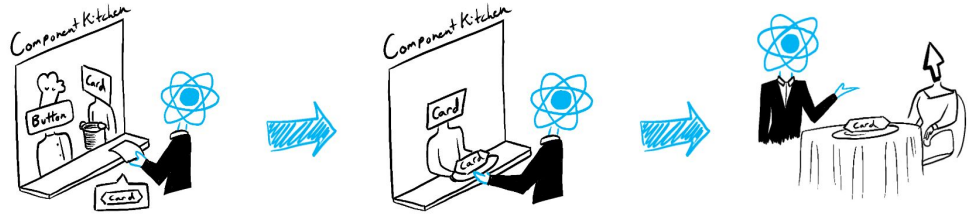
*useState* returns an array with exactly two items:

1. The current state of this state variable, initially set to the initial state you provided.
2. The set function that lets you change it to any other value in response to interaction.

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

function MyComponent() {
  const [age, setAge] = useState(42);
  const [name, setName] = useState('Taylor');
  // ...
```

# Render & Commit

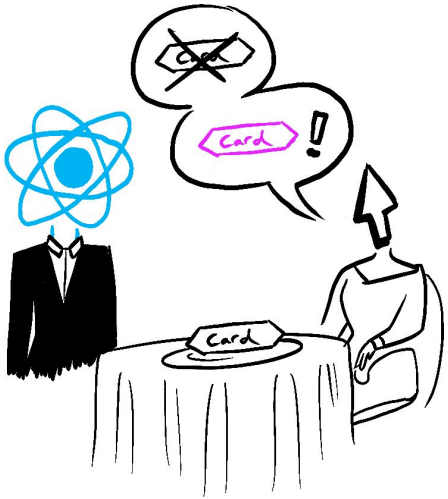
Imagine that your components are cooks in the kitchen, assembling tasty dishes from ingredients. In this scenario, React is the waiter who puts in requests from customers and brings them their orders. This process of requesting and serving UI has three steps:

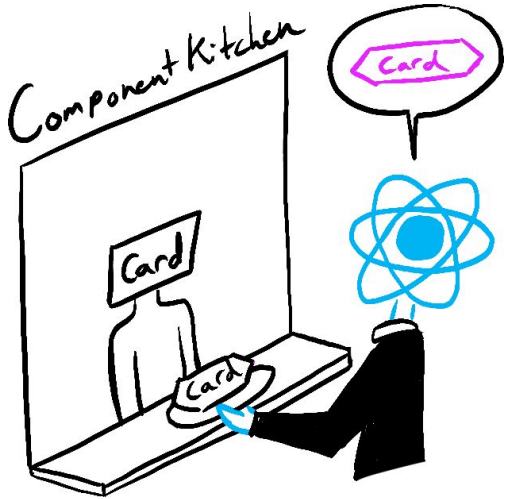


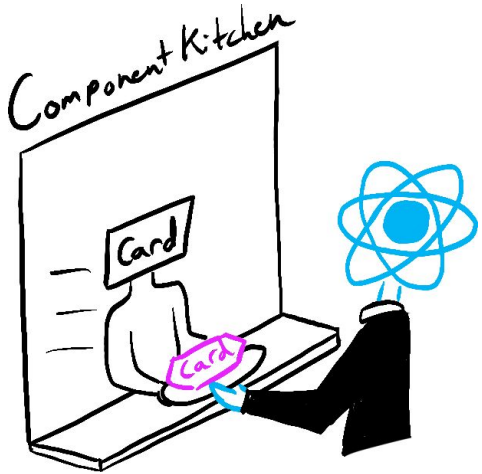

Trigger Render Commit

# Re-renders when state updates

Updating your component's state automatically queues a render. (You can imagine these as a restaurant guest ordering tea, dessert, and all sorts of things after putting in their first order, depending on the state of their thirst or hunger.)

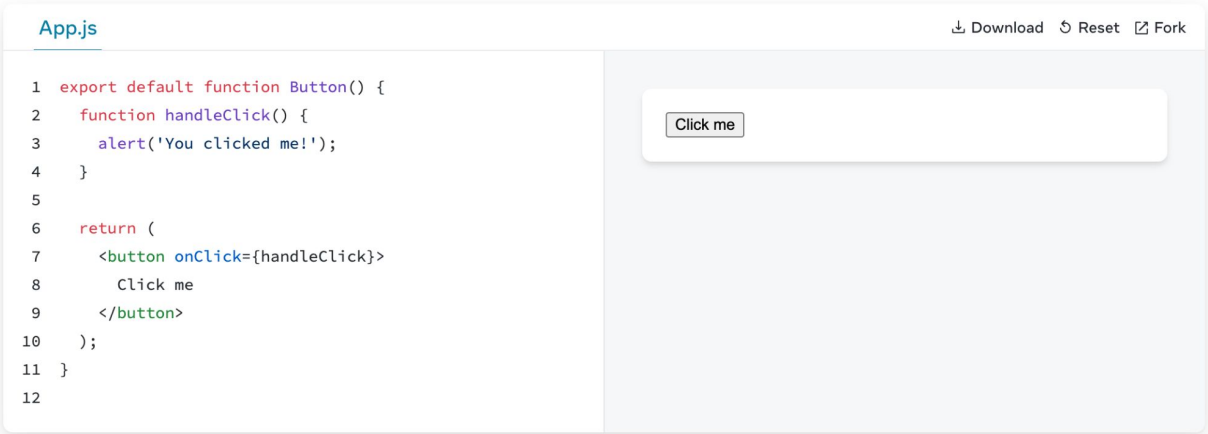


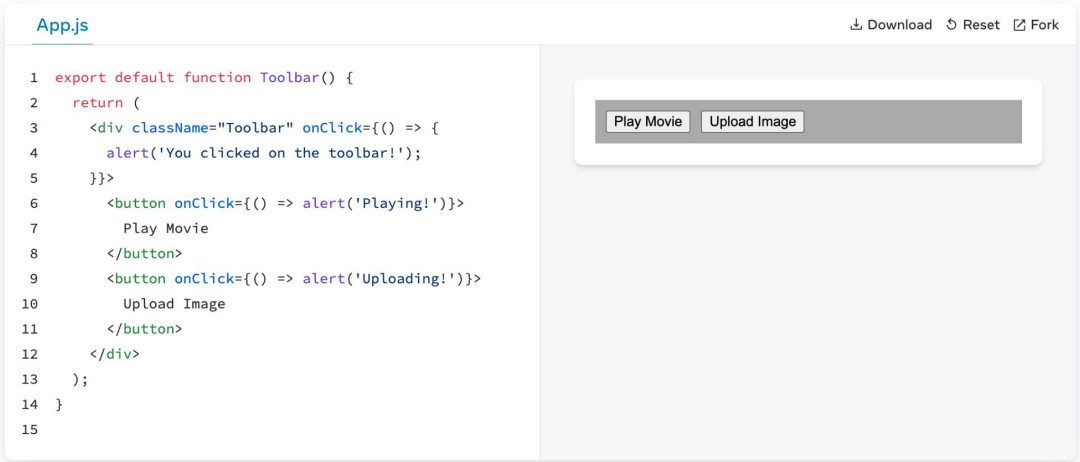
# Responding to events

React lets you add event handlers to your JSX. Event handlers are your own functions that will be triggered in response to user interactions like clicking, hovering, focusing on form inputs, and so on.



# Event propagation

Event handlers will also catch events from any children your component might have. We say that an event “bubbles” or “propagates” up the tree: it starts with where the event happened, and then goes up the tree.

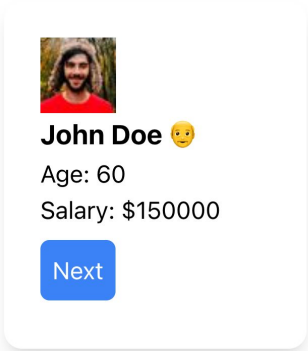


# Event propagation

```
function Card({children}) {

  function handleClick() {
    console.log('clicked');
  }

  return (
    <div className="p-6 mt-6 w-[200px] mx-auto rounded-xl shadow-lg"  onClick={handleClick} >
      {children}
    </div>
  );
}


function Employee({ name, age, salary }) {

  return (
    <Card>
      <img src={`https://i.pravatar.cc/50?img=${age}`} alt="Avatar" />
      <h2 className="text-lg font-bold">
        {name} {age > 50 && '👴'}
      </h2>
      <p>Age: {age}</p>
      <p>Salary: ${salary}</p>
      <button className="bg-blue-500 text-white p-2 my-2 rounded-md"> Click me </button>
    </Card>
  );
}
```

# Exercise: A Employees Gallery

Create a gallery where you can switch between the Employees by clicking Next



# Exercise: A small Blog Site

1. Let’s extract components together
2. Implementation of the components