Introduction to React

Jorge Miguel Ferreira da Silva

Researcher at IEETA

Learning Objectives

By the end of this session, you will:

- Understand React's core concepts and architecture
- Be able to create and manage React components
- Understand state management and hooks
- Be familiar with modern React features

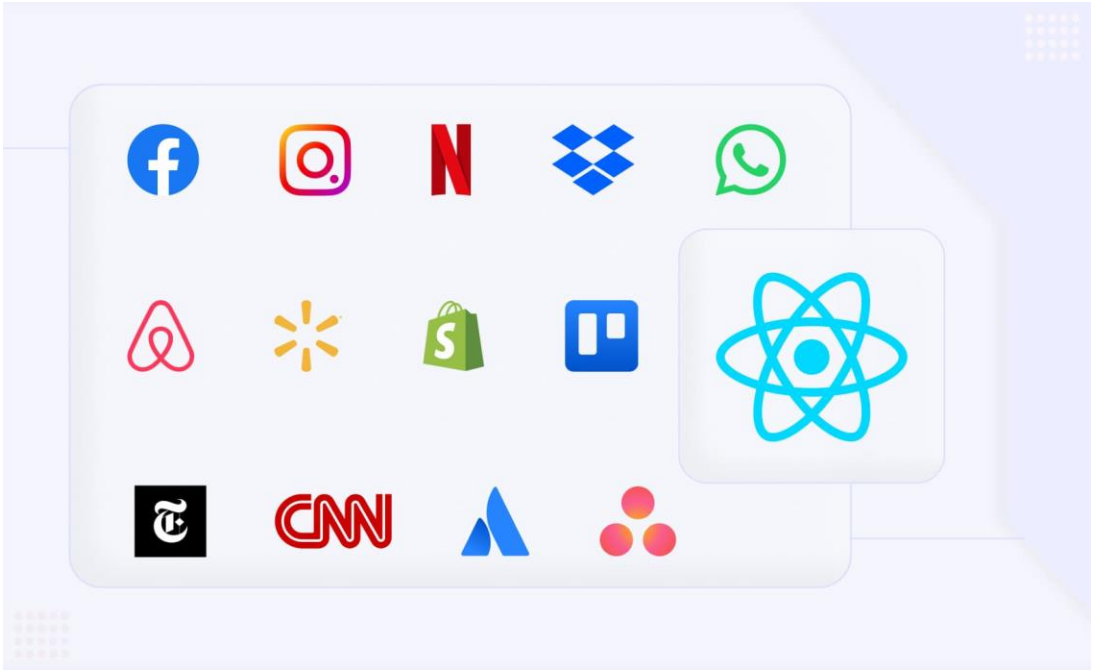
Introduction

Definition of React

- React is a JavaScript library for building user interfaces
- Developed and maintained by Facebook
- Released in 2013
- Uses a component-based architecture
- Follows a declarative programming approach

Examples of Popular React Applications

- **Facebook** - The original React application
- **Instagram** - Photo sharing platform
- **Netflix** - Streaming service
- **Airbnb** - Vacation rental platform
- **Discord** - Communication platform
- **WhatsApp Web** - Messaging application

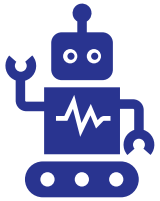


React in the Context of Software Engineering

Embodies key software engineering principles:

- **Modularity**: Components encapsulate functionality.
- **Reusability**: Components can be reused across projects.
- **Maintainability**: Easier to update and manage code.
- **Scalability**: Supports large and complex applications.

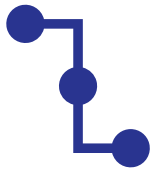
Comparing React with Other Frameworks



React vs. Angular:

React: Library focused on UI components

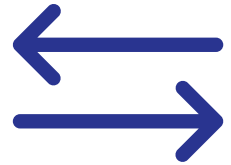
Angular: Full-fledged MVC (Model-View-Controller) framework



React vs. Vue.js:

Vue.js: Combines ideas from React and Angular

React: Larger community and ecosystem

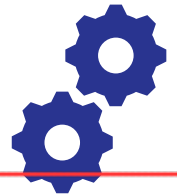


When to choose React:

Need for flexibility and control

Preference for a strong ecosystem

When to Use React

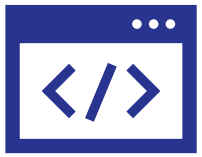


Ideal Scenarios:

Building dynamic and interactive user interfaces

Applications requiring high performance

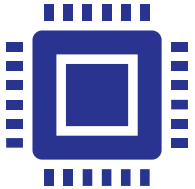
Projects that benefit from reusable components



Considerations:

Team familiarity with JavaScript and JSX

Long-term maintenance and scalability



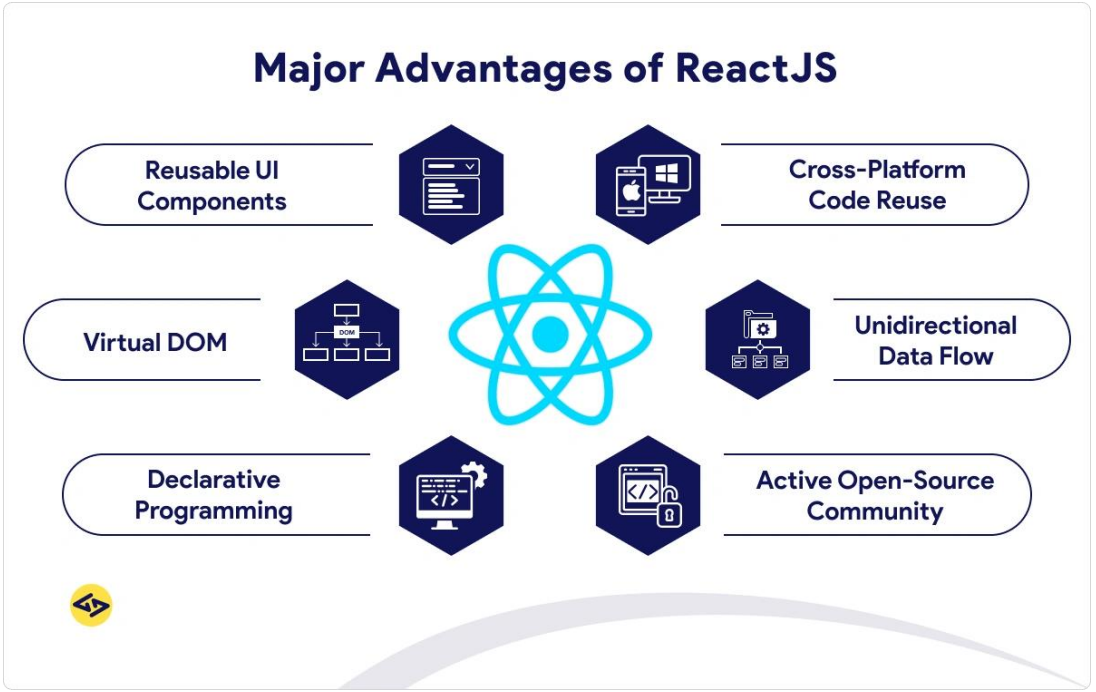
When React may not be the best choice:

Simple static websites

Projects requiring minimal JavaScript

Key Features and Benefits of React

- Component-based architecture
- Virtual DOM for fast rendering
- Reusable components
- Unidirectional data flow
- Rich ecosystem and community
- Strong community support and extensive documentation



Problems React Solves

Challenges in building complex user interfaces:

- Frequent updates and re-rendering
- Managing state across the application
- Performance issues with direct DOM manipulation

How React addresses these challenges:

- Efficient updates with the Virtual DOM
- Simplified state management
- Component-based architecture for reusability

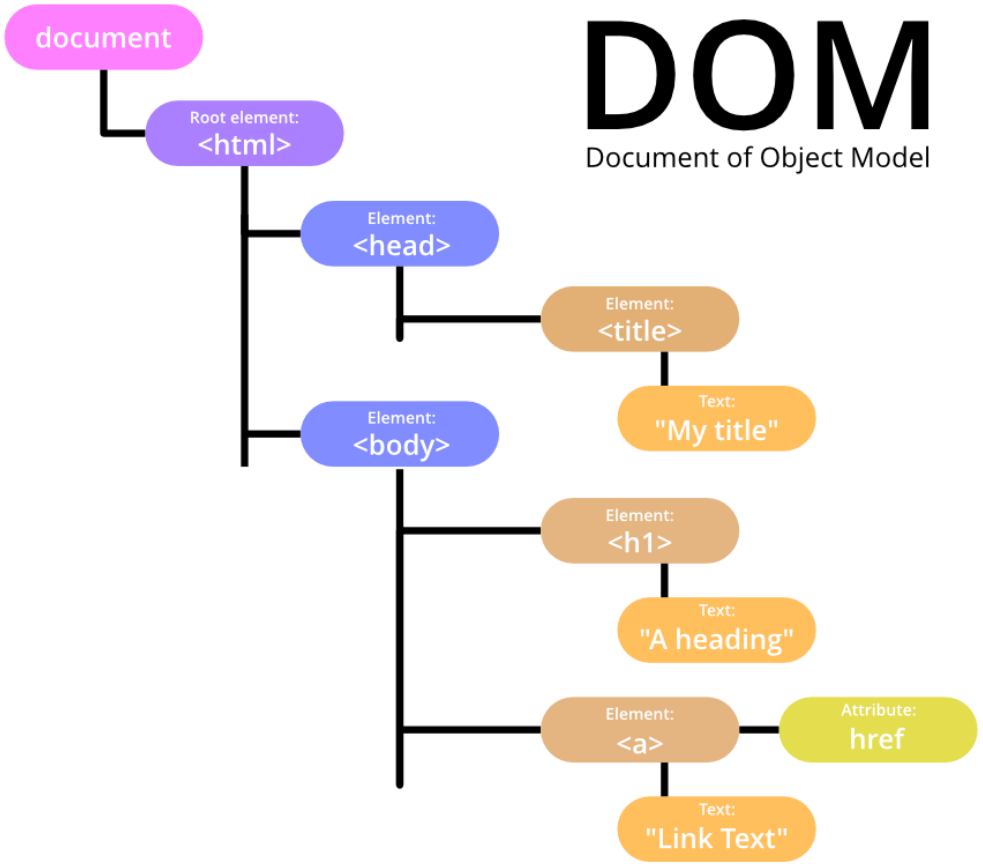
Core Concepts

React's Virtual DOM Explained

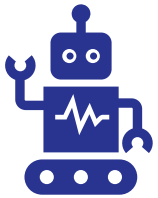
The **DOM** (Document Object Model) is a programming interface for web documents. It represents the structure of a web page in a way that programs can interact with. When a web page loads, the browser creates a **DOM** of the page, which is like a tree structure of all the elements on the page (like , <p>, <img>, etc.).

How the DOM Works

- The DOM represents the **HTML document as a tree of nodes**. Each element, attribute, and piece of text in the HTML becomes a "node" in this tree.
- JavaScript can interact with the DOM to **read** or **manipulate** the structure, style, or content of a web page.
- For example, using JavaScript, you can change text, add or remove elements, or update styles on the page without reloading it.



React's Virtual DOM Explained



What is the Virtual DOM?

An in-memory representation of the real DOM

Updates are first applied to the Virtual DOM

How React uses the Virtual DOM:

Calculates the difference (diffing) between versions

Efficiently updates only the necessary parts of the real DOM

Benefits:

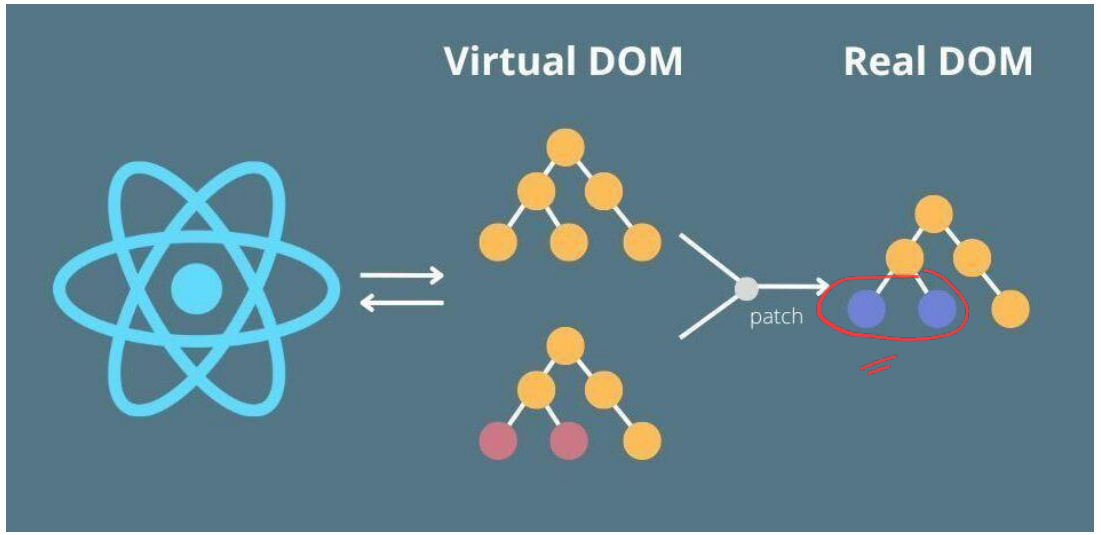
Improved performance

Smooth user experience

Visualizing the Virtual DOM

Understanding the update process:

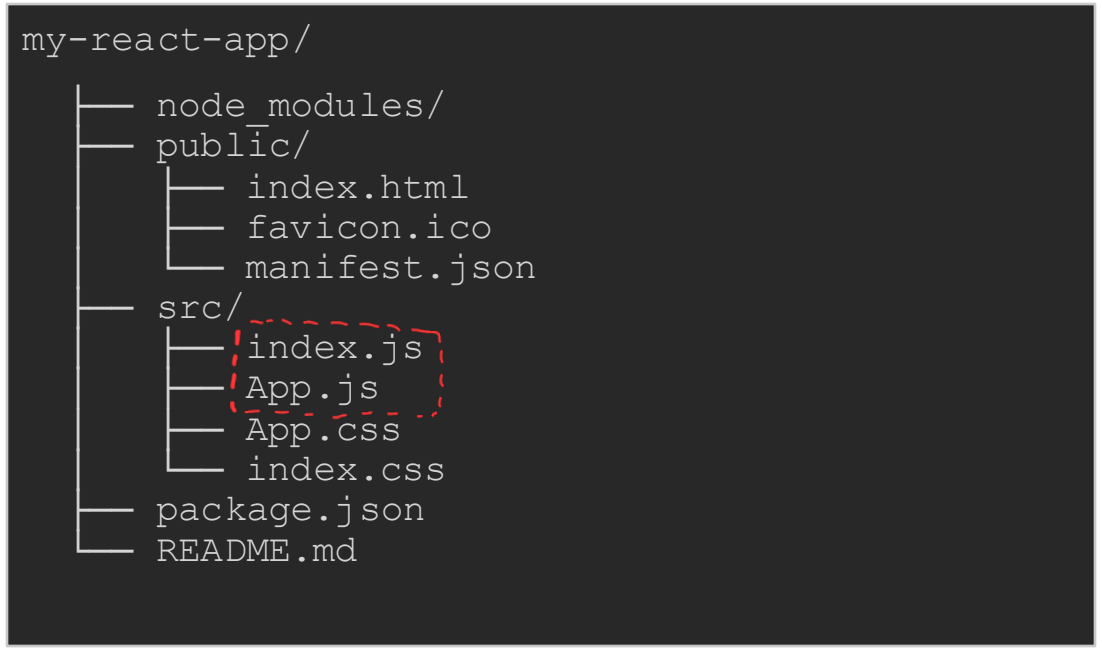
- Initial rendering creates a Virtual DOM tree. in memory!
- State changes produce a new Virtual DOM.
- React diffs the new tree with the previous one.
- Only the changed elements are updated in the real DOM.



Overview of Default Folder Structure

Key directories and files:

- **src/** - Source code files
- **public/** - Static assets
- **node_modules/** - Dependencies
- **package.json** - Project configuration
- Important files in src/:
  - **index.js** - Entry point
  - **App.js** - Root component



Key Files in React - index.js

index.js

- **Entry Point**: This file is the main entry point of the React application, where the app is first initialized.
- **React**: The core library for building the component structure of the app.
- **ReactDOM**: Used to render React components to the DOM.
- The **App component** is imported as the **root component**, representing the main part of the application.
- **ReactDOM.createRoot**: Creates a root object, which represents where the React app will render in the HTML document.
- **root.render()**: This method is called to render the App component inside

raramente se altera...

```
// index.js
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
const root =
ReactDOM.createRoot(document.getElementById(
'root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```

Key Files in React - App.js

- **Main Component**: App.js serves as the main component of the React app.
- **Defines UI Structure**: Sets up the layout and overall structure of the interface.
- **State and Effects**: Can manage state and side effects as needed.
- **Container for Components**: Holds and organizes other components that make up the application.

```
// App.js
import React from 'react';
import './App.css';
function App() {
  return (
    <div className="App">
      <h1>Welcome to the React App </h1>
      {/* Other components can be added here */}
    </div>
  );
}
export default App;
```

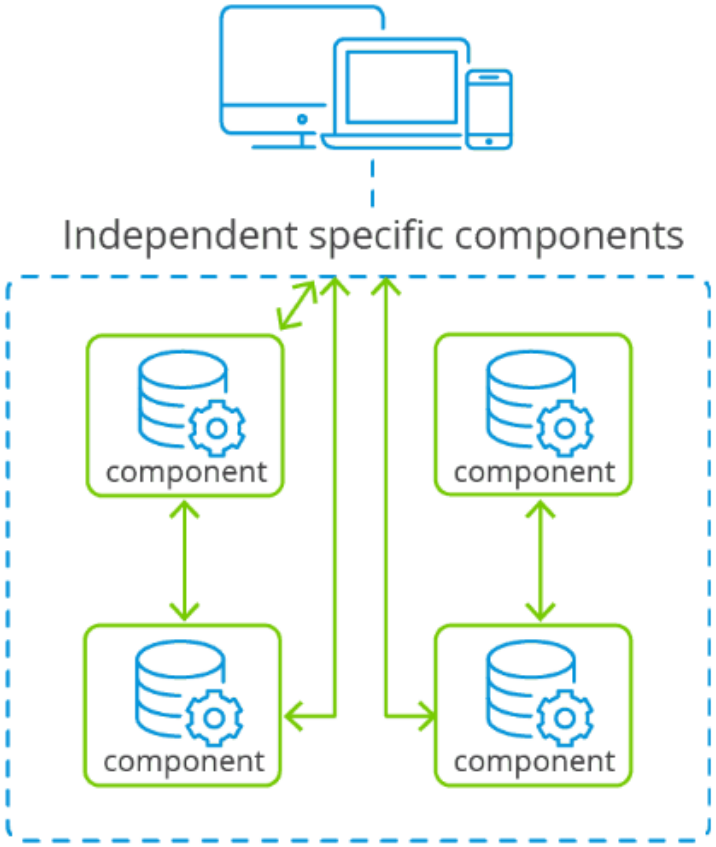
Key Files in React - public/index.html

- **HTML Template File**: This file provides the basic HTML structure for the React application.
- **Mount Point**: Contains a element, which serves as the mounting point where the React app is injected.
- **Root Div**: is the container where the React application is rendered by the JavaScript injected from index.js.
- **JavaScript Injection Point**: After building the app, the compiled JavaScript is injected here by the build tool (e.g., Webpack).

```
<!-- public/index.html -->
<!DOCTYPE html>
<html lang="pt">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-
width, initial-scale=1.0" />
  <title>React App</title>
</head>
<body>
  <noscript>You need to enable JavaScript to run
this app.</noscript>
  <div id="root"></div>
  <!- Compilled JavaScript will be injected here --
>
</body>
</html>
```

Benefits of Component-Based Architecture

- Promotes **independence** and **autonomy** of components.
- Enhances **code reusability** and **consistency**
- Facilitates **parallel development** among teams.
- Simplifies **testing** and **maintenance**.



JSX and Components

Introduction to JSX and its Syntax

JSX Fundamentals:

- HTML-like syntax in JavaScript
- Must return a single parent element
- Use camelCase for attributes
- Use curly braces for JavaScript expressions
- Self-closing tags must end with />

```
// JSX Example
function Welcome() {
  const name = 'User';
  return (
    <div className="welcome-container">
      <h1>Hello, {name}!</h1>
      <img src={userPhoto} alt="Profile" />
      {isLoggedIn && <button onClick={handleClick}>
        Logout
      </button>}
    </div>
  );
}
```

How JSX is Transformed to JavaScript

Babel's Role:

- Converts JSX to React.createElement() calls
- Transforms modern JavaScript features
- Ensures browser compatibility
- Handles JSX expressions and attributes

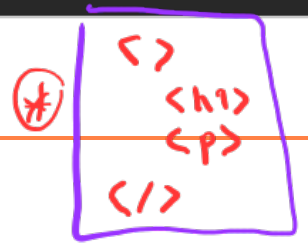
```
// JSX Code
const element = (
  <h1 className="greeting">
    Hello, World!
  </h1>
);
// Transformed by Babel into:
const element = React.createElement(
  'h1',
  { className: 'greeting' },
  'Hello, World!'
);
```

Writing JSX: Rules and Syntax

Essential JSX Rules:

- Single root element requirement
  - Each JSX should have only one father element
- Use camelCase for attributes
  - E.g. onClick, className
- Close all tags (including self-closing)
- Use className instead of class
- Escape quotes in attributes
- Handle boolean attributes must be explicitly defined
  - (e.g., readOnly={true}).

```
function ExampleComponent() {
  return (
    <div className="container">  {/* Use className,
not class */}
      <input type="text" readOnly={true} />
      <img src={imageUrl} alt="description" />
      <button onClick={handleClick}>
        Click me
      </button>
      {/* Invalid JSX: Multiple root elements */}
      {/* <h1>Title</h1>
      <p>Content</p> */}
    </div>
  );
}
```



Embedding JavaScript Expressions in JSX

Expression Types in JSX:

- Variables and props
  - {name}, {props.value}
- Function calls
  - {myFunction()}.
- Ternary operators
  - {condition ? True' : 'False'}}
- Array methods
  - {list.map(item => <Item key={item.id} />)}
- Object properties
  - {object.property}
- Template literal
  - `Hi, ${name}!`.

```
function UserGreeting({ user, isAdmin }) {
  const greeting = `Welcome back, ${user.name}`;
  return (
    <div>
      {/* Variable */}
      <h1>{greeting}</h1>
      {/* Ternary operator */}
      {isAdmin ? (
        <AdminPanel />
      ) : (
        <UserPanel />
      )}
      {/* Array method */}
      <ul>
        {user.roles.map(role => (
          <li key={role.id}>{role.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

Using Arrow Functions to Define Components

Arrow Function Patterns:

- For simple components, you can use an **implicit return** by omitting curly braces {} and the return keyword.
- When a component needs more logic (e.g., calculations, variable definitions) before returning the JSX, you use an **explicit return** with {} and return.
- Arrow functions allow you to **destructure props** directly in the function parameters. This means you can access props like text or title without writing props.text or props.title
- You can set **default values** for props directly in the function parameters.
- **Concise Syntax**: Shortens code, especially for simple components, making it easier to read and maintain.

```
// Implicit return (single line)
const Button = ({ text }) => <button>{text}</button>;

// Explicit return (multi-line)
const Card = ({ title, content }) => {
  return (
    <div className="card">
      <h2>{title}</h2>
      <p>{content}</p>
    </div>
  );
};

// With default props
const Header = ({ title = 'Default Title' }) => (
  <header>
    <h1>{title}</h1>
  </header>
);
```

implicit

Understanding React Fragments

- React Fragments allow you to return multiple elements without adding extra nodes to the DOM.
- Syntax:
  - <React.Fragment></React.Fragment> or <> </>
- Use cases:
  - When a component needs to return multiple sibling elements.
  - Avoiding unnecessary div wrappers.

or React.Fragment

```
// Using React Fragments
function Table() {
  return (
    <>
      <thead>
        <tr>
          <th>Name</th>
          <th>Age</th>
        </tr>
      </thead>
      <tbody>
        {/* Table rows */}
      </tbody>
    </>
  );
}
```

Examples of JSX Syntax

Common JSX Patterns:

- **Component Composition**: Combine multiple components to create complex interfaces.
- **Conditional Rendering**: Show or hide elements based on conditions.
- **List Rendering**: Use methods like map to render arrays of elements.
- **Event Handling**: Add events like onClick, onChange.
- **Styling**: Apply styles with the style attribute or CSS classes.
- **Dynamic Class Naming**: Build class names based on state.

```
function App() {

  const items = ['Apple', 'Banana', 'Orange'];
  const [isActive, saetIsActive] = useState(false);
  return (
    <div className={`app ${isActive ? 'active' :
''}`}>
      {/* Conditional rendering */}
      {isActive && <Alert message="Active!" />}
      {/* Event handling */}
      <button onClick={() => setIsActive(!isActive)}>
        Toggle Active
      </button>
      {/* List rendering */}
      <ul style={{ padding: '20px' }}>
        {items.map((item, index) => (
          <li key={index}>{item}</li>
        ))}
      </ul>
    </div>
  );
}
```

list

Components Deep Dive

Definition of Functional Components

Functional Component Basics:

- **JavaScript** functions returning **JSX**
- Can accept **props** as **parameters**
- Support hooks for **state** and **effects**
- Can be **arrow functions** or **regular functions**
- Must start with **capital letter**

```
// Regular function component
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}
// Arrow function component
const Profile = ({ user }) => {
  return (
    <div>
      <img src={user.avatar} alt="Profile" />
      <h2>{user.name}</h2>
      <p>{user.bio}</p>
    </div>
  );
};
```

Basic Example of a React Component

Import Statements: Import React and any required modules or styles.

Function Definition: Define the component as a function, optionally receiving props.

JSX Return: Return JSX that outlines the UI structure.

Props Destructuring: Destructure props to access specific values, with optional default values.

Export Statement: Export the component to make it usable in other files.

```jsx
// Welcome.jsx
import React from 'react';
import './Welcome.css';
function Welcome({ name, greeting = 'Hello' }) {
  return (
    <div className="welcome-container">
      <h1>{greeting}, {name}!</h1>
      <p>Welcome to our application.</p>
    </div>
  );
}
export default Welcome;
// Usage
<Welcome name="John" greeting="Hi" />
```

How to Create a Functional Component

Component Creation Methods:

- Function declaration
  - function ComponentName(props) { ... }.
- Arrow function expression
  - const ComponentName = (props) => { ... }
- Named export vs default export
  - export default MyComponent
  - MyComponent = () => { ... }; — needs keys to import

Key Requirements:

- Must return JSX or null
- Must start with capital letter
- Can receive props as parameter

```
// Function declaration
function Button({ text, onClick }) {
  return (
    <button onClick={onClick}>
      {text}
    </button>
  );
}

// Arrow function
const Card = ({ title, children }) => (
  <div className="card">
    <h2>{title}</h2>
    {children}
  </div>
);

// Export options
export default Button;
export { Card };
```

Component Naming Conventions

Naming Guidelines:

- Use PascalCase for component names
- Use descriptive, clear names
- Match component name with file name

File Organization:

- One component per file
- Group related components
- Use index.js for exports

```
// UserProfile.jsx
export function UserProfile() { ... }
// Button/index.jsx
export function Button() { ... }
// Layout/Header/NavigationBar.jsx
export function NavigationBar() { ... }
// Bad naming examples:
// function userProfile() { ... }  // lowercase
// function User_Profile() { ... }  // snake case
// function userprofile() { ... }   // no camelCase
```

Importing Components Using ES6 (ECMAScript 6) import/export

Import/Export Patterns:

- Default exports vs named exports
- Importing multiple components
- Aliasing imports

Best Practices:

- Group related imports
- Order imports consistently
- Use absolute imports when configured

```
// components/Button.jsx
export default function Button() { ... }

// components/Layout.jsx
export function Header() { ... }
export function Footer() { ... }

// App.jsx
import Button from './components/Button';
import { Header, Footer } from './components/Layout';import {
Button as CustomButton } from './components/CustomButton';

// Using index.js for cleaner imports
// components/index.js
export { default as Button } from './Button';export { Header,
Footer } from './Layout';

// App.jsx
import { Button, Header, Footer } from './components';
```

Advantages of Functional Components

Key Benefits:

- Simpler syntax and readability
- Better performance optimization
- Easier testing and debugging

Modern Features:

- Hooks for state management
- Effect hooks for lifecycle events
- Custom hooks for logic reuse

```
// Example of a functional component with hooks
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);
  useEffect(() => {
    async function fetchUser() {
      const data = await getUser(userId);
      setUser(data);
      setLoading(false);
    }
    fetchUser();
  }, [userId]);
  if (loading) return <div>Loading...</div>;
  return (
    <div>
      <h1>{user.name}</h1>
      <p>{user.email}</p>
    </div>
  );
}
```

When to Use Functional Components

Best Use Cases:

- Presentational components
- Simple UI elements
- Reusable component libraries

Ideal Scenarios:

- Components focused on UI rendering
- Components with minimal state management
- Components that primarily handle props

```
// Simple presentational component
function UserCard({ user }) {
  return (
    <div className="user-card">
      <img src={user.avatar} alt={user.name}
/>
      <h3>{user.name}</h3>
      <p>{user.role}</p>
      {user.isOnline && <span
className="online-badge" />}
    </div>
  );
}
// Usage
<UserCard user={{
  name: 'John Doe',
  role: 'Developer',
  avatar: '/avatar.jpg',
  isOnline: true
}} />
```

Reusing Components to Build Complex UIs

Component Composition Patterns:

- Nesting components
- Component props for customization
- Children prop for flexibility
- Container/Presenter pattern

Benefits:

- Maintainable code
- Consistent UI
- Reduced duplication

```
// Building a complex UI from smaller
components
function Dashboard() {
  return (
    <Layout>
      <Header>
        <Navigation items={menuItems} />
        <UserProfile user={currentUser} />
      </Header>
      <Sidebar>
        <Menu items={sidebarItems} />
      </Sidebar>
      <MainContent>
        <DataGrid data={userData} />
        <Charts data={analyticsData} />
      </MainContent>
    </Layout>
  );
}
```

Props and State

What are Props in JSX?

- **Props** (short for "properties") are a way to pass data from one component to another in React.

- They are similar to arguments for functions
  - Helping parent components share information with child components.

```
// Code in Parent Component
function App() {
  return (
    <div>
      <Greeting name="Jorge" />
      <Greeting name="Miguel" />
    </div>
  );
}

// Code in Child Component

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

Output:
Hello, Jorge!
Hello, Miguel!
```

Understanding Props and Their Role in React

Props Fundamentals:

- Read-only data passed to components
- Flow from parent to child
- Can be any JavaScript value
- Immutable within the child component

Common Props Types:

- Primitive values (strings, numbers)
- Objects and arrays
- Functions (event handlers)
- JSX elements

```
// Code in Parent Component
function App() {
  return (
    <div>
      <Greeting name="Jorge" />
      <Greeting name="Miguel" />
    </div>
  );
}

// Code in Child Component

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

Output:
Hello, Jorge!
Hello, Miguel!
```

Passing Props to Child Components

- Prop Passing Patterns:
  - **Individual Prop Passing:** Specify each prop manually.
  - **Spread Operator:** Pass all properties of an object at once.
  - **Default Props:** Set default values for props.
  - **Prop Types Validation:** Use TypeScript or PropTypes.
- Best Practices:
  - Keep prop names descriptive.
  - Avoid passing unnecessary props.
  - Document required props.
  - Use TypeScript for type safety.

```
// Individual prop passing
function Greeting({ name, age }) {
 return <p>Hello, {name}. You are {age} years
old.</p>;
}

// Usage
<Greeting name="Alice" age={25} />

//Spread Operator
const person = { name: "Bob", age: 30 };

function Greeting({ name, age }) {
return <p>Hello, {name}. You are {age} years
old.</p>;
}

// Usage
<Greeting {...person} />
```

Using Spread Operator for Props

Spread Operator:

- Pass all object properties as props.
- Useful for passing many props at once.

Example:

- Define an object with all the props.
- Use `{...object}` syntax.

```
// Using spread operator
function App() {
  const userProps = {
    name: 'Jane',
    email: 'jane@example.com',
    role: 'admin'
  };
  return (
    <UserProfile {...userProps} />
  );
}
// Child Component
function UserProfile({ name, email, role }) {
  return (
    <div>
      <h2>{name} ({role})</h2>
      <p>{email}</p>
    </div>
  );
}
```

Prop Types Default

- Set default values for props within the component.

Default values

```
function Greeting({ name, age }) {
return <p>Hello, {name}. You are {age} years old.</p>;
}

Greeting.defaultProps = {
name: "Anonymous",
age: 18
};

// Usage
<Greeting /> // Renders "Hello, Anonymous. You are 18 years old."
```

Prop Types Validation

- Default Props:
  - Set default values for props within the component.
- Prop Types Validation:
  - Use TypeScript or PropTypes to enforce prop types.
- Benefits:
  - Improves code reliability.
  - Documents expected prop types.

```
// Default props example
function UserProfile({ name, email, role = 'user'
}) {
  return (
    <div>
      <h2>{name} ({role})</h2>
      <p>{email}</p>
    </div>
  );
}
// Using PropTypes (if not using TypeScript)
UserProfile.propTypes = {
  name: PropTypes.string.isRequired,
  email: PropTypes.string.isRequired,
  role: PropTypes.string
};
```

Using Children Props for Nested Components

Children Prop Features:

- **Renders Nested JSX Elements:** The children prop allows you to pass any JSX elements inside a component, making it possible to nest elements for flexible layouts.
- **Enables Component Composition:** With children, components can be combined to create complex UIs. It helps in breaking down UI into smaller, reusable pieces.
- **Supports Multiple Children:** The children prop can handle multiple elements, allowing you to pass a list of components or HTML elements to be rendered within a parent component.

Common Use Cases:

- Layout components.
- Wrapper components.

Children Props Layout and Card Use Cases

Imagine you have a Layout component with a header and footer, but the main content varies.

You can pass that varying content as children.

```
function Layout({ children }) {
return (
<div>
    <header>Header</header>
    <main>{children}</main>
    <footer>Footer</footer>
</div>
);
}

// Usage
<Layout>
    <h1>Welcome to My Website</h1>
    <p>Here's the main content.</p>
</Layout>
```

If you have a **Card** component that adds **styling** or **structure** to whatever it **contains**, you can use **children** to handle any **content inside**.

The **Card** component will **render** whatever you put **inside** the **<Card>** tags, allowing you to **use** it with different **content each time**.

```
function Card({ children }) {
    return <div
className="card">{children}</div>;
}

// Usage
<Card>
    <h2>Title</h2>
    <p>Some content inside the card.</p>
</Card>
```

Manipulating Children Props

Advanced Use of Children:

- Manipulate or filter children elements.
- Use React.Children utilities.

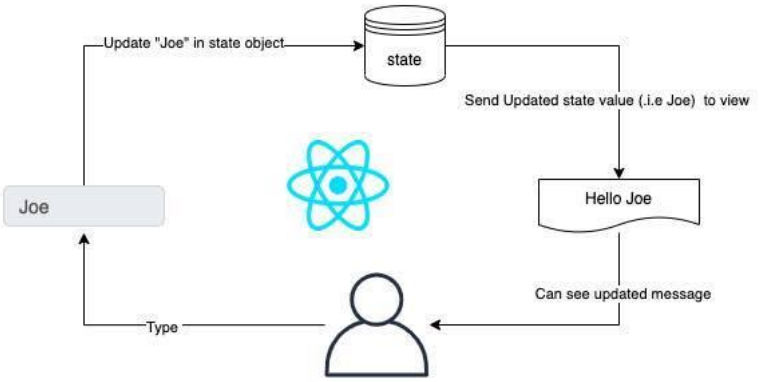
Example:

- Adding classes to specific child elements.

```
// Manipulating children
function ItemList({ children }) {
  return (
    <ul>
      {React.Children.map(children, child => {
        return <li className="item">{child}</li>;
      })}
    </ul>
  );
}
// Using ItemList
function App() {
  return (
    <ItemList>
      <span>Item 1</span>
      <span>Item 2</span>
    </ItemList>
  );
}
```

What is State?

- **State Fundamentals:**
  - Unlike **props**, which are **passed** from **parent** to **child** and are **read-only**, **state** is **mutable** within the **component**.
  - You can **change** the **state** using a **function** and React will **automatically update** the **component** with the **new data**.
  - When the **state** of a **component changes**, React will **re-render** that **component** to **reflect** the **new state**. This is what makes the **UI dynamic**.
  - **State** is usually tied to a **specific component** and is used to **track data** that only that **component cares** about.
  - **State** is typically **initialized** in the **component** where it's used.



What is State?

- **Characteristics:**
  - Each **component** can have its **own state**, which is **separate** from other **components**' state.
  - You can pass the state as props to child components. Useful if the child components need to **display** or **interact** with the **data stored** in the parent's state.
  - **State** should **not** be modified **directly**. Instead, **React** provides **methods** to **update** the **state** (such as **setState** for **class components** and **useState** for **functional components**) to ensure that changes are tracked properly.

Why Use State?

State allows React components to:

- Dynamically respond to user input or other events.
- Update the UI in real-time.
- Track data that changes over time within a component.

In **Example**:

- UseState to declare state Count and function setCount to update the state;
- When setCount is called, react re-renders the component and updates the interface.

```
// Basic state usage
function Counter() {
  const [count, setCount] = useState(0);
  const increment = () => {
    setCount(count + 1);
  };
  return (
    <div>a
      <p>Count: {count}</p>
      <button
onClick={increment}>Increment</button>
    </div>
  );
}
```

Managing State with useState Hook

- **useState Features:**
  - useState is a React Hook that lets you add state to functional components. Hooks are special functions that let you "hook into" React features.
  - Returns state value and updater function. E.g. const [count, setCount] = useState(0);
    - State Value: The current value of the state.
    - Updater Function: A function to update the state value.
  - Supports primitive (strings, numbers) and complex types (objects, arrays).
  - Preserves state between renders.

Managing State with useState Hook

- Best Practices:
  - If your new state depends on the previous state, use the functional form of the updater function to ensure you're working with the most recent state.
    E.g. setCount(prevCount => prevCount + 1);
  - **Keep state minimal and relevant.**
    - Only store data in state that is necessary for rendering. Avoid storing redundant or computed values that can be derived during rendering.
  - **Split state if it becomes too complex.**
    - If a state object becomes too complex (e.g., deeply nested), consider splitting it into multiple state variables for better readability and maintainability

Use Case

- **State Initialization:** name and email are two independent state variables initialized as empty strings.
- **Updating State on Change:** The onChange handlers for each input update their respective state (setName and setEmail) whenever the input value changes.
- **Form Submission:** handleSubmit prevents the default form submission behavior (page reload).

```
// Multiple state variables
function UserForm() {
  const [name, setName] = useState('');
  const [email, setEmail] = useState('');
  const handleSubmit = (e) => {
    e.preventDefault();
    // Process form data
  };
  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        onChange={e => setName(e.target.value)}
        placeholder="Name"
      />
      <input
        type="email"
        value={email}
        onChange={e => setEmail(e.target.value)}
        placeholder="Email"
      />
      <button type="submit">Submit</button>
    </form>
  );
}
```

Complex State Management

- Managing Complex State:
  - Use objects or arrays in state.
  - Do the update of state immutably.
  - Spread operator for copying state.
- Example - Managing a list of items.
  - Each todo object has three properties: id, text, and completed.
  - The addTodo function is called when the "Add Todo" button is clicked.
  - setTodos([...todos, newTodo]) updates the state by creating a new array with all existing todos and the new one at the end.

```
// Complex state example
function TodoList() {
  const [todos, setTodos] = useState([
    { id: 1, text: 'Learn React', completed: false }
  ]);
  const addTodo = (text) => {
    const newTodo = {
      id: todos.length + 1,
      text,
      completed: false
    };
    setTodos([...todos, newTodo]);
  };
  return (
    <div>
      <ul>
        {todos.map(todo => (
          <li key={todo.id}>{todo.text}</li>
        ))}
      </ul>
      <button onClick={() => addTodo('New Task')}>Add
Todo</button>
    </div>
  );
}
```

Hooks and Advanced Concepts

What are Hooks?

- Hook Fundamentals:
  - Functions starting with 'use'.
  - Only call at the top level.
  - Only call from React functions.
- Benefits:
  - Reuse stateful logic.
  - Combine multiple hooks.
  - Improve code organization.
- Example: Create a personal hook ‘'useCurrentTime’ that manages the state of the current time.

```
// Custom hook example
function useCurrentTime() {
  const [time, setTime] = useState(new Date());
  useEffect(() => {
    const timer = setInterval(() => {
      setTime(new Date());
    }, 1000);
    return () => clearInterval(timer);
  }, []);
  return time;
}

// Using the custom hook
function Clock() {
  const time = useCurrentTime();
  return <div>{time.toLocaleTimeString()}</div>;
}
```

no início depois de renderizar

When **setTime** is called, React knows the **state** has changed, so it **re-renders** the component that’s using **useCurrentTime**.

This re-rendering ensures that the **latest time value** is displayed on the screen.

Rules for Using Hooks

Essential Hook Rules:

- Only call hooks at the top level.
- Only call hooks from React functions.
- Do not call hooks inside loops or conditions.

Common Mistakes to Avoid:

- Conditional hook calls.
- Hooks in regular functions.
- Nested hook calls.

```
// Correct usage of hooks
function GoodComponent() {
  const [isActive, setIsActive] = useState(false);
  useEffect(() => {
    // Effect logic here
  }, []);
  if (isActive) {
    return <ActiveComponent />;
  }
  return <InactiveComponent />;
}
```

TOP LEVEL

Commonly Used Hooks

useState:

- Manages state within a component.
- Allows you to create and update data that changes over time, like form inputs or counters.
- Example use: Tracking a count value that changes when a button is clicked.

useEffect:

- Handles side effects in a component.
- Runs code after the component renders, such as fetching data or updating the DOM.
- Example use: Fetching data from an API when the component mounts

Commonly Used Hooks

useContext:

- Accesses React context to share data between components without prop drilling.
- Simplifies passing data like themes, user info, or settings across multiple components.
- Example use: Accessing a theme or user object in deeply nested components.

useRef:

- References DOM elements or stores mutable values without triggering a re-render.
- Often used to directly access or manipulate HTML elements (like focusing an input).
- Example use: Storing a reference to an input element to focus it programmatically.

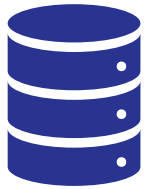
useEffect

- useEffect Patterns:
  - Data fetching.
  - Subscriptions.
  - DOM manipulations.
  - Timer management.
- Key Concepts:
  - Dependency array defines when the Effect is executed.
  - Cleanup functions can ve returned to clean resources.
- Example:
  - 'useEffect is used to fetch data for user whe userId changes.
  - We garantee that only we update the sate if the component is mounted.

```
// Data fetching with useEffect
function UserProfile({ userId }) {
  const [user, setUser] = useState(null);
  useEffect(() => {
    let isMounted = true;
    fetch(`/api/users/${userId}`)
      .then(response => response.json())
      .then(data => {
        if (isMounted) setUser(data);
      });
    return () => {
      isMounted = false;
    };
  }, [userId]);
  if (!user) return <div>Loading...</div>;
  return <div>{user.name}</div>;
}
```

Conclusion

Common Pitfalls and How to Avoid Them



Overusing State:

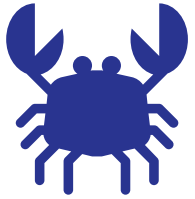
Keep state minimal and relevant

Use derived data when possible

Improper Use of Keys in Lists:

Keys help React identify elements

Use stable and unique keys



Not Following Hooks Rules:

Always call hooks at the top level

Only call hooks from React functions

Summary of Core Concepts

Components:	Building blocks of React applications
JSX:	Syntax extension for JavaScript
State and Props:	State: Component's internal data Props: Data passed from parent to child
Hooks:	Functions that let you use state and other features
Key Takeaways:	Understand the unidirectional data flow Embrace component reusability and composition

Further Resources and Next Steps

Official Documentation:

React Docs: https://reactjs.org/docs/getting-started.html

Recommended Tutorials:

React Tutorial: https://reactjs.org/tutorial/tutorial.html

Thank You!

Questions?

Jorge Miguel Ferreira da Silva

Researcher at IEETA