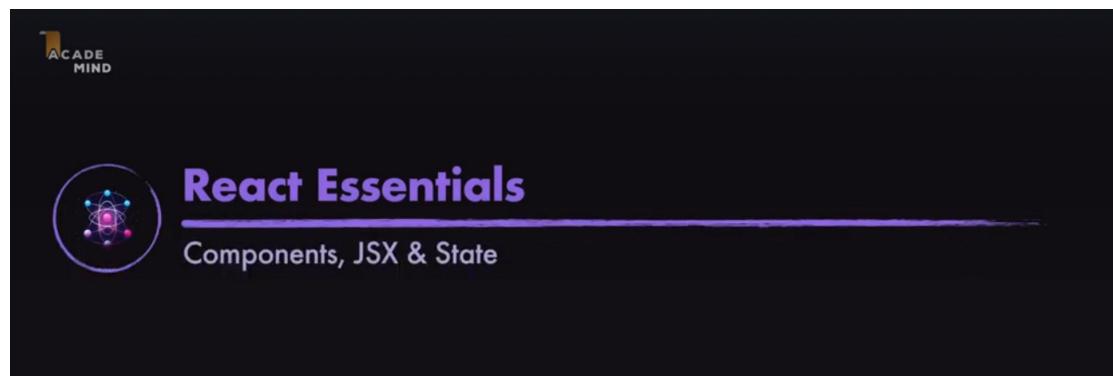


Section 3: React Essentials - Components, JSX, Props, State & More

16 September 2024 21:46

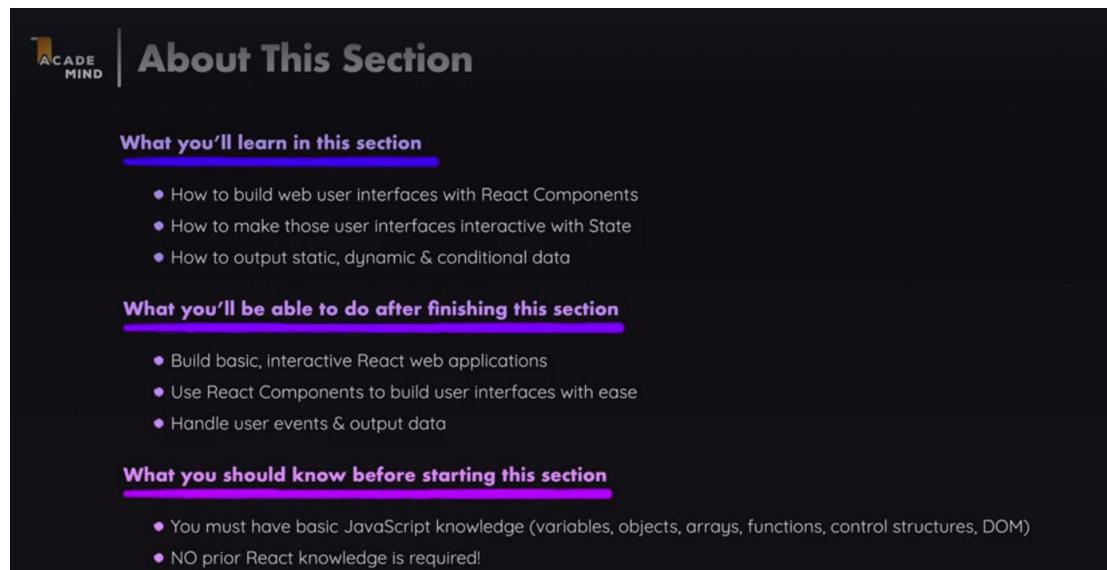


The banner features the Academind logo at the top left. The main title "React Essentials" is in large, bold, blue letters. Below it, the subtitle "Components, JSX & State" is in smaller, white, sans-serif font. To the left of the title is a circular icon containing a stylized atomic or molecular model.



The page has a dark background with the Academind logo at the top left. The title "React Essentials" and subtitle "Components, JSX & State" are identical to the banner. Below the subtitle is a list of four bullet points, each preceded by a black triangle icon:

- ▶ Building User Interfaces with **Components**
- ▶ Using, Sharing & Outputting **Data**
- ▶ Handling User **Events**
- ▶ Building Interactive UIs with **State**

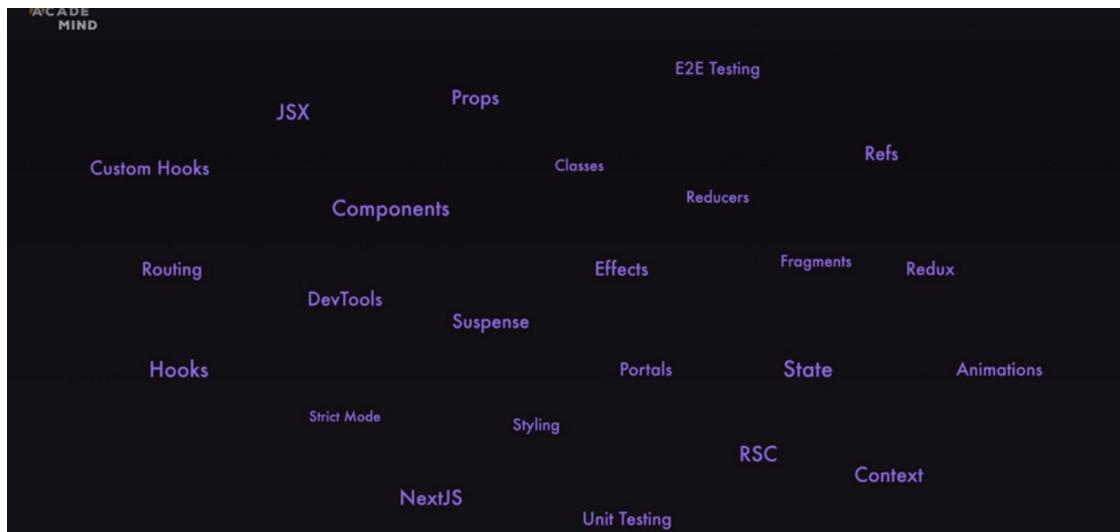


The page has a dark background with the Academind logo at the top left. The main heading "About This Section" is in large, bold, white letters. Below it are three sections with headings in white text:

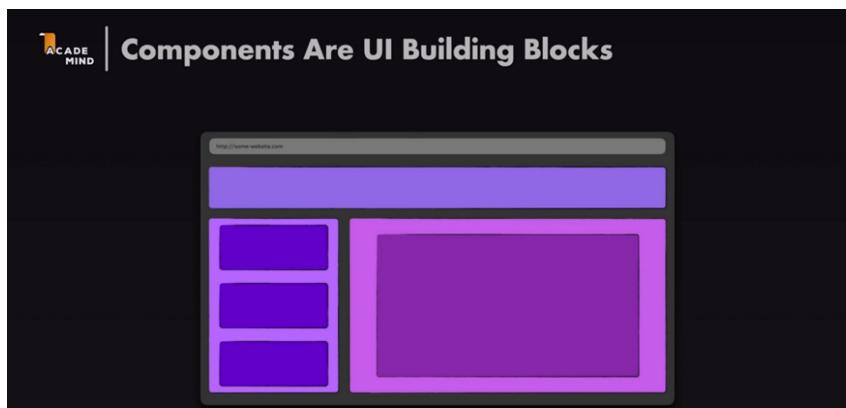
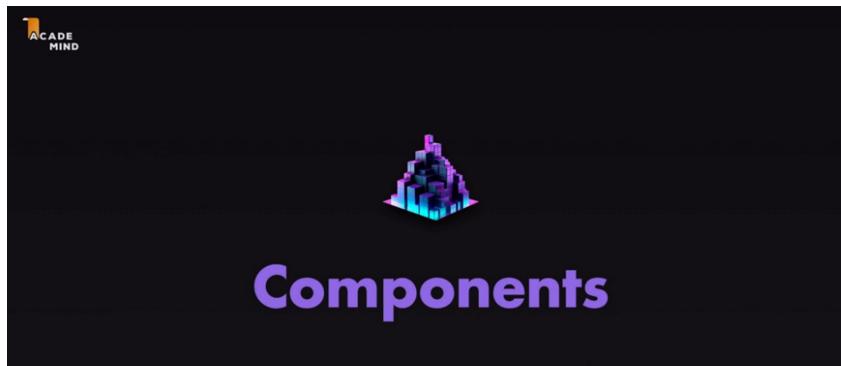
- What you'll learn in this section**
 - How to build web user interfaces with React Components
 - How to make those user interfaces interactive with State
 - How to output static, dynamic & conditional data
- What you'll be able to do after finishing this section**
 - Build basic, interactive React web applications
 - Use React Components to build user interfaces with ease
 - Handle user events & output data
- What you should know before starting this section**
 - You must have basic JavaScript knowledge (variables, objects, arrays, functions, control structures, DOM)
 - NO prior React knowledge is required!

It's All About Component! (Core Concept)

React and its ecosystem provide dozens of useful and important features and concepts, which, of course, are all going to be covered in depth throughout this course. But if you were forced to identify only one core concept which you will need for all React apps,



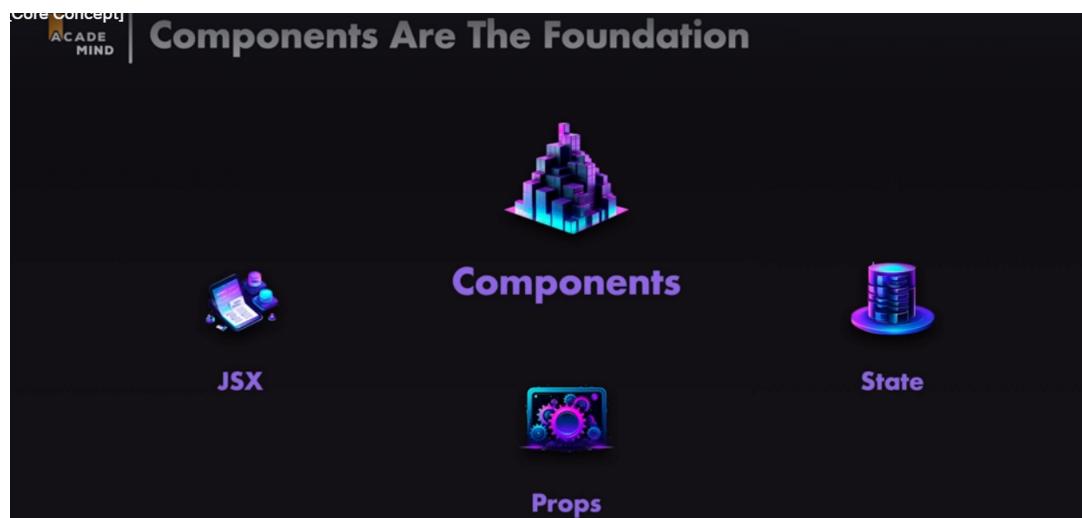
But if you were forced to identify only one core concept which you will need for all React apps, it would be the concept of working with Components.



React Apps Are Built By Combining Components

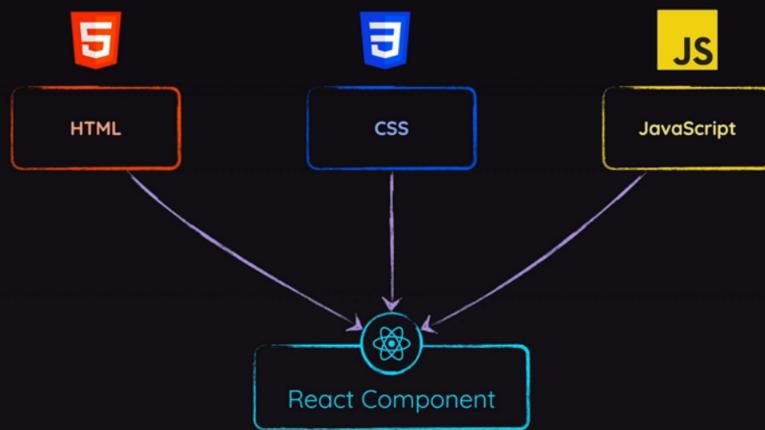
Because Components are potentially reusable building blocks which you, as a React developer, can create, and which you then combine to build the overall user interface.

So React apps are in the end built by combining Components. And whilst a couple of additional features and concepts would be required to create truly interactive user interfaces, Components are the one concept every React app, no matter its complexity, will use.



The diagram shows a "React Essentials" box containing four core concepts: Components, JSX, Props, and State. A "Header" is shown above the box, and "Interactive Tabs" are shown below it. A callout points to the "Props" section with the text: "to define how big or small a component should be." Another callout points to the "Core Concept Items" with the text: "Any website or app can be broken down into smaller building blocks: Components". A third callout points to the "Header" with the text: "It can therefore also be built by creating & combining such components".

Creating Components



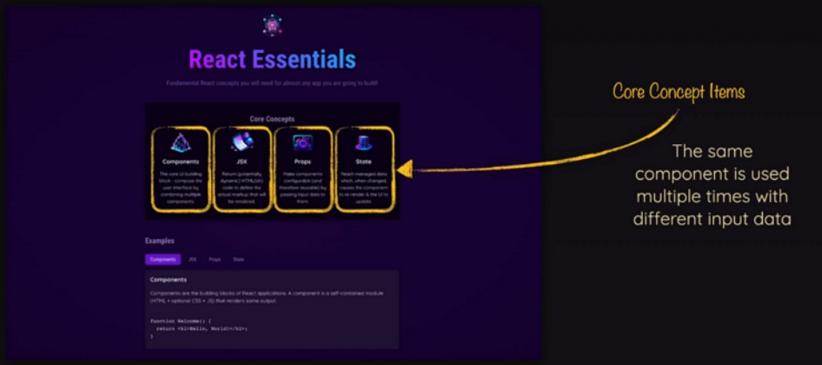
But the idea behind those Components really just is that they wrap HTML and possibly also CSS code, as well as any related JavaScript logic that might be needed. And together, those languages and code pieces then define and control a part of the UI. This allows you, the developer, to split up complex user interfaces into smaller, more manageable parts, which, at least potentially,

may also be reused in different places of the UI.

Components Can Potentially Be Reused



Components Can Potentially Be Reused



Why Components?



Reusable Building Blocks

Create **small building blocks** & **compose** the UI from them

If needed: **Reuse** a building block in different parts of the UI (e.g., a reusable button)



Related Code Lives Together

Related HTML & JS (and possibly CSS) code is **stored together**

Since JS influences the output, storing JS + HTML together makes sense



Separation of Concerns

Different components handle different data & logic

Vastly **simplifies** the process of working on complex apps

Components Are A Fundamental Concept



Describe The Target UI With JSX

JavaScript Syntax eXtension

JSX

```
<div>
  <h1>Hello World!</h1>
  <p>JSX code is awesome!</p>
</div>
```

Used to describe & create HTML elements in JavaScript in a **declarative** way

BUT

Browsers do not support JSX!

React projects come with a **build process** that **transforms** JSX code (behind the scenes) to code that **does work** in browsers

Component Functions Must Follow Two Rules



Name Starts With Uppercase Character

The function name **must start** with an **uppercase** character

Multi-word names should be written in **PascalCase** (e.g., "MyHeader")

It's **recommended** to pick a name that **describes** the UI building block (e.g., "Header" or "MyHeader")



Returns "Renderable" Content

The function must **return** a value that can be **rendered** ("displayed on screen") by React

In **most** cases: Return **JSX**
Also allowed: string, number, boolean, null, array of allowed values

Components are just functions in React

```

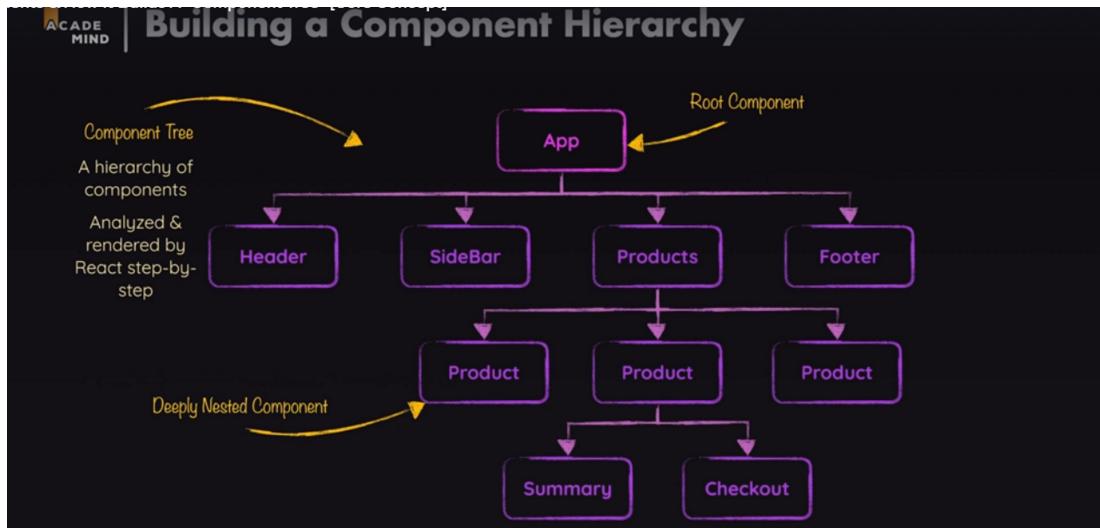
REACT-ESSENTIALS ...
> node_modules
< src
  > assets
  ⚭ App.jsx
  # index.css
  ⚭ index.jsx
  .gitignore
  <> index.html
  {} package-lock.json
  {} package.json
  JS vite.config.js
  # yarn.lock
  
```

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <meta charset="UTF-8" />
5      <link rel="icon" type="image/svg+xml" href="/vite.svg" />
6      <meta name="viewport" content="width=device-width, initial-scale=1" />
7      <title>React Essentials</title>
8    </head>
9    <body>
10      <div id="root"></div>
11      <script type="module" src="/src/index.jsx"></script>
12    </body>
13  </html>
14
  
```

```
{} package-lock.json    12  |  </body>
{} package.json        13  |  </html>
JS vite.config.js     14
yarn.lock
```

```
1 import ReactDOM from "react-dom/client";
2
3 import App from "./App.jsx";
4 import "./index.css";
5
6 const entryPoint = document.getElementById("root");
7 ReactDOM.createRoot(entryPoint).render(<App />);
8
```



From Component Tree To DOM Node Tree

```
<div>
  <header>
    
    <h1>React Essentials</h1>
    <p>Learn about all the core React concepts!</p>
  </header>
  <main>
    <p>Time to get started!</p>
  </main>
</div>
```

Built-in Components vs Custom Components



Built-in Components

Name starts with a **lowercase** character

Only **valid, officially defined** HTML elements are allowed

Are **rendered as DOM nodes** by React (i.e., displayed on the screen)



Custom Components

Name starts with a **uppercase** character

Defined by you, "wraps" built-in or other custom components

React **"traverses"** the component tree until it has only built-in components left

Outputting Dynamic Content in JSX



Static Content

Content that's **hardcoded** into the JSX code

Can't change at runtime

Example
<h1>Hello World!</h1>



Dynamic Content

Logic that **produces the actual value** is added to JSX

Content / value is **derived** at runtime

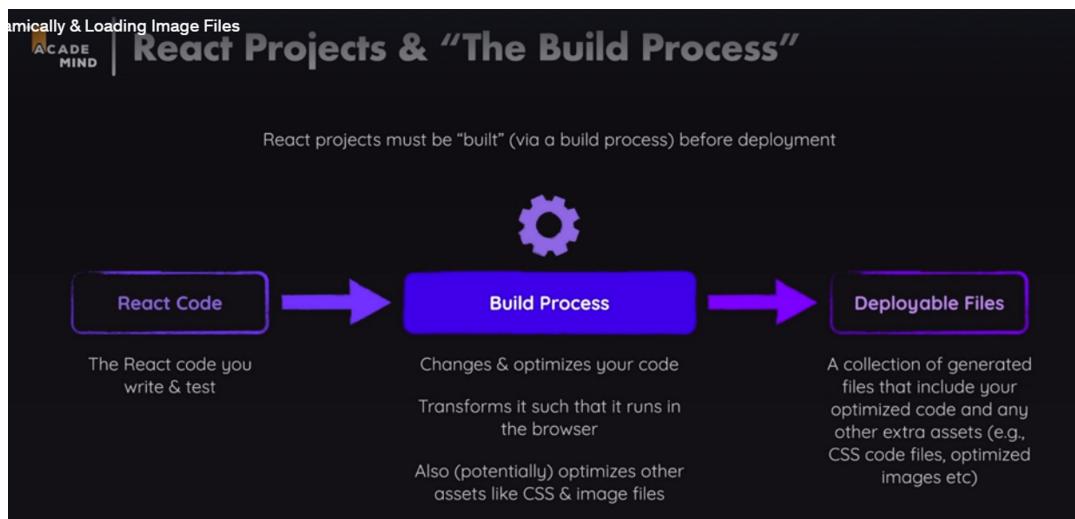
Example
<h1>{user_name}</h1>

```

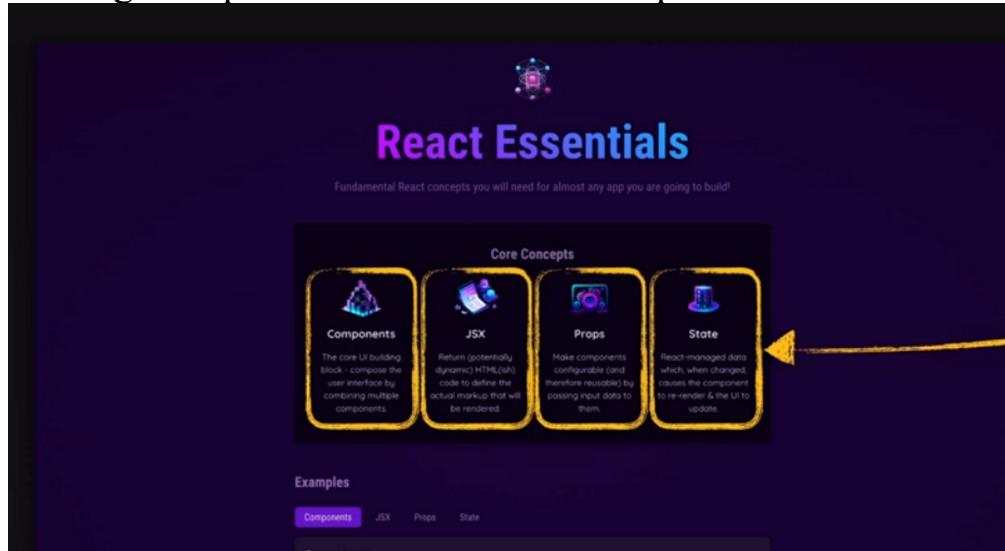
1 const reactDescriptions = ['Fundamental', 'Crucial', 'Core'];
2
3 function genRandomInt(max) {
4   return Math.floor(Math.random() * (max + 1));
5 }
6
7 function Header() {
8   return (
9     <header>
10    
11    <h1>React Essentials</h1>
12    <p>
13      {1+1} React concepts you will need for almost any app you are
14      going to build!
15    </p>

```

Important: if-statements, for-loops, function definitions and other block statements are not allowed here!
Only expressions that directly produce a value.

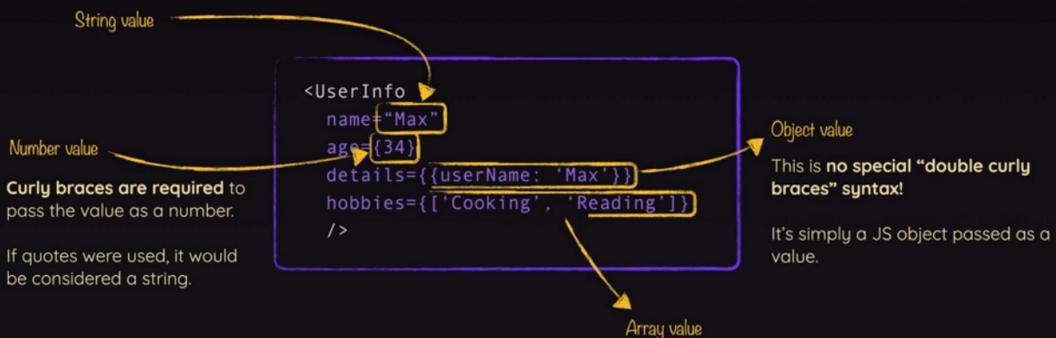


Making Components Reusable with Props



Props Accept All Value Types

You're not limited to text values!



Example with Props [Core Concept]

Understanding Props

1 Set Props

Props are “custom HTML attributes” set on components

```
<CoreConcept>
  title="Components"
  description="Core UI Building Blocks"
</CoreConcept>
```

2 Merge into object

React merges all props into a single object

3 Receive Props

Props are passed to the component function as the first argument by React

```
function CoreConcept(props) {
  return <h3>{props.title}</h3>;
}
```

```
{
  title: 'Components',
  description: 'Core UI ...'
}
```

Introduction: The special “children” Prop [Core Concept]

The Special “children” Prop

The “children” Prop

React automatically passes a special prop named “children” to every custom component

```
<Modal>
  <h2>Warning</h2>
  <p>Do you want to delete this file?</p>
</Modal>
```

Content for “children”
The content between component opening and closing tags is used as a value for the special “children” prop

Modal Component

```
function Modal(props) {
  return <div id="modal">{props.children}</div>;
}
```

“children” Prop vs “Attribute Props”

Using “children”

<TabButton>Components</TabButton>

```
function TabButton({ children }) {  
  return <button>{children}</button>;  
}
```

For components that take a **single piece of renderable content**, this approach is closer to “normal HTML usage”

This approach is **especially convenient** when passing **JSX code as a value** to another component.

Using Attributes

```
<TabButton label="Components"></TabButton>
```

```
function TabButton({ label }) {  
  return <button>{label}</button>;  
}
```

This approach makes sense if you got **multiple smaller pieces of information** that must be passed to a component.

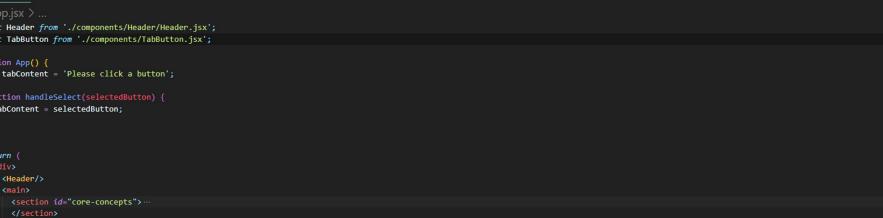
Adding **extra props** instead of just wrapping the content with the component tags mean **extra work**.

Ultimately, it comes down to your use-case and personal preferences.

```
1 export default function TabButton({ children }) {
2   function
3
4     return (
5       <li>
6         <button onClick={()>{children}</button>
7       </li>
8     );
9 }
```



The advantage of defining these event handler functions inside the component function is that they then have access to the component's props and state (covered later).



```
src > App.jsx < ...  
3 import Header from './components/Header/Header.jsx';  
4 import TabButton from './components/TabButton.jsx';  
5  
6 function App() {  
7   let tabContent = 'Please click a button';  
8  
9   function handleSelect(selectedButton) {  
10     tabContent = selectedButton;  
11   }  
12  
13  
14   return (  
15     <div>  
16       <Header/>  
17       <main>  
18         <section id="core-concepts"> ...  
19         </section>  
20  
21         <section id="examples">  
22           <h2>Examples</h2>  
23           <menu>  
24             <TabButton  
25               onClick={() => handleSelect('components')}>  
26               Components  
27             </TabButton>  
28             <TabButton onClick={() => handleSelect('jsx')}>  
29               JSX  
30             </TabButton>  
31             <TabButton onClick={() => handleSelect('props')}>  
32               Props  
33             </TabButton>  
34             <TabButton onClick={() => handleSelect('state')}>  
35               State  
36             </TabButton>  
37           </menu>  
38           <tabContent>  
39             </tabContent>  
40           </section>  
41         </main>  
42       </div>  
43     );  
44   }  
45  
46   export default App;  
47 }
```



By Default, React Components Execute Only Once

React has no built-in component re-



By Default, React Components Execute Only Once

You have to “tell” React If
A Component Should be
Executed Again

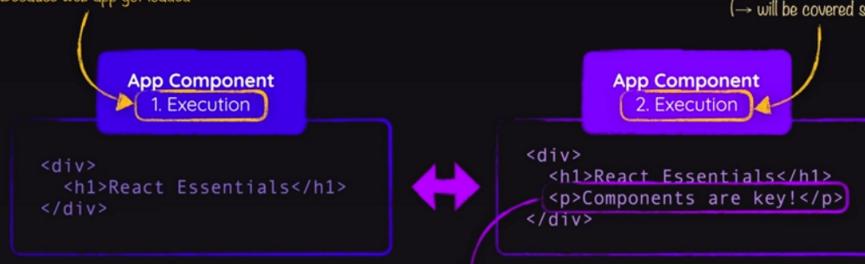
A Look Behind The Scenes of React [Core Concept]

How React Checks If UI Updates Are Needed

React **compares** the **old output** (“old JSX code”) of your component function to the **new output** (“new JSX code”) and **applies any differences** to the actual website UI

Because web app got loaded

Because of a state update
(→ will be covered soon)



Necessary updates will be applied to the real DOM,
ensuring that the visible UI matches the expected output.

ks [Core Concept]

Rules of Hooks

Rules of Hooks

1

Only call Hooks inside of Component Functions

React Hooks must not be called outside of React component functions

```
function App() {
  const [val, setVal] = useState(0);
}
```

```
const [val, setVal] = useState(0);
function App() { ... }
```

2

Only call Hooks on the top level

React Hooks must not be called in nested code statements (e.g., inside of if-statements)

```
function App() {
  const [val, setVal] = useState(0);
}
```

```
function App() {
  if (someCondition)
    const [val, setVal] = useState(0);
}
```

Custom Hook

useState() Yields An Array With Two Elements

And it will **always** be **exactly** two elements

```
const stateArray = useState('Please click a button');
```

Array produced and returned by React's useState() function
Contains exactly two elements

Manage State

Manage data & "tell" React to **re-execute** a component function via React's **useState()** Hook

State updates lead to new state values
(as the component function executes again)

```
const [counter, setCounter] = useState(0);
```

Current state value
Provided by React
May change if the component function is executed again

State updating function
Updates the stored value & "tells" React to re-execute the component function in which useState() was called

Initial state value
Stored by React

```
7
8 Function App() {
9   const [ selectedTopic, setSelectedTopic ] = useState('Please click a but
10
11   function handleSelect(selectedButton) {
12     // selectedButton => 'components', 'jsx', 'props', 'state'
13     setSelectedTopic(selectedButton);
14     console.log(selectedTopic);
15 }
```



When you call state updating function here,
React in the end schedules, this state update and it then re-executes this component
function.

So therefore the updated value will only be available after this app component function
executed again.

Outputting List Data Dynamically

The screenshot shows the React Essentials website on the left and a browser's developer tools panel on the right. The website features a dark purple header with the title 'React Essentials' in large, colorful letters. Below the header, a section titled 'Core Concepts' is displayed with four cards: 'Components' (represented by a building icon), 'JSX' (represented by a document icon), 'Props' (represented by a gear icon), and 'State' (represented by a database icon). Each card has a brief description. To the right, the browser's developer tools show a 'Issues' tab with one warning: 'Warning: Each child in a list should have a unique "key" prop.' The URL for this warning is <https://reactjs.org/link/warning-keys>. The developer tools also show the execution stack for the 'App' component, with the warning being the first item in the list.

```
APP COMPONENT EXECUTING App.jsx:18
✖ Warning: Each child in a list should
  react-jsx-dev-runtime.development.js:
  87
    have a unique "key" prop.

Check the render method of `App`. See
https://reactjs.org/link/warning-keys
for more information.
  at CoreConcept (http://localhost:5
  173/src/components/CoreConcept.jsx:18:
  3)
    at App (http://localhost:5173/src/
  App.jsx?t=1689145943140:26:45)

4 TABBUTTON COMPONENT EXECUTING TabButton.jsx:2
APP COMPONENT EXECUTING App.jsx:18
```

Module Summary



React Essentials

Components, JSX & State

- ▶ Building User Interfaces with **Components**
- ▶ Using, Sharing & Outputting **Data**
- ▶ Handling User **Events**
- ▶ Building Interactive UIs with **State**