

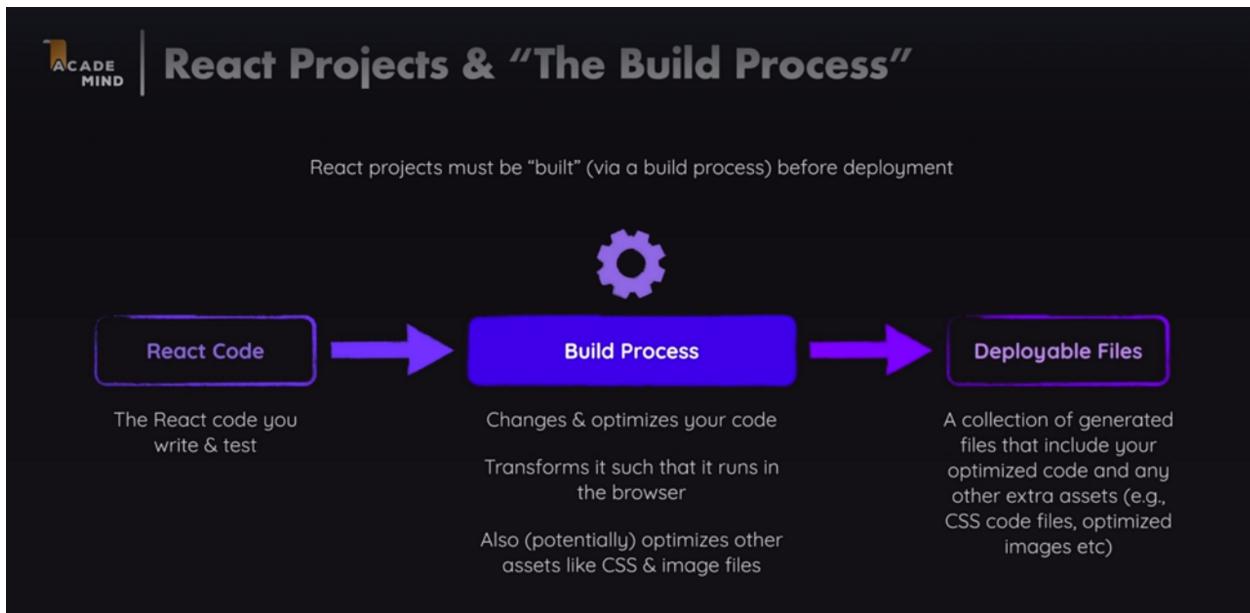
## Section 4: React Essentials - Deep Dive

19 September 2024 09:00

The thumbnail features the Academind logo at the top left. The main title 'React Essentials - Deep Dive' is in large, bold, white font. Below it, a subtitle 'Beyond The Basics' is also in white. To the left of the title is a circular icon containing a stylized atomic model with three spheres. A horizontal blue bar runs across the middle of the title area. At the bottom right corner, there is a small 'Udemy' logo.

- ▶ Behind The Scenes of **JSX**
- ▶ Structuring **Components** and **State**
- ▶ **Advanced State** Usage
- ▶ **Patterns & Best Practices**

Now we'll get started in this section by taking another look at this JSX code, this HTML in JavaScript code about which you learned in the last course section. Because there I mentioned that this JSX code, of course, is a non-standard feature. It's not supported by the browser and therefore the code you write in your code files here is not the code that ends up in the browser. Instead, as you learned, we got a build process.



You could also build React Apps without using JSX.

## You Don't Need JSX (But It's Convenient)

```
<div id="content">
  <p>Hello World!</p>
</div>
```

Requires build process & code transformation

```
React.createElement(
  'div',
  { id: 'content' },
  React.createElement(
    'p',
    null,
    'Hello World'
  )
);
```

Works without special build process & transformation

**Component Type**  
Identifies the to-be-rendered component

**Props Object**  
Sets component props

**Child Content**  
The content passed between the component tags

## Working With Fragments

```
App.jsx 1 ●
32
33
34
35   return [
36     <Header />
37     <main>...
38   ];
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81   export default App;
82 }
```

you can't return multiple items in a return statement.

```
return (
  <Fragment>
    <Header />
    <main>...
  </main>
</Fragment>
);
```

React Essentials

Crucial React concepts will need for almost any app you are going to build!

**Core Concepts**

- Components**: The core UI building block - compose the user interface by combining multiple components.
- JSX**: Return (potentially dynamic) HTML(ish) code to define the actual markup that will be rendered.
- Props**: Make components configurable (and therefore reusable) by passing input data to them.
- State**

```

<!DOCTYPE html>
<html lang="en">
  <head> ... </head>
  <body>
    <div id="root">
      <header> ... </header> == $0
      <main> ... </main>
    </div>
    <script type="module" src="/src/index.js?t=1689227738859"></script>
  </body>
</html>

```

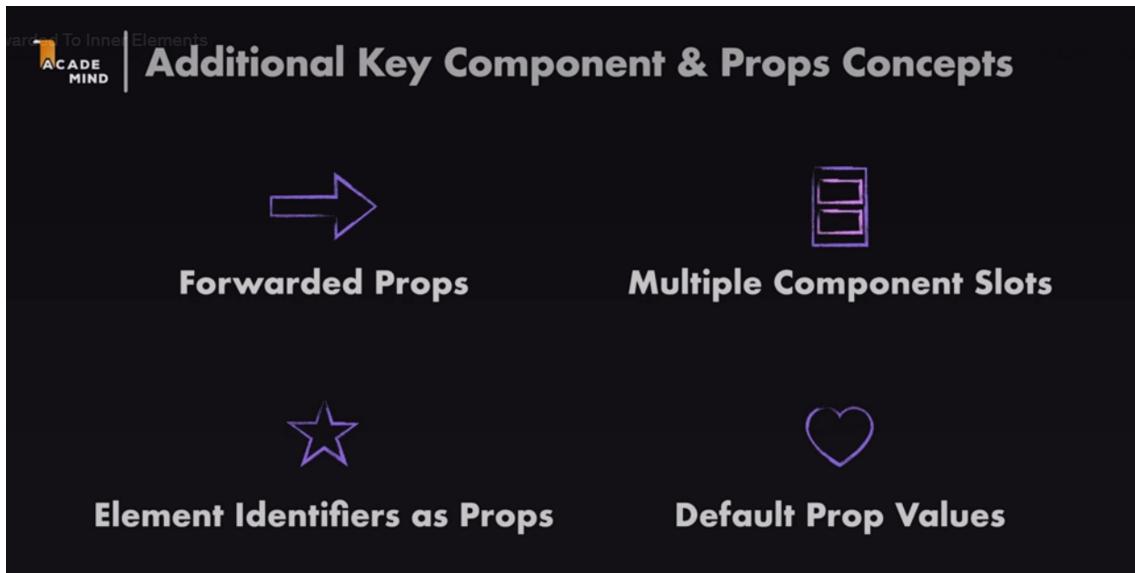
html body div#root header

Styles Computed Layout >

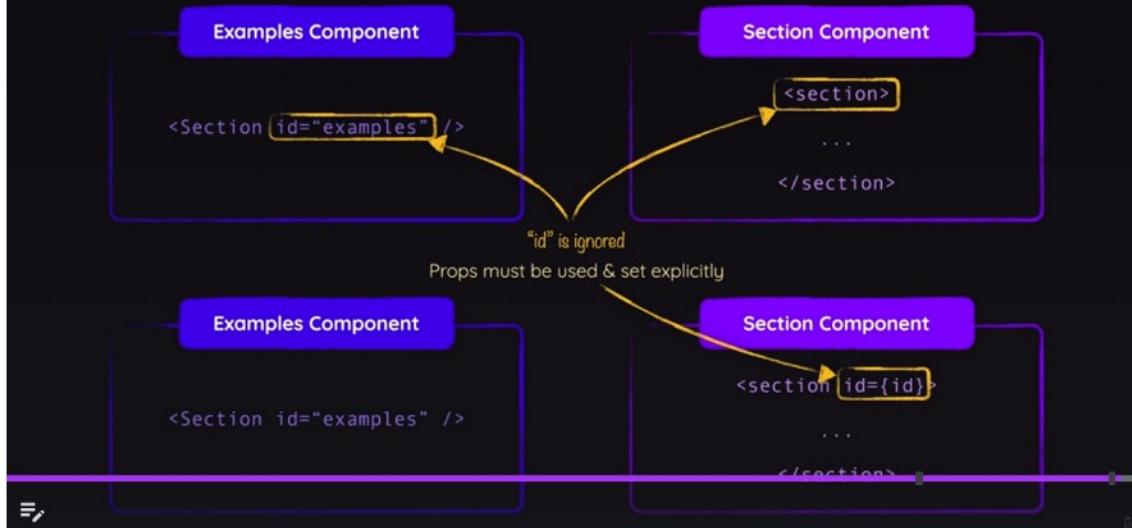
no extra div!

## Splitting Components By Feature & State

for preventing unnecessary Header and CoreConcepts Component rendering, we have split the components.



# Props Are Not Forwarded Automatically



## Forwarding Props To Wrapped Elements

```
return (
  <Section title="Examples" id="examples">
    <menu>
      <TabButton
        isSelected={selectedTopic === 'components'}
        onClick={() => handleSelect('components')}>
        Components
      </TabButton>
      <TabButton
        isSelected={selectedTopic === 'jsx'}
        onClick={() => handleSelect('jsx')}>
        JSX
      </TabButton>
      <TabButton
        isSelected={selectedTopic === 'props'}
        onClick={() => handleSelect('props')}>
        Props
      </TabButton>
```

```
monents > Section.jsx > [o] Section
import React from 'react';

const Section = ({ title, children, ...props }) => {
  return (
    <section {...props}>
      <h2>{title}</h2>
      {children}
    </section>
  );
};

export default Section;
```

But this three dots thing here is built into JavaScript and basically tells JavaScript to collect all other props that might be received on this section component and merge them into a props object. So into one JavaScript object

**Onwards To The Next Project & Advanced Concepts**

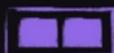


React & Advanced Concepts  
ACADE MIND | **New Project, New Concepts**

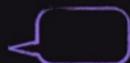
  
**Multiple State Values**

  
**Nested Lists**

  
**Lifting State Up**

  
**Array & Object States**

  
**Derived State**

  
**Component Functions vs  
“Normal Functions”**

**Updating State Based On Old State Correctly**



## Updating State Based On Old State



```
setIsEditing(!isEditing);
```

If your **new state depends on your previous state** value, you should **not** update the state like this



```
setIsEditing(wasEditing => !wasEditing);
```

Instead, **pass a function** to your state updating function

This function will **automatically be called** by React and will receive the **guaranteed latest state value**

Well, the problem with this code here is that React behind the scenes is, in the end, scheduling those state updates you're triggering with those state updating functions, like `setIsEditing`. So this state update here is not performed instantly. Instead, it's scheduled by React to be performed in the future. Now, the future will probably be in one or two milliseconds.

So it's really fast, but it's not instant. That's important. Now, as a React developer, it's really important to understand that React behaves that way and that behavior also explains why when you copy this state updating function and you paste it in immediately after the first state updating function, you might not get the result you might expect.

## Somewhere in your code

Code snippets in different places in your app's code

```
setIsEditing(true); → 1 → Update state to true  
...  
setIsEditing(false); → 2 → Update state to false  
...  
setIsEditing(true); → 3 → Update state to true  
...  
setIsEditing(false); → 4 → Update state to false
```

## React's State Updating Schedule

Managed behind the scenes by React

State updates are **not performed instantly** but at some point in the future  
(when React has time for it)

In most cases, those state updates of course still are executed **almost instantly**

```
export default function Player({ name, symbol }) {  
  const [isEditing, setIsEditing] = useState(false);  
  
  function handleEditClick() {  
    setIsEditing(!isEditing, () => true);  
    setIsEditing(!isEditing, () => false);  
  }  
  
  let playerName = <span className="player-name">{name}</span>;  
  // let btnCaption = 'Edit';  
  
  if (isEditing) {  
    playerName = <input type="text" required value={name} />;  
    // btnCaption = 'Save';  
  }  
}
```



## Update Object-State Immutable



Objects & arrays (which technically are objects) are reference values in JavaScript



NOT creating a copy  
(because user is an object = a reference value)



```
const updatedUser = user;  
updatedUser.name = 'Max'
```

Editing the user object in memory

You should therefore not mutate them directly — instead create a (deep) copy first!



```
const updatedUser = [...user];  
updatedUser.name = 'Max'
```

Editing the copy, not the original

Udemy

```
const GameBoard = () => {  
  const [gameBoard, setGameBoard] = useState(initialGameBoard);  
  
  function handleSelectSquare(rowIndex, colIndex) {  
    //not recommended.  
    setGameBoard((prevGameBoard) => {  
      prevGameBoard[rowIndex][colIndex] = 'X';  
      return prevGameBoard;  
    });  
  }  
}
```

you're dealing with a reference value in JavaScript. and therefore if you would be updating it like this, you would be updating the old value in -memory immediately, even before this scheduled state update was executed by React. And this can again lead to strange bugs or side-effect.

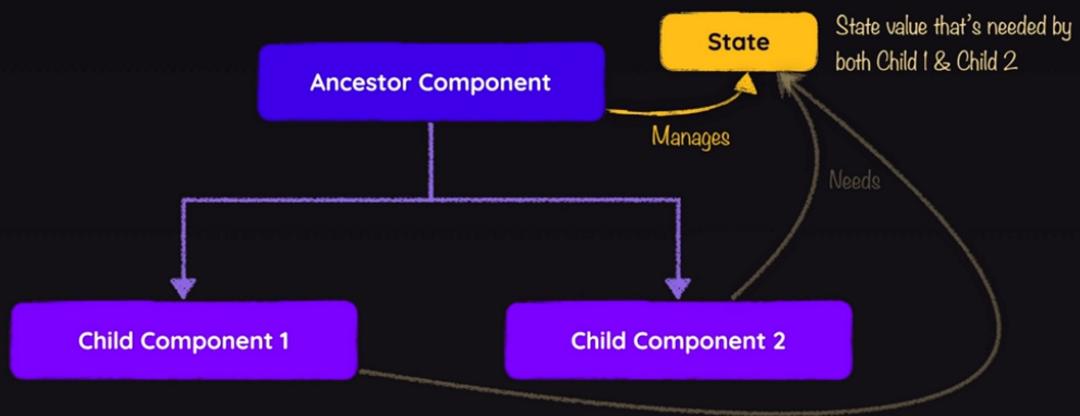
<https://academind.com/tutorials/reference-vs-primitive-values>

# Lifting State Up

Lift the state up to the **closest ancestor component** that has access to all components that need to work with that state

# Lifting State Up

Lift the state up to the **closest ancestor component** that has access to all components that need to work with that state



```
7   tion App() {
8     nst [gameTurns, setGameTurns] = useState([]);
9     nst [activePlayer, setActivePlayer] = useState('X');
10
11    nction handleSelectSquare(rowIndex, colIndex) {
12      setActivePlayer((curActivePlayer) => (curActivePlayer === 'X' ? 'O' : 'X'));
13      setGameTurns((prevTurns) => {
14        const updatedTurns = [
15          { square: { row: rowIndex, col: colIndex }, player: activePlayer },
16          ...prevTurns,
17        ];
18      });
19
20
```