React Props and State

Software Development Bootcamp

Topic

Props

What Are Props?

Props (short for properties) are a way to pass data from parent to child components in React.

- Props are read-only
- They help make your components reusable
- Props flow downwards from parent to child
- Props can be any JavaScript data type: strings, numbers, objects, arrays, or even functions

Why Use Props?

Think of props like arguments passed to a function - they allow you to pass data into a component to customize its behavior or appearance.

- They allow you to customize child components based on data from their parent
- Props promote a unidirectional data flow, making your application easier to understand and debug

How To Use Props

`ParentComponent.jsx`

- This is the parent component that renders the ChildComponent.
- It passes a prop called `message` to the ChildComponent.
- The prop is set using an attribute-like syntax: `message="Hello from props"`

```jsx
import React from 'react'
import ChildComponent from './ChildComponent'

function ParentComponent() {
 return (
   <ChildComponent message="Hello From Props!" />
 )
}

export default ParentComponent
```

How To Use Props

`ChildComponent.jsx`

- This is the child component that receives and uses the prop.
- It takes a single parameter conventionally named **`props`**
- We access the **`message`** prop using dot notation: **`props.message`**
- The component renders a paragraph element containing the value of **`props.message`**

```jsx
import React from 'react'

function ChildComponent(props) {
 return (
    <div>
        <p>{props.message}</p>
    </div>
 )
}

export default ChildComponent
```

How To Use Props

`App.jsx`

- Import `ParentComponent`
- `App.jsx` renders the `ParentComponent`
- When this code runs it will display a paragraph with the text “Hello From Props!”

```
import ParentComponent from
"./ParentComponent"

function App() {
 return (
   <>
     <ParentComponent />
   </>
 )
}

export default App
```

Passing Multiple Props

- You can pass multiple props to a component
- And access them in the child component

Remember, props are read-only. The child component should never modify the props it receives. If you need to modify data, you should use state instead, which we'll cover in the next slides.

Topic

Rendering A List Using .map()

Fruit List Example

- Component Declaration
  - `function FruitList()` defines a functional component named `FruitList`
- Data Source
  - `const fruits = [...]`
  - Define an array a fruit names. In a real application this could come from props or state.

```
import React from "react";

function FruitList() {
 const fruits = ["Apple", "Banana",
"Cherry", "Date"];
 return (
   <ul>
     {fruits.map((fruit, index) => (
       <li key={index}>{fruit}</li>
     ))}
   </ul>
 );
}

export default FruitList;
```

Fruit List Example

- Return Statement:
  - The component returns a single `<ul>` element
- Rendering Multiple Items
  - `fruits.map((fruit, index)=> (...))}`
  - We use the `map` function to iterate over the `fruits` array
  - For each fruit, we create a new `<li>` element
  - The `key` prop is set to the index (Note: using index as key is not ideal if the list order can change)
- JSX and JavaScript
  - The curly braces allow us to embed JavaScript expressions within JSX

```
import React from "react";

function FruitList() {
 const fruits = ["Apple", "Banana",
"Cherry", "Date"];
 return (
   <ul>
     {fruits.map((fruit, index) => (
       <li key={index}>{fruit}</li>
     ))}
   </ul>
 );
}

export default FruitList;
```

Topic

State

What Is State?

State is a way to store and manage data within a component that can change over time.

Think of state as a component's memory. It's where you store property values that belong to the component and may change over time.

Important Aspects Of State

- State is mutable (can be changed)
- Changes to state trigger re-renders of the component
- State is managed within the component
- State represents the internal data of a component
- It's used for data that can change based on user actions or other events
- State is private to a component by default
- When state changes, React efficiently updates only the necessary parts of the DOM
- State is managed using the `useState` hook

What Are React Hooks?

React Hooks are functions that let you "hook into" React state and lifecycle features from function components.

- Hooks can only be used in functional components
- They must be called at the top level of your component

Why Use Hooks?

Hooks solve several problems in React:

- Reusing stateful logic between components without changing your component hierarchy
- Splitting complex components into smaller functions based on what pieces are related

Rules Of Hooks

To ensure hooks work correctly, follow these two rules:

1. Only call hooks at the top level of your component
    a. Don’t call hooks inside loops, conditions, or nested functions
2. Only call hooks from React function components or custom hooks
    a. Don't call hooks from regular JavaScript functions

Using State With Hooks

1. Importing `useState`
    a. Import the `useState` hook from React.
2. Declaring State:
    a. `const [count, setCount] = useState(0)`
    b. This line uses array destructuring to declare a state variable `count` and its corresponding setter function `setCount`
    c. The `useState` hook returns an array with two elements: the current state value and a function to update it.

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

function Counter() {
   const [count, setCount] = useState(0)
 return (
   <div>
        <p>You Clicked {count} times</p>
        <button onClick={() => setCount(count
+ 1)}>
            Click Me!
        </button>
   </div>
 )
}

export default Counter;
```

Using State With Hooks

3. Using State:
    a. `<p>You Clicked {count} times</p>`
    b. We can use the `count` state variable directly in JSX
4. Updating State:
    a. `<button onClick={() => setCount(count + 1)}>`
    b. When the button is clicked, we call `setCount(count + 1)` to increment the count
    c. React will then re-render the component with the new state value

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

function Counter() {
   const [count, setCount] = useState(0)
 return (
   <div>
        <p>You Clicked {count} times</p>
        <button onClick={() => setCount(count
+ 1)}>
            Click Me!
        </button>
   </div>
 )
}

export default Counter;
```

Using `useState` With A Simple Form

Let's create a simple form that uses the useState hook to manage separate states for each input

Simple Form Example

- State Initialization:
  - Separate `useState` calls for `username` and `email`
  - This creates two independent state variables and their setter functions
- Change Handler Functions:
  - Separate handler functions for each input: `handleUsernameChange` and `handleEmailChange`
  - Each function updates its respective state directly

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

function SimpleForm() {
 const [username, setUsername] = useState('');
 const [email, setEmail] = useState('');
 const handleUsernameChange = (e) => {
   setUsername(e.target.value);
 };
 const handleEmailChange = (e) => {
   setEmail(e.target.value);
 };
 const handleSubmit = (e) => {
   e.preventDefault();
   console.log('Form submitted:', { username,
email });
 };
```

Simple Form Example

- `handleSubmit` function:
  - Prevents the default form submission behavior
  - Logs both the `username` and `email` states (in a real app, you might send this data to a server)

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

function SimpleForm() {
 const [username, setUsername] = useState('');
 const [email, setEmail] = useState('');
 const handleUsernameChange = (e) => {
   setUsername(e.target.value);
 };
 const handleEmailChange = (e) => {
   setEmail(e.target.value);
 };
 const handleSubmit = (e) => {
   e.preventDefault();
   console.log('Form submitted:', { username,
email });
 };
```

Simple Form

- Form Structure:
  - The form's **`onSubmit`** event is handled by **`handleSubmit`**
  - Each input's **`value`** is controlled by its respective state
  - Each input has its own **`onChange`** handler

```
return (
  <form onSubmit={handleSubmit}>
    <div>
      <label htmlFor="username">Username:</label>
      <input
        type="text"
        id="username"
        value={username}
        onChange={handleUsernameChange}
      />
    </div>
    <div>
      <label htmlFor="email">Email:</label>
      <input
        type="email"
        id="email"
        value={email}
        onChange={handleEmailChange}
      />
    </div>
    <button type="submit">Submit</button>
  </form>
);
```

Exercise

Name Display