Day 6

React Props

Data Passing

kodemaven-portfolio.vercel.app

What are Props?

Props are **objects** that hold the values of attributes of a component and are used to **pass data** from a **parent component** to a **child component**.

They are read-only, meaning they cannot be modified by the child component receiving them.

Props help make components more dynamic and interactive by allowing them to receive data from other components and render accordingly.

Parent

Child

Key Characteristics of Props:

- **Immutable:** Once passed to a component, props cannot be modified by that component.
- **Unidirectional Flow:** Props are passed from parent to child, maintaining a unidirectional data flow, which simplifies the application's state management.
- **Reusable:** Props enable reusability of components by allowing them to accept dynamic data inputs.

How to Use Props In React

To understand how props work, let's consider an example where we have a Parent component that passes data to a Child component.

Step 1: Define the Child Component:
Remember, we learned how to create a component in the lesson for Day 3. **Right-click** on your "**components**" folder, and select "**New file**". Type in your component name as "**ChildComponent.jsx**" and hit the enter button. Then write this codes:

```
// ChildComponent.jsx

const ChildComponent = (props) => {
  return (
    <div>
      <h1>{props.greeting}</h1>
      <p>{props.message}</p>
    </div>
  );
};

export default ChildComponent;
```

kodemaven-portfolio.vercel.app

In the codes above, the **ChildComponent** expects to receive two props: greeting and message. The props are accessed using props.greeting and props.message.

Step 2: Pass Props from the Parent Component

Also create the "**ParentComponent.jsx**" file and write these codes:

```
1  // ParentComponent.jsx
2  import ChildComponent from './ChildComponent';
3
4  const ParentComponent = () ⇒ {
5    return (
6      <div>
7        <ChildComponent greeting="Hello, World!"
          message="Welcome to learning React props!" />
8      </div>
9    );
10 };
11
12 export default ParentComponent;
```

Here, the **ParentComponent** passes two props, **greeting** and **message**, to the **ChildComponent**. The values "**Hello, World!**" and "**Welcome to learning React props!**" are sent down to the child, which then displays them.

Step 3: Render the Parent Component

Finally, the **ParentComponent** is rendered in the root component, usually “**App.jsx**”:

```
1  import './App.css'
2  import ParentComponent from "./ParentComponent"
3
4  function App() {
5
6    return (
7      <>
8        <ParentComponent />
9      </>
10   )
11
12 }
13 export default App
```

To view if this works, go to your terminal on your IDE, navigate to your project's folder (the folder you created for your project), and run your development server using this command: "**npm run dev**".

Then click on or paste the localHost URL to your browser.

Hello, World!

Welcome to learning React props!

If you followed these steps accurately, you should get this in your browser.

Destructuring Props

Props can also be destructured directly within the function signature or inside the function body, making the code cleaner and easier to read:

Props destructuring is a way to extract props values from the props object in a React component. It allows you to access props values directly, without having to use the props. prefix.

instead of (props)

used directly

```
1 const ChildComponent = ({ greeting, message }) ⇒ {
2   return (
3     <div>
4       <h1>{greeting}</h1>
5       <p>{message}</p>
6     </div>
7   );
8 };
9 export default ChildComponent;
```

This is called "**function parameter destructuring**" and it's a concise way to extract props values.

Or, inside the function body:

```
const ChildComponent = (props) => {
  const { greeting, message } = props;

  return (
    <div>
      <h1>{greeting}</h1>
      <p>{message}</p>
    </div>
  );
};
export default ChildComponent;
```

In this example, the **greeting** and **message** props are extracted from the **props** object using destructuring. This allows you to use the greeting and message variables directly in the component.

Default Props

You can define **default values** for props in case they are not provided by the parent component. This is useful for preventing errors or providing fallback values:

```
1  const ChildComponent = ({ greeting, message }) => {
2    return (
3      <div>
4        <h1>{greeting}</h1>
5        <p>{message}</p>
6      </div>
7    );
8  };
9
10 export default ChildComponent;
11
12 ChildComponent.defaultProps = {
13   greeting: 'Hello!',
14   message: 'This is the default message.',
15 };
```

If the **ParentComponent** does not pass a **greeting** or **message** prop, the ChildComponent will use the default values specified in **defaultProps**.

Different Types of Props in React

1. String Props: String props are used to pass text or characters to a component. Ideal for passing static text, such as names, labels, or messages.

```
1 const Greeting = (props) => {
2   return <h1>Hello, {props.name}!</h1>;
3 };
4
5 // Usage
6 <Greeting name="John" />
```

Here, the **name** prop is a string "**John**" passed to the **Greeting** component.

2. Number Props: Number props are used to pass numeric values, such as integers or floats. Commonly used for passing quantities, IDs, or any numeric data.

Example:

```
const AgeDisplay = (props) => {
  return <p>Age: {props.age}</p>;
};

// Usage
<AgeDisplay age={30} />
```

The **age** prop is a number (30) passed to the **AgeDisplay** component by wrapping it with curly braces.

3. Boolean Props: Are used to pass true or false values. Useful for toggling states, conditional rendering, or setting flags.

Example:

```
1 const Status = (props) => {
2   return <p>{props.isActive ? "Active" : "Inactive"}</p>;
3 };
4
5 // Usage
6 <Status isActive={true} />
7
```

The **isActive** prop is a boolean (true), which determines the displayed text.

4. Object Props: Object props are used to pass complex data structures like JavaScript objects. Helpful for passing related data items, such as user profiles, configurations, or settings.

Example:

```
const UserProfile = (props) => {
  return <p>{props.user.name} is {props.user.age} years old.</p>;
};

// Usage
const user = { name: "Jane", age: 25 };

<UserProfile user={user} />
```

The **user** prop is an object containing **name** and **age** properties, passed to the **UserProfile** component.

5. Array Props: Array props are used to pass lists or collections of data. Ideal for rendering lists, looping through items, or passing multiple values.

Example:

```
1 const ItemList = (props) => {
2   return (
3     <ul>
4       {props.items.map((item, index) => (
5         <li key={index}>{item}</li>
6       ))}
7     </ul>
8   );
9 };
10
11 // Usage
12 const items = ["Apple", "Banana", "Cherry"];
13 <ItemList items={items} />
```

The **items prop** is an array (["Apple", "Banana", "Cherry"]), used to dynamically render a list of items.

6. Function Props: Function props are used to pass functions that can be executed within the child component.

Used for handling events, callbacks, or sharing behavior between components.

Example:

```
1 const Button = ({ handleClick }) => {
2   return <button onClick={handleClick}>Click Me</button>;
3 };
4
5 // Usage
6 const handleClick = () => {
7   alert("Button clicked!");
8 };
9 <Button handleClick={handleClick} />
```

The **handleClick prop** is a function passed to the **Button** component to handle the button's click event.

Prop Types Checking

React provides a way to validate the props passed to a component using **prop-types**.

It helps ensure that the component receives the **correct type of data** and prevents errors caused by incorrect prop types.

For example:

Check our **ChildComponent.jsx**, there is an error specifying that the props “greeting” and “message” are missing validation. This is because we didn’t tell React the types of data the props are holding and how important they are.

```
const ChildComponent = (props) => {
  return (
    <div>
      <h1>{props.greeting}</h1>    'greeting' is missing in props validation
      <p>{props.message}</p>    'message' is missing in props validation
    </div>
  );
};

export default ChildComponent;
```

To correct this, we will have to declare the data types for each prop.

How it works:

First, install the prop-types package from react: Open your terminal, in your project folder directory, type “**npm install prop-types**”. The library will be installed.

Now, let’s use this library in our component by importing it like this:

```
1   // ChildComponent.js
2   import PropTypes from "prop-types"
```

And declare our prop types:

```
13  ChildComponent.propTypes = {
14    greeting: PropTypes.string.isRequired,
15    message: PropTypes.string,
16  };
```

Full code:

```
1 // ChildComponent.js
2 import PropTypes from "prop-types"
3
4 const ChildComponent = (props) => {
5   return (
6     <div>
7       <h1>{props.greeting}</h1>
8       <p>{props.message}</p>
9     </div>
10   );
11 };
12 export default ChildComponent;
13
14 // props data type declaration
15 ChildComponent.propTypes = {
16   greeting: PropTypes.string.isRequired,
17   message: PropTypes.string,
```

PropTypes.string indicates that **greeting** and **message** variables must be a string while the **.isRequired** attached to the variable **greeting** means that it **must** be provided. If greeting is not provided, a warning will be shown in the console during development.

kodemaven-portfolio.vercel.app

The different types of prop validators are:

- **string**
- **number**
- **bool**
- **array**
- **object**
- **func**
- **node**
- **element**
- **symbol**
- **oneOfType (e.g., PropTypes.oneOfType([PropTypes.string, PropTypes.number]))**
- **oneOf (e.g., PropTypes.oneOf(['value1', 'value2']))**
- **shape (e.g., PropTypes.shape({ name: PropTypes.string, age: PropTypes.number }))**

Props Drilling

This can create a situation where components are "drilled" with props they don't use, just to ensure the right data reaches the necessary destination.

Illustrating Prop Drilling:

Let's look at an example where we have a deeply nested structure and a piece of data (userName) needs to be passed from a top-level App component to a deeply nested UserProfile component.

```jsx
import React from 'react';

// Component structure
const App = () => {
  const userName = "Oluwakemi Oluwadahunsi";

  return (
    <div>
      <h1>Welcome to the App</h1>
      {/* Passing userName prop down to ParentComponent */}
      <ParentComponent userName={userName} />
    </div>
  );
};

const ParentComponent = ({ userName }) => {
  return (
    <div>
      <h2>This is the Parent Component</h2>
      {/* Passing userName prop down to ChildComponent */}
      <ChildComponent userName={userName} />
    </div>
  );
};

const ChildComponent = ({ userName }) => {
  return (
    <div>
      <h3>This is the Child Component</h3>
      {/* Passing userName prop down to UserProfile component */}
      <UserProfile userName={userName} />
    </div>
  );
};

const UserProfile = ({ userName }) => {
  return (
    <div>
      <h4>User Profile</h4>
      <p>User Name: {userName}</p>
    </div>
  );
};
```

kodemaven-portfolio.vercel.app

- **Prop Drilling Issue:** In this example, the **userName** prop is defined in the top-level App component. It is passed down through the **ParentComponent** and **ChildComponent** to finally reach the **UserProfile** component.
- The intermediate components (**ParentComponent** and **ChildComponent**) do not actually use **userName** but are forced to pass it down to fulfill the needs of **UserProfile**
- **Unnecessary Prop Passing:** This leads to a situation where components are unnecessarily aware of props they don't need, making the code less readable and harder to maintain.

Props drilling can make large projects to be difficult to maintain states and props. To avoid this, use the **Context API** or state management libraries like Redux to manage global or shared states and props.

Best Practices for Using Props

1. Use Destructuring: Always destructure props to make your code more readable and concise.

So, instead of this:

```
// ChildComponent.js
const ChildComponent = (props) => {
  return (
    <div>
      <h1>{props.greeting}</h1>
    </div>
  );
};
export default ChildComponent;
```

Do this:

```
// ChildComponent.js
const ChildComponent = ({ greeting }) => {
  return (
    <div>
      <h1>{greeting}</h1>
    </div>
  );
};
export default ChildComponent;
```

2. Use PropTypes to validate the type and presence of required props to catch potential bugs during development.

```
1  // ChildComponent.js
2  Import PropTypes from "prop-types"
3
4  const ChildComponent = ({ greeting }) ⇒ {
5    return (
6      <div>
7        <h1>{greeting}</h1>
8      </div>
9    );
10 };
11 export default ChildComponent;
12
13 ChildComponent.propTypes = {
14   greeting: PropTypes.string.isRequired,
15 }
```

3. Keep Props Simple: Only pass the necessary data to the child components. Avoid passing entire objects or deeply nested data structures unless necessary.

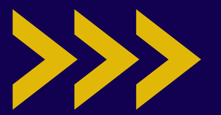
4. Avoid Passing Too Many Props: If you find yourself passing too many props to a component, consider if the component can be broken down further or if a different approach (like context) is more appropriate.

Conclusion

Using props appropriately ensures a smooth and efficient data flow in your application, making components flexible and easy to maintain.

But overusing it can lead to potential bugs and increased development time.

Leveraging the Context API, state management libraries, and better component composition techniques can help avoid unnecessary prop passing and make your application more scalable, readable, and easier to manage.



I hope you found this material useful and helpful.

Remember to:

Like

Save for future reference

&

Share with your network, be helpful to someone 👌

Hi There!

Thank you for reading through

Did you enjoy this knowledge?

Follow my LinkedIn page for more work-life balancing and Coding tips.

LinkedIn: Oluwakemi Oluwadahunsi

kodemaven-portfolio.vercel.app