Web Technologies

CAP 756

Presenter: Dr. Mirza Shuja

ReactJS Components

- A **Component** is one of the core building blocks of React. In other words, we can say that every application you will develop in React will be made up of pieces called components.
- **Components** make the task of building UIs much easier. You can see a UI broken down into multiple individual pieces called components and work on them independently and merge them all in a parent component which will be your final UI.
- In React, we mainly have two types of components:
- **Functional Components**
- **Class Components**

- **Functional Components**: Functional components are simply javascript functions. We can create a functional component in React by writing a javascript function.
- These functions may or may not receive data as parameters. Below example shows a valid functional component in React:
- In React, function components are a way to write components that only contain a render method and don't have their own state. The functional component is also known as a stateless component because they do not hold or manage state.
- They are simply JavaScript functions that may or may not receive data as parameters. We can create a function that takes props(properties) as input and returns what should be rendered.

```
const Democomponent=()=>
{
    return <h1>Welcome Message!</h1>;
}
```

- **Class Components**: The class components are a little more complex than the functional components.
- The functional components are not aware of the other components in your program whereas the class components can work with each other. We can pass data from one class component to other class components.
- The class component is also known as a stateful component because they can hold or manage local state.
- We can use JavaScript ES6 classes to create class-based components in React. Below example shows a valid class-based component in React:

```
class Democomponent extends React.Component
{
    render(){
        return <h1>Welcome Message!</h1>;
    }
}
```

- **Rendering Components**
- React is also capable of rendering user-defined components.
- To render a component in React we can initialize an element with a user-defined component and pass this element as the first parameter to ReactDOM.render() or directly pass the component as the first argument to the ReactDOM.render() method.
- Below syntax shows how to initialize a component to an element:
- *const elementName = <ComponentName />;*

- Open your **index.js** file from your project directory, and make the given below changes in **src/ index.js:**

```
import React from 'react';
import ReactDOM from 'react-dom';
// This is a functional component
const Welcome=()=>
{
        return <h1>Hello World!</h1>
}

ReactDOM.render(
    <Welcome />,
    document.getElementById("root")
);
```

- **Decomposing Components:**
- Decomposing a Component means breaking down the component into smaller components.

```
import React from 'react';
import ReactDOM from 'react-dom';
const Form=()=>
{
    return (
        <div>
            <input type = "text" placeholder = "Enter Text.." />
            <br />  <br />
            <input type = "text" placeholder = "Enter Text.." />
            <br /> <br />
            <button type = "submit">Submit</button>
        </div>
    );
}
ReactDOM.render(
    <Form />,
    document.getElementById("root")
);
```

```
import React from 'react';
import ReactDOM from 'react-dom';
// Input field component
const Input=()=>
{
      return(
            <div>
                  <input type="text" placeholder="Enter Text.." />
                  <br />  <br />
            </div>
      );
}
// Button Component
const Button=()=>
{
      return <button type = "submit">Submit</button>;
}
// Form component
const Form=()=>
{
    return (
            <div>
                  <Input />
                  <Input />
                  <Button />
            </div>
      );
}

ReactDOM.render(
      <Form />,    document.getElementById("root")
);
```

ReactJS Pure Components

- Generally, In **ReactJS**, we use **shouldComponentUpdate()** Lifecycle method to customize the default behavior and implement it when the React component should re-render or update itself.
- Now, **ReactJS** has provided us a **Pure Component**.
- If we extend a class with **Pure Component**, there is no need for **shouldComponentUpdate()** Lifecycle Method.
- **ReactJS Pure Component** Class compares current state and props with new props and states to decide whether the React component should re-render itself or Not.
- **Example**:

```
import React from 'react';

export default class Test extends React.PureComponent{
render(){
    return <h1>Welcome to Learning</h1>;
}
}
```

ReactJS Functional Components

- These are simply JavaScript functions. We can create a functional component to React by writing a JavaScript function.
- These functions may or may not receive data as parameters. In the functional Components, the return value is the JSX code to render to the DOM tree.
- **Example:** Program to demonstrate the creation of functional components.
- **Filepath- src/index.js:** Open your React project directory and edit the **index.js** file from src folder:

```
import React from 'react';
import ReactDOM from 'react-dom';
import Demo from './App';

ReactDOM.render(
<React.StrictMode>
      <Demo />
</React.StrictMode>,
document.getElementById('root')
);
```

- **Filepath- src/App.js:** Open your React project directory and edit the **App.js** file from src folder:

```
import React from 'react';
import ReactDOM from 'react-dom';

const Demo=()=>
{
return <h1>Welcome to Learning</h1>;
}

export default Demo;
```

- We can also use a functional component into another component.
- **Filepath- src index.js:** Open your React project directory and edit the **Index.js** file from src folder

```
import React from 'react';
import ReactDOM from 'react-dom';
import Welcome from './App';

ReactDOM.render(
<React.StrictMode>
     <Welcome />
</React.StrictMode>,
document.getElementById('root')
);
```

- **Filepath- src App.js:** Open your React project directory and edit the **App.js** file from src folder:

```
import React from 'react';
const Welcome=()=>
{
return (
     <h1>Welcome to Learning</h1>
     );
}
const functionExample=()=>
{
     return (
     <Welcome/>
     );
}
export default functionExample;
```

- **React State**
- The state is an updatable structure that is used to contain data or information about the component.
- The state in a component can change over time. The change in state over time can happen as a response to user action or system event. A component with the state is known as stateful components.
- A state must be kept as simple as possible. It can be set by using the **setState()** method and calling setState() method triggers UI updates.
- To set an initial state before any interaction occurs, we need to use the **getInitialState()** method.
- **Defining State**
- To define a state, you have to first declare a default set of values for defining the component's initial state. To do this, add a class constructor which assigns an initial state using **this.state**.
- The '**this.state**' property can be rendered inside **render()** method.

- Example
- The below sample code shows how we can create a stateful component using ES6 syntax.

```
import React, { Component } from 'react';
class App extends React.Component {
 constructor() {
       super();
       this.state = { displayBio: true };
       }
       render() {
            const bio = this.state.displayBio ? (
                 <div>
<p><h3>Welcome to React Learning.</h3></p>
               </div>
                 ) : null;
                 return (
                      <div>
     <h1> Welcome to JavaTpoint!! </h1>
                           { bio }
                      </div>
                 );
      }
}
export default App;
```

- Changing the State
- We can change the component state by using the setState() method and passing a new state object as the argument.
- Now, create a new method **toggleDisplayBio()** in the previous example and **bind this** keyword to the **toggleDisplayBio()** method otherwise we can't access this inside **toggleDisplayBio()** method.
- ***this.toggleDisplayBio = this.toggleDisplayBio.bind(this);***

React Props

- Props stand for "**Properties**." They are **read-only** components.
- It is an object which stores the value of attributes of a tag and work similar to the HTML attributes.
- It gives a way to pass data from one component to other components. It is similar to function arguments.
- Props are passed to the component in the same way as arguments passed in a function.
- Props are **immutable** so we cannot modify the props from inside the component. Inside the components, we can add attributes called props.
- These attributes are available in the component as **this.props** and can be used to render dynamic data in our render method.
- When you need immutable data in the component, you have to add props to **reactDom.render()** method in the **main.js** file of your ReactJS project and used it inside the component in which you need.

- Example
- **App.js**

```
import React, { Component } from 'react';
class App extends React.Component {
   render() {
      return (
         <div>
            <h1> Welcome to { this.props.name } </h1>
   <p> <h4> Welcome to React Learning. </h4> </p>
         </div>
      );
   }
}
export default App;
```

- **Main.js**

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

ReactDOM.render(<App name = "JavaScript DOM!!" />,
document.getElementById('app'));
```

Default Props

- It is not necessary to always add props in the **reactDom.render()** element. You can also set **default** props directly on the component constructor.
- Example: **App.js**

```
import React, { Component } from 'react';
class App extends React.Component {
   render() {
      return (
          <div>
              <h1>Default Props Example</h1>
        <h3>Welcome to {this.props.name}</h3>
  <p>Welcome to Learning.</p>
          </div>
        );
    }
}
App.defaultProps = {
   name: "JavaScriptLearn"
}
export default App;
```

Main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

ReactDOM.render(<App/>,
document.getElementById('app'));
```

State and Props

- It is possible to combine both state and props in your app. You can set the state in the parent component and pass it in the child component using props

```
import React, { Component } from 'react';
class App extends React.Component {
   constructor(props) {
      super(props);
      this.state = {
         name: "JavaScript-learn",
      }
   }
   render() {
      return (
         <div>
            <JTP jtpProp = {this.state.name}/>
         </div>
      );
   }
}
class JTP extends React.Component {
   render() {
      return (
          <div>
               <h1>State & Props Example</h1>
               <h3>Welcome to {this.props.jtpProp}</h3>
               <p>Happy Learning</p>
         </div>
      );
   }
}
export default App;
```

App.JS

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

ReactDOM.render(<App/>,
document.getElementById('app'));
```

Main.JS

React Props Validation

- Props are an important mechanism for passing the **read-only** attributes to React components.
- The props are usually required to use correctly in the component.
- If it is not used correctly, the components may not behave as expected.
- Hence, it is required to use **props validation** in improving react components.
- Props validation is a tool that will help the developers to avoid future bugs and problems. It is a useful way to force the correct usage of your components.
- React components used special property **PropTypes** that help you to catch bugs by validating data types of values passed through props, although it is not necessary to define components with propTypes.

Validating Props

- **App.propTypes** is used for props validation in react component.
- When some of the props are passed with an invalid type, you will get the warnings on JavaScript console.
- After specifying the validation patterns, you will set the **App.defaultProps**.
- Syntax:

```
class App extends React.Component {
        render() {}
}
Component.propTypes = { /*Definition */};
```

• ReactJS Props Validator

SN	PropsType	Description
1.	PropTypes.any	The props can be of any data type.
2.	PropTypes.array	The props should be an array.
3.	PropTypes.bool	The props should be a boolean.
4.	PropTypes.func	The props should be a function.
5.	PropTypes.number	The props should be a number.
6.	PropTypes.object	The props should be an object.
7.	PropTypes.string	The props should be a string.
8.	PropTypes.symbol	The props should be a symbol.
9.	PropTypes.instanceOf	The props should be an instance of a particular JavaScript class.
10.	PropTypes.isRequired	The props must be provided.
11.	PropTypes.element	The props must be an element.
12.	PropTypes.node	The props can render anything: numbers, strings, elements or an array (or fragment) containing these types.
13.	PropTypes.oneOf()	The props should be one of several types of specific values.
14.	PropTypes.oneOfType([PropTypes.string,PropTypes.number])	The props should be an object that could be one of many types.

- Example:
- Here, we are creating an App component which contains all the props that we need.
- In this example, **App.propTypes** is used for props validation.
- For props validation, you must have to add this line:
- **import PropTypes from 'prop-types'** in **App.js file**.

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';
class App extends React.Component {
   render() {
      return (
          <div>
<h1>ReactJS Props validation example</h1>
<table>
<tr>
 <th>Type</th>
 <th>Value</th>
  <th>Valid</th>
  </tr>
    <tr>
     <td>Array</td>
      <td>{this.props.propArray}</td>
      <td>{this.props.propArray ? "true" : "False"}</td>
      </tr>
      <tr>
     <td>Boolean</td>
     <td>{this.props.propBool ? "true" : "False"}</td>
     <td>{this.props.propBool ? "true" : "False"}</td>
     </tr>
     <tr>
     <td>Function</td>
     <td>{this.props.propFunc(5)}</td>
     <td>{this.props.propFunc(5) ? "true" : "False"}</td>
     </tr>
     <td>String</td>
    <td>{this.props.propString}</td>
     <td>{this.props.propString ? "true" : "False"}</td>
      </tr>
              <tr>
                 <td>Number</td>
                 <td>{this.props.propNumber}</td>
                 <td>{this.props.propNumber ? "true" : "False"}</td>
              </tr>
          </table>
      </div>
      );
  } }
```

```
App.propTypes = {
    propArray: PropTypes.array.isRequired,
    propBool: PropTypes.bool.isRequired,
    propFunc: PropTypes.func,
    propNumber: PropTypes.number,
    propString: PropTypes.string,
}
App.defaultProps = {
    propArray: [1,2,3,4,5],
    propBool: true,
    propFunc: function(x){return x+5},
    propNumber: 1,
    propString: "JavaScript-learning",
}
export default App;
```

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

ReactDOM.render(<App/>,
document.getElementById('app'));
```

Main.JS

- ReactJS Custom Validators
- ReactJS allows creating a custom validation function to perform custom validation.
- The following argument is used to create a custom validation function.

1. **props:** It should be the first argument in the component.
2. **propName:** It is the propName that is going to validate.
3. **componentName:** It is the componentName that are going to validated again.

```
var Component = React.createClass({
App.propTypes = {
   customProp: function(props, propName, componentName) {
        if (!item.isValid(props[propName])) {
          return new Error('Validation failed!');
        }
      }
   }
})
```

Difference between State and Props

SN	Props	State
1.	Props are read-only.	State changes can be asynchronous.
2.	Props are immutable.	State is mutable.
3.	Props allow you to pass data from one component to other components as an argument.	State holds information about the components.
4.	Props can be accessed by the child component.	State cannot be accessed by child components.
5.	Props are used to communicate between components.	States can be used for rendering dynamic changes with the component.
6.	Stateless component can have Props.	Stateless components cannot have State.
7.	Props make components reusable.	State cannot make components reusable.
8.	Props are external and controlled by whatever renders the component.	The State is internal and controlled by the React Component itself.

- The below table will guide you about the changing in props and state.

SN	Condition	Props	State
1.	Can get initial value from parent Component?	Yes	Yes
2.	Can be changed by parent Component?	Yes	No
3.	Can set default values inside Component?	Yes	Yes
4.	Can change inside Component?	No	Yes
5.	Can set initial value for child Components?	Yes	Yes
6.	Can change in child Components?	Yes	No

React Constructor

- The concept of a constructor is the same in React.
- The constructor in a React component is called before the component is mounted.
- When you implement the constructor for a React component, you need to call **super(props)** method before any other statement.
- If you do not call super(props) method, **this.props** will be undefined in the constructor and can lead to bugs.
- Syntax

```
Constructor(props){
    super(props);
}
```

- In React, constructors are mainly used for two purposes:

1. It used for initializing the local state of the component by assigning an object to **this.state**.
2. It used for binding event handler methods that occur in your component.

- You cannot call **setState()** method directly in the **constructor()**.
- If the component needs to use local state, you need directly to use '**this.state**' to assign the initial state in the constructor.
- The constructor only uses **this.state** to assign initial state, and all other methods need to use **set.state()** method.

- App.js

```
import React, { Component } from 'react';
class App extends Component {
  constructor(props){
    super(props);
    this.state = {
         data: 'www.javatpoint.com'
      }
    this.handleEvent = this.handleEvent.bind(this);
  }
  handleEvent(){
    console.log(this.props);
  }
  render() {
    return (
      <div className="App">
    <h2>React Constructor Example</h2>
    <input type ="text" value={this.state.data} />
        <button onClick={this.handleEvent}>Please Click</button>
      </div>
    );
  }
}
export default App;
```

main.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App.js';

ReactDOM.render(<App />,
document.getElementById('app'));
```

React Component API

- ReactJS component is a top-level API. It makes the code completely individual and reusable in the application. It includes various methods for:

1. Creating elements
2. Transforming elements
3. Fragments

- Here, we are going to explain the three most important methods available in the React component API.

1.setState()

2.forceUpdate()

3.findDOMNode()

- **setState()**
- This method is used to update the state of the component. This method does not always replace the state immediately.
- Instead, it only adds changes to the original state. It is a primary method that is used to update the user interface(UI) in response to event handlers and server responses.
- Syntax

```
this.stateState(object newState[, function callback]);
```

- **forceUpdate()**
- This method allows us to update the component manually.
- Syntax

```
Component.forceUpdate(callback);
```

- **findDOMNode()**
- For DOM manipulation, you need to use **ReactDOM.findDOMNode()** method. This method allows us to find or access the underlying DOM node.
- Syntax

```
ReactDOM.findDOMNode(component);
```