ReactJS - Learning Notes (Module-3)

We will cover below topic in this session:

- State
- Props
- State vs Props
- Component API
- Component Life Cycle
- Forms
- Events

ReactJS – State

State is the place where the data comes from. We should always try to make our state as simple as possible and minimize the number of stateful components. If we have, for example, ten components that need data from the state, we should create one container component that will keep the state for all of them.

Using Props

The following sample code shows how to create a stateful component using EcmaScript2016 syntax.

App.jsx

```
import React from 'react';

class App extends React.Component {

  constructor(props) {

    super(props);


    this.state = {

      header: "Test header..",

      content: "Test Contents...."

    }
```

```
  }
  render() {
    return (
      <div>
        <h1>{this.state.header}</h1>
        <h2>{this.state.content}</h2>
      </div>
    );
  }
}
export default App;
```

main.js

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

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

Props Overview

The main difference between state and props is that **props** are immutable. This is why the container component should define the state that can be updated and changed, while the child components should only pass data from the state using props.

Using Props

When we need immutable data in our component, we can just add props to **reactDOM.render()** function in **main.js** and use it inside our component.

App.jsx

```
import React from 'react';


class App extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.headerProp}</h1>
        <h2>{this.props.contentProp}</h2>
      </div>
    );
  }
}
export default App;
```

main.js

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


ReactDOM.render(<App headerProp = "Header from props..." contentProp = "Content
  from props..."/>, document.getElementById('app'));


export default App;
```

Default Props

You can also set default property values directly on the component constructor instead of adding it to the **reactDom.render()** element.

App.jsx

```
import React from 'react';


class App extends React.Component {
  render() {
    return (
      <div>
        <h1>{this.props.headerProp}</h1>
        <h2>{this.props.contentProp}</h2>
      </div>
    );
  }
}
App.defaultProps = {
  headerProp: "Header from props...",
  contentProp:"Content from props..."
}
export default App;
```

main.js

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


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

State and Props

The following example shows how to combine **state** and props in your app. We are setting the state in our parent component and passing it down the component tree using **props**. Inside the **render** function, we are setting **headerProp** and **contentProp** used in child components.

App.jsx

```
import React from 'react';


class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      header: "Header from props...",
      content: "Content from props..."
    }
  }
  render() {
    return (
      <div>
        <Header headerProp = {this.state.header}/>
        <Content contentProp = {this.state.content}/>
      </div>
    );
  }
}
class Header extends React.Component {
  render() {
    return (
```

```
      <div>
        <h1>{this.props.headerProp}</h1>
      </div>
    );
  }
}
class Content extends React.Component {
  render() {
    return (
      <div>
        <h2>{this.props.contentProp}</h2>
      </div>
    );
  }
}
export default App;
```

main.js

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

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

The result will again be the same as in the previous two examples, the only thing that is different is the source of our data, which is now originally coming from the **state**. When we want to update it, we just need to update the state, and all child components will be updated. More on this in the Events chapter.

Props Validation

Validating Props

In this example, we are creating **App** component with all the **props** that we need. **App.propTypes** is used for props validation. If some of the props aren't using the correct type that we assigned, we will get a console warning. After we specify validation patterns, we will set **App.defaultProps**.

App.jsx

```
import PropTypes from 'prop-types';
import React from 'react';
import ReactDOM from 'react-dom';


class App extends React.Component {
  render() {
    return (
      <div>
        <h1> Hello, {this.props.name} </h1>
        <h3>Array: {this.props.propArray}</h3>
        <h3>Bool: {this.props.propBool ? "True..." : "False..."}</h3>
        <h3>Func: {this.props.propFunc(3)}</h3>
        <h3>Number: {this.props.propNumber}</h3>
        <h3>String: {this.props.propString}</h3>
      </div>
    );
  }
}
App.propTypes = {
  name: PropTypes.string,
  propArray: PropTypes.array.isRequired,
  propBool: PropTypes.bool.isRequired,
```

```
  propFunc: PropTypes.func,
  propNumber: PropTypes.number,
  propString: PropTypes.string,
};
App.defaultProps = {
  name: Tech Vision',
  propArray: [1, 2, 3, 4, 5],
  propBool: true,
  propFunc: function(e) {
    return e
  },
  propNumber: 1,
  propString: "String value..."
}
export default App;
```

main.js

```
import React from 'react';
import PropTypes from 'prop-types';
import ReactDOM from 'react-dom';
import App from './App.jsx';

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

webpack.config.js

```
var config = {
  entry: './main.js',
```

```
  output: {
    path:'/',
    filename: 'index.js',
  },
  devServer: {
    inline: true,
    port: 8080
  },
  externals: {
    'react': 'React'
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /node_modules/,
        loader: 'babel-loader',

        query: {
          presets: ['es2015', 'react']
        }
      }
    ]
  }
}
```

```
module.exports = config;
```

Since all props are valid, we will get the following result.

As can be noticed, we have use **isRequired** when validating **propArray** and **propBool**. This will give us an error, if one of those two don't exist. If we delete **propArray: [1,2,3,4,5]** from the **App.defaultProps** object, the console will log a warning.

```
Warning: Failed propType: Required prop `propArray` was not specified in `App`.
```

If we set the value of **propArray: 1**, React will warn us that the propType validation has failed, since we need an array and we got a number.

```
Warning: Failed propType: Invalid prop `propArray` of type `number` supplied to
`App`, expected `array`.
```

Component API

In this chapter, we will explain React component API. We will discuss three methods: **setState(), forceUpdate** and **ReactDOM.findDOMNode()**. In new ES6 classes, we have to manually bind this. We will use **this.method.bind(this)** in the examples.

Set State

setState() method is used to update the state of the component. This method will not replace the state, but only add changes to the original state.

```
import React from 'react';


class App extends React.Component {

  constructor() {

    super();


    this.state = {

      data: []

    }
```

```
    this.setStateHandler = this.setStateHandler.bind(this);
  };
  setStateHandler() {
    var item = "setState..."
    var myArray = this.state.data.slice();
          myArray.push(item);
    this.setState({data: myArray})
  };
  render() {
    return (
      <div>
        <button onClick = {this.setStateHandler}>SET STATE</button>
        <h4>State Array: {this.state.data}</h4>
      </div>
    );
  }
}
export default App;
```

We started with an empty array. Every time we click the button, the state will be updated. If we click five times, we will get the following output.

Force Update

Sometimes we might want to update the component manually. This can be achieved using the **forceUpdate()** method.

```
import React from 'react';
```

```
class App extends React.Component {
  constructor() {
    super();
    this.forceUpdateHandler = this.forceUpdateHandler.bind(this);
  };
  forceUpdateHandler() {
    this.forceUpdate();
  };
  render() {
    return (
      <div>
        <button onClick = {this.forceUpdateHandler}>FORCE UPDATE</button>
        <h4>Random number: {Math.random()}</h4>
      </div>
    );
  }
}
export default App;
```

Find Dom Node

For DOM manipulation, we can use **ReactDOM.findDOMNode()** method. First we need to import **react-dom**.

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

class App extends React.Component {
```

```
  constructor() {
    super();
    this.findDomNodeHandler = this.findDomNodeHandler.bind(this);
  };
  findDomNodeHandler() {
    var myDiv = document.getElementById('myDiv');
    ReactDOM.findDOMNode(myDiv).style.color = 'green';
  }
  render() {
    return (
      <div>
        <button onClick = {this.findDomNodeHandler}>FIND DOME NODE</button>
        <div id = "myDiv">NODE</div>
      </div>
    );
  }
}
export default App;
```

Note − Since the 0.14 update, most of the older component API methods are deprecated or removed to accommodate ES6.

Component Life Cycle

Lifecycle Methods

- **componentWillMount** is executed before rendering, on both the server and the client side.
- **componentDidMount** is executed after the first render only on the client side. This is where AJAX requests and DOM or state updates should occur. This method is also used for integration with other JavaScript frameworks and any functions with delayed execution such

as **setTimeout** or **setInterval**. We are using it to update the state so we can trigger the other lifecycle methods.

- **componentWillReceiveProps** is invoked as soon as the props are updated before another render is called. We triggered it from **setNewNumber** when we updated the state.
- **shouldComponentUpdate** should return **true** or **false** value. This will determine if the component will be updated or not. This is set to **true** by default. If you are sure that the component doesn't need to render after **state** or **props** are updated, you can return **false** value.
- **componentWillUpdate** is called just before rendering.
- **componentDidUpdate** is called just after rendering.
- **componentWillUnmount** is called after the component is unmounted from the dom. We are unmounting our component in **main.js**.

In the following example, we will set the initial **state** in the constructor function. The **setNewnumber** is used to update the **state**. All the lifecycle methods are inside the Content component.

App.jsx

```
import React from 'react';


class App extends React.Component {
  constructor(props) {
    super(props);


    this.state = {
      data: 0
    }
    this.setNewNumber = this.setNewNumber.bind(this)
  };
  setNewNumber() {
    this.setState({data: this.state.data + 1})
```

```
  }
  render() {
    return (
      <div>
        <button onClick = {this.setNewNumber}>INCREMENT</button>
        <Content myNumber = {this.state.data}></Content>
      </div>
    );
  }
}
class Content extends React.Component {
  componentWillMount() {
    console.log('Component WILL MOUNT!')
  }
  componentDidMount() {
    console.log('Component DID MOUNT!')
  }
  componentWillReceiveProps(newProps) {
    console.log('Component WILL RECIEVE PROPS!')
  }
  shouldComponentUpdate(newProps, newState) {
    return true;
  }
  componentWillUpdate(nextProps, nextState) {
    console.log('Component WILL UPDATE!');
  }
```

```
  componentDidUpdate(prevProps, prevState) {
    console.log('Component DID UPDATE!')
  }
  componentWillUnmount() {
    console.log('Component WILL UNMOUNT!')
  }
  render() {
    return (
      <div>
        <h3>{this.props.myNumber}</h3>
      </div>
    );
  }
}
export default App;
```

main.js

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

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

setTimeout(() => {
  ReactDOM.unmountComponentAtNode(document.getElementById('app'));}, 10000);
```

Only **componentWillMount** and **componentDidMount** will be logged in the console, since we didn't update anything yet.

```
Component WILL MOUNT!
Component DID MOUNT!
```

When we click the **INCREMENT** button, the update will occur and other lifecycle methods will be triggered.

```
Component WILL RECIEVE PROPS!
Component WILL UPDATE!
Component DID UPDATE!
```

After ten seconds, the component will unmount and the last event will be logged in the console.

```
Component WILL UNMOUNT!
```

Note – Lifecycle methods will always be invoked in the same order so it is a good practice to write it in the correct order as shown in the example.

ReactJS – Forms

Example

In the following example, we will set an input form with **value = {this.state.data}**. This allows to update the state whenever the input value changes. We are using **onChange** event that will watch the input changes and update the state accordingly.

App.jsx

```
import React from 'react';


class App extends React.Component {
  constructor(props) {
    super(props);


    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
```

```
  };
  updateState(e) {
    this.setState({data: e.target.value});
  }
  render() {
    return (
      <div>
        <input type = "text" value = {this.state.data}
          onChange = {this.updateState} />
        <h4>{this.state.data}</h4>
      </div>
    );
  }
}
export default App;
```

main.js

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

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

Example

In the following example, we will see how to use forms from child component. **onChange** method will trigger state update that will be passed to the child input **value** and rendered on the screen. A similar

example is used in the Events chapter. Whenever we need to update state from child component, we need to pass the function that will handle updating (**updateState**) as a prop (**updateStateProp**).

App.jsx

```
import React from 'react';


class App extends React.Component {
  constructor(props) {
    super(props);


    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
  };
  updateState(e) {
    this.setState({data: e.target.value});
  }
  render() {
    return (
      <div>
        <Content myDataProp = {this.state.data}
          updateStateProp = {this.updateState}></Content>
      </div>
    );
  }
}
```

```
class Content extends React.Component {
  render() {
    return (
      <div>
        <input type = "text" value = {this.props.myDataProp}
          onChange = {this.props.updateStateProp} />
        <h3>{this.props.myDataProp}</h3>
      </div>
    );
  }
}
export default App;
```

main.js

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

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

ReactJS – Events

Example

This is a simple example where we will only use one component. We are just adding **onClick** event that will trigger **updateState** function once the button is clicked.

App.jsx

```
import React from 'react';

class App extends React.Component {
```

```
  constructor(props) {
    super(props);

    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
  };
  updateState() {
    this.setState({data: 'Data updated...'})
  }
  render() {
    return (
      <div>
        <button onClick = {this.updateState}>CLICK</button>
        <h4>{this.state.data}</h4>
      </div>
    );
  }
}
export default App;
```

main.js

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

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

Child Events

When we need to update the **state** of the parent component from its child, we can create an event handler (**updateState**) in the parent component and pass it as a prop (**updateStateProp**) to the child component where we can just call it.

App.jsx

```
import React from 'react';


class App extends React.Component {
  constructor(props) {
    super(props);


    this.state = {
      data: 'Initial data...'
    }
    this.updateState = this.updateState.bind(this);
  };
  updateState() {
    this.setState({data: 'Data updated from the child component...'})
  }
  render() {
    return (
      <div>
        <Content myDataProp = {this.state.data}
          updateStateProp = {this.updateState}></Content>
      </div>
```

```
    );
  }
}
class Content extends React.Component {
  render() {
    return (
      <div>
        <button onClick = {this.props.updateStateProp}>CLICK</button>
        <h3>{this.props.myDataProp}</h3>
      </div>
    );
  }
}
export default App;
```

main.js

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

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