React - Forms, refs and keys

Agenda

- Creating Forms in React
- Controlled and Uncontrolled Forms
- Handling Forms events
- Form Validation & Error handling
- Using Refs
- Using Keys

Forms

Forms in React are different than plain HTML forms. A component has its data in its state and/or props received from parent. So we need to design our Form component that reads data from its state and component should be aware of the events (i.e. focus, unfocus, clicks, keypress etc) and update its state accordingly.

Based on how we design our Form we can have two different type of forms

Uncontrolled and **Controlled** forms.

Uncontrolled Forms

Uncontrolled forms are similar to the tradition HTML.

```
class Form extends Component {
    render( ) {
        return (
            <form>
                <input type=”text” />
            </form>
        )
    }
}
```

Controlled Forms

A controlled input accepts its current value as a prop, as well as a callback to change that value. You could say it's a more "React way" of approaching this.

```
<input value={someValue} onChange={handleChange} />
```

the value of this input (**`someValue`**) has to live in the state somewhere. Typically, the component that renders the input (Form component) saves that in its state

Controlled Forms cond..

```
class Form extends Component {
     constructor(){
          super()
          This.state = { name: ‘John’ }
     }
     render () {
          return (
               <input type=”text” value={this.state.name} />
          )
     }
}
```

Handling Form Events

We can handle form events by providing a handler callback to our controlled inputs. In below example we are checking what user is typing and then converting it into upper case.

```
class Form extends Component {
     constructor(){
          super()
          This.state  = { name: ‘’ }
          this.handleChange = this.handleChange.bind(this)
     }
     handleChange(event){
          this.setState({ name: event.target.value.toUpperCase()})
     }
     render () {
          return (
               <input type=”text” value={this.state.name} onChange={this.handleChange} />
          )
     }
}
```

Form validation and error handling

In below example, we're going to create a simple signup form and onSubmit we're going to validate if user has entered 'name', password and confirm password should be same. .

```
import React, { Component } from 'react'

export default class Form extends Component {
  constructor(){
    super()
    this.state = {
      name: '',
      pass: '',
      cnf_pass: '',
      message: ''
    }
  }
```

example

```
handleChange(e) {
    let state = {}
    state[e.target.name] = e.target.value
    this.setState(state)
  }
  handleSubmit (event) {
    event.preventDefault();
    if(!this.state.name || !this.state.pass || !this.state.cnf_pass){
      this.setState({message: 'All fields are required'})
    }
    else if(this.state.pass !== this.state.cnf_pass){
      this.setState({message: 'passwords do not match!'})
    }
    else {
      this.setState({message: "You're registered"})
    }
  }
```

example

```
render () {
    return (
      <form onSubmit={this.handleSubmit.bind(this)}>
        <p>
          { this.state.message }
        </p>
        <input type='text' name='name' onChange={this.handleChange.bind(this)}
value={this.state.name} placeholder="name"/>
        <input type='password' name='pass' onChange={this.handleChange.bind(this)}
value={this.state.pass} placeholder="password"/>
        <input type='password' name='cnf_pass' onChange={this.handleChange.bind(this)}
value={this.state.cnf_pass} placeholder="confirm your password"/>
        <input type="submit" value="Signup"/>
      </form>
    )
  }
}
```

Using Refs

The **ref** is used to return a reference to your element. **Refs** should be avoided in most cases but they can be useful when you need DOM measurements or to add methods to your components.

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

export default class RefDemo extends React.Component {

   constructor(props) {
      super(props);

      this.state = {
         data: ''
      }

      this.updateState = this.updateState.bind(this);
      this.clearInput = this.clearInput.bind(this);
   };
```

```
updateState(e) {
      this.setState({data: e.target.value});
   }

   clearInput() {
      this.setState({data: ''});
      ReactDOM.findDOMNode(this.refs.myInput).focus();
   }

   render() {
      return (
         <div>
            <input value = {this.state.data} onChange = {this.updateState}
               ref = "myInput"></input>
            <button onClick = {this.clearInput}>CLEAR</button>
            <h4>{this.state.data}</h4>
         </div>
      );
   }
}
```

Keys

React **keys** are useful when working with dynamically created components or when your lists are altered by users. Setting the **key** value will keep your components uniquely identified after the change.

Let’s create a list of users dynamically

```
class Keys extends Component {
   constructor(props) {
      super(props);
      this.state = {
         users: ['Chandler', 'Monica', 'Ross', 'Pheobe', 'Rachel']
      }
   }
   render() {
     let userList  =  this.state.users.map((user,i) => {
       return <li key={i}> {user} </li>
     })
      return (
         <div>
            {userList}
         </div>
      );
   }
}
```

Keys contd..

If we add or remove some elements in the future or change the order of the dynamically created elements, React will use the key values to keep track of each element.

Exercise

Create a cart application having a form component to add new items in cart, a itemlist component to display added items and CartTotal to show total amount.