Forms, Events and Refs

DEVELOPING APPLICATIONS USING REACTJS

Objectives

- To understand how to create controlled and uncontrolled form components in forms
- To be able to attach events to elements in ReactJS
- To be able to use refs appropriately
- To be able to pass state to child components

Forms and React

- Forms inherently keep some internal state and therefore React has to work differently with them
  - A standard form with a submit button would work in React out of the box (as shown below)
  - Usually want to have access to the form values that have been submitted
  - Controlled components are the recommended way to achieve this

```
<form>
  <label>
     Name:
     <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

3

Controlled Components

- Form elements usually maintain their own state and it is changed through user input
- React component's mutable state is kept in the state property and can only be changed by calling `setState()`
- React makes a "single source of truth"
  - React component that renders a form also controls what happens on the form in subsequent user input
  - Form input elements whose value is controlled by React in this way is called a CONTROLLED COMPONENT

4

Collecting Data from Forms

- Data needs to be collected from the form before input loses focus
  - It actually needs to be collected each time it changes
    - Otherwise it is difficult to make function elegantly
- State in the component containing the form can be used to hold the data until it is passed back to the application
- Each input element needs to call a 'handleChange' function, passing in its name so that its value can be retrieved and saved

```
…
    <input type=text name="myName" value={this.state.myName}
       onChange={this.handleChange("myName")} />
…
```

5

Collecting Data from Forms

- An anonymous function in the constructor can then be used to set the state as required using the valueName passed in

```
this.handleChange = (valueName) => {
  return (event) =>
      { this.setState({ [valueName]: event.target.value });};
}
```

6

Collecting Data from Forms

- Once the data has been collected on the form, the onSubmit event can handle passing data back to the component's parent

```
…
     <form id="addMyName"
       onSubmit={this.handleSubmit.bind(this)} >
…
```

7

Collecting Data from Forms

- The function handleSubmit also needs to be in the form component's class – it calls the onSubmit in the parent
  - Current state is passed in and then reset to clear the form

```
this.handleSubmit = (event) => {
        event.preventDefault();
        this.props.onSubmit(this.state);
        this.setState({
                myName: ''
        });
}
```

8

Collecting Data from Forms

- In the parent component, the form's tag and an onSubmit should be used to call the submit handler in the parent
  - The arguments that were provided in the form component are automatically passed through the the handler

```
…
     <FormComponent
    onSubmit={this.handleSubmitInParent} />
…
```

  - This will place the FormComponent in the app and ensure that its submission is handled
- handleSubmitInParent() can then handle the data
  - In this example, the new data is pushed into an existing array and then output to localStorage as JSON
  - Anything that JavaScript can do with data can now be done!

9

Collecting Data from Forms

- handleSubmitInParent code:

```
this.handleSubmitInParent = dataFromForm) => {
        let array = this.state.array;
        array.push( {
                id : array.length,
                name: dataFromForm.myName
        });

        this.setState({array});
        localStorage.array = JSON.stringify(array);
}
```

10

EXAMPLE FORM - CONTROLLED COMPONENTS

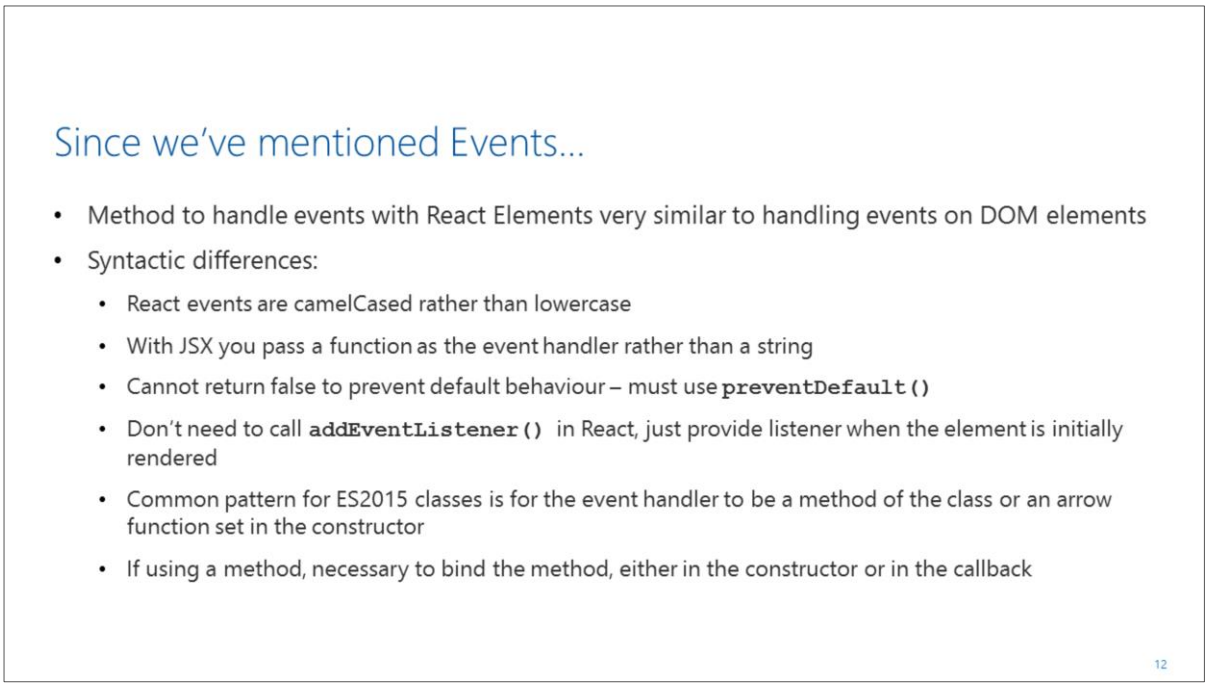
```
import React from 'react';

export default class App extends React.Component {

    constructor(props) {
      super(props);

      this.state = {
        data: 'Initial data...'
      }
      this.updateState = e => {
        this.setState({data: e.target.value});
    }

render() {
        return (
            <form>
              <input type="text" value={this.state.data}
               onChange = {this.updateState} />
              <h4>{this.state.data}</h4>
            </form>
        );
    }
}
```

The event handler has been declared as an arrow function. This preserves the context of this to the component rather than the caller.

11

In the code example above, when the user types into the text box, the text displayed in the <h4> element is automatically updated to reflect what has been typed. The value of the text box is set to be the current state of data, but when the text box value changes, the onChange method is triggered which then subsequently calls setState to set data to whatever the value in the text box has been changed to.

onChange is obviously an event and the handler function for this is called updateState. Notice that the call to this function in the input element has .bind(this) added to it. This is necessary so the reference to the component is maintained when calling the handler function and the use of this.setState uses the component and not the calling element. The event object is passed to the handler so that the value of the element that raised the event can be accessed through it.



The React Documentation explain the meaning of this as follows:

You have to be careful about the meaning of this in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind this.handleClick and pass it to onClick, this will be undefined when the function is actually called.

This is not React-specific behavior; it is a part of how functions work in JavaScript. Generally, if you refer to a method without () after it, such as onClick={this.handleClick}, you should bind that method.

If calling bind annoys you, there are two ways you can get around this. If you are using the experimental property initialiser syntax, you can use property initializers to correctly bind callbacks.

If you aren't using property initializer syntax, you can use an arrow function in the callback.

The problem with this syntax is that a different callback is created each time the LoggingButton renders. In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering. We generally recommend binding in the constructor to avoid this sort of performance problem.

Back to forms...

- Controlled components are recommended for for implementation
  - Form data is handled by a React component
- Uncontrolled components let form data be handled by the DOM
  - Instead of an event handler for every state, ref can be used to get the values from a form
  - "Source of truth" kept in the DOM
    - Sometimes easier to integrate React and non-React code
    - Can be slightly less code

13

UNCONTROLLED COMPONENTS EXAMPLE

```
import React from 'react';

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

    handleSubmit(e) {
      alert("A name was submitted" + this.input.value);
      e.preventDefault();
    }
    render() {
        return (
            <form onSubmit={this.handleSubmit.bind(this)}>
               <label> Name:
                       <input
                          type="text" ref={(input) => this.input=input}
                       />
               </label>
               <input type="submit" value="Submit" />
            </form>
        );
    }
}
```

In this code, the event handler has been written as a method of the class.

In this case .bind(this) has to be added to the function to preserve the context of this.

Refs are also used here...

14

refs

- ref is used to return a reference to an element
  - Can be useful when you need DOM measurements or to add methods to components
- Imagine a form with an input element and you want to clear it with a button click and reset the focus to it
  - How could we get the app to recognise which DOM Node to clear?
    - refs could be used here:

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

15

refs

- The `ref` defined is now used as part of the `clearInput()` method defined in the class:
  - It is used as the component argument for **`findDOMNode()`** to return focus to the input when cleared

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

16

ref Callback Attribute

- ref can be attached to any component
  - It can take a callback function executed immediately after component is mounted or unmounted
  - Callback receives underlying DOM element as argument when used on an HTML element

```
<input
      type="text"
      ref= {(input) => {this.textInput = input;}}
/>
```

  - In this example, the callback is used to store a reference to a DOM node
- ref callback called with the DOM element when the component mounts
  - Called with null when it unmounts
- Recommended pattern for for accessing DOM elements

17

refs and custom components

- If used in a custom component, callback receives mounted instance of component as its argument

```
class App extends React.Component {
   componentDidMount() {
      this.textInput.focus();
   }
   render() {
      return(
         <CustomComponent
            ref= {(input) => {this.textInput = input;}}
         />
      );
   }
}
```

- This would simulate the CustomComponent being clicked immediately after mounting

18

refs and functional components

- ref attribute cannot be used on functional components as there is no instance
  - Can use ref inside the render function of a functional component

```
const CustomComponent = (props) => {
    // textInput must be declared here so the ref callback can
    // refer to it
    let textInput = null;
    const handleClick = () => {textInput.focus()};

    return (
        <div>
          <input type="text"
               ref= {(input) => {textInput = input;}} />
          <input type="button" value="Focus Input"
          onClick={handleClick} />
        </div>
    );
}
```

19

Other notes of ref

- refs should be avoided as much as possible
  - Could be inclined to use them to make things happen in an app
    - Probably need to re-evaluate state ownership
    - Often proper place to own state is in a higher level in component hierarchy
- ref attribute cannot be used on functional components
  - There is no instance
  - Can use ref inside the render function of a functional component

20

Default Values and Uncontrolled Components

- Value in DOM is overridden by `value` attribute on form elements in React rendering lifecycle
  - Often want React to specify initial value and leave subsequent updates uncontrolled
  - To do this, specify a **`default value`** rather than a **`value`**

```
render() {
   return (
        <form onSubmit={this.handleSubmit.bind(this)}>
           <label> Name:
                <input defaultValue="Jane" type="text"
                       ref={(input) => this.input=input} />
           </label>
           <input type="submit" value="Submit" />
        </form>
   );
}
```

  - **If using checkboxes or radio buttons `defaultChecked` can be used**

21

Forms as Child Components

- Forms can be used as a child component of another components
- State updates can be triggered and passed to the child input value and rendered
  - Updating state from child component is done by passing function handling updating as a prop

```
// In parent render…
…
    return (<Content myDataProp = {this.state.data}
                updateStateProp = {this.updateState.bind(this)} />);
…
```

```
// In child render…
…
    return (
        <input type=text value={this.props.myDataProp}
                onChange={this.props.updateStateProp} />);
…
```

22

The same principles of passing state as props into a form as a child component can be applied to any component.

Objectives

- To understand how to create controlled and uncontrolled form components in forms
- To be able to attach events to elements in ReactJS
- To be able to use refs appropriately
- To be able to pass state to child components

Exercise time!

- EG07 – Forms and Events

24