React Forms

Goals

- Build forms with React
- Understand what controlled components are

Forms

- HTML form elements work differently than other DOM elements in React
  - Form elements naturally keep some internal state.
  - For example, this form in plain HTML accepts a single name:

```
<form>
  <label for="fullname">Full Name:</label>
  <input name="fullname" />
  <button>Add!</button>
</form>
```

Thinking About State

```
<form>
  <label for="fullname">Full Name:</label>
  <input name="fullname" />
  <button>Add!</button>
</form>
```

- It's convenient to have a JS function that
  - handles the submission of the form *and*
  - has access to the data the user entered.
- The technique to get this is *controlled components*.

Controlled Components

- In HTML, form elements such as ***<input>***, ***<textarea>***, and ***<select>*** typically maintain their own state and update it based on user input.
- In React, mutable state is kept in the ***state*** of components, and only updated with ***setState()***.
- How do we use React to control form input state?

One Source of Truth

- We make the React state be the "single source of truth"

- React controls:
  - What is *shown* (the value of the component)
  - What happens the user types *(this gets kept in state)*
- Input elements controlled in this way are called “controlled components”.

Example Form Component

```
class NameForm extends Component {
  constructor(props) {
    super(props);
    // default fullName is an empty string
    this.state = { fullName: '' };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleSubmit(evt) {
    // do something with form data
  }
  handleChange(evt) {
    // runs on every keystroke event
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label for="fullname">Full Name:</label>
        <input name="fullname" value={this.state.fullName}
          onChange={this.handleChange}
        />
        <button>Add!</button>
      </form>
    );
  }
}
```

How the Controlled Form Works

- Since value attribute is set on element, displayed value will always be ***this.state.fullName*** — making the React state the source of truth.
- Since ***handleChange*** runs on every keystroke to update the React state, the displayed value will update as the user types.
- With a controlled component, every state mutation will have an associated handler function. This makes it easy to modify or validate user input.

handleChange Method

Here is the method that updates state based on input.

```
class NameForm extends Component {
  // ...

  handleChange(evt) {
```

```
    // runs on every keystroke
    this.setState({
      fullName: evt.target.value
    });
  }

  // ...
}
```

Handling Multiple Inputs

ES2015 Review

- ES2015 introduced a few object enhancements…
- This includes the ability to create objects with dynamic keys based on JavaScript expressions.
- The feature is called ***computed property names***.

Computed Property Names

ES5

```
var catData = {};
var microchip = 1432345421
catData[microchip] = "Blue Steele";
```

ES2015

```
let microchip = 1432345421;
let catData = {
    // propery computed inside the object literal
    [microchip]: "Blue Steele"
};
```

Application To React Form Components

Instead of making a separate ***onChange*** handler for every single input, we can make on generic function for multiple inputs!

Handling Multiple Inputs

To handle multiple controlled inputs, add the HTML ***name*** attribute to each JSX input element and let handler function decide the appropriate key in state to update based on ***event.target.name***.

```
class YourComponent extends Component {
  // ...

  handleChange(evt) {
    this.setState({
      [evt.target.name]: evt.target.value
    });
  }
```

```
  // ...
}
```

- Using this method, the keys in state have to match the input `name` attributes exactly.

The state:

```
this.state = { firstName: "", lastName: "" };
```

demo/name-form-demo/src/NameForm.js

```
class NameForm extends Component {  // ...
  handleChange(evt) {
    this.setState({ [evt.target.name]: evt.target.value });
  }

  render() {
    return (
        <form onSubmit={this.handleSubmit}>

          <label htmlFor="firstName">First:</label>
          <input id="firstName" name="firstName"
                 value={this.state.firstName}
                 onChange={this.handleChange} />

          <label htmlFor="lastName">Last:</label>
          <input id="lastName" name="lastName"
                 value={this.state.lastName}
                 onChange={this.handleChange} />
          <button>Add a new person!</button>

        </form>
    );
  }
} // end
```

Passing Data Up to a Parent Component

In React we generally have downward data flow. “Smart” parent components with simpler child components.

- But it is common for form components to manage their own state…
- But the smarter parent component usually has a ***doSomethingOnSubmit*** method to update its state after the form submission…
- So what happens is the parent will pass its ***doSomethingOnSubmit*** method down as a prop to the child.
- The child component will call this method which will then update the parent’s state.
- The child is still appropriately “dumber”, all it knows is to pass its data into a function it was given.

Shopping List Example

- Parent Component: ShoppingList (manages a list of shopping items)
- Child Component: NewListItemForm (a form to add a new shopping item to the list)

demo/shopping-list/src/ShoppingList.js

```
class ShoppingList extends Component {
  /** Add new item object to cart. */
  addItem(item) {
    let newItem = { ...item, id: uuid() };
    this.setState(state => ({
      items: [...state.items, newItem]
    }));
  }

  render() {
   return (
    <div className="ShoppingList">
      <NewListItemForm addItem={this.addItem}/>
      {this.renderItems()}
    </div>
   );
  }
}
```

demo/shopping-list/src/NewListItemForm.js

```
class NewListItemForm extends Component {
  /** Send {name, quantity} to parent
   *    & clear form. */

  handleSubmit(evt) {
    evt.preventDefault();
    this.props.addItem(this.state);
    this.setState({ name: "", qty: 0 });
  }

}
```

Keys and UUIDs

Using UUID for Unique Keys

- We've seen that using an iteration index as a ***key*** prop is a bad idea
- No natural unique key? Use a library to create a *uuid*
- Universally unique identifier (UUID) is a way to uniquely identify info
- Install it using `npm install uuid`

Using the UUID Module

demo/shopping-list/src/ShoppingList.js

```
import uuid from 'uuid/v4';
```

demo/shopping-list/src/ShoppingList.js

```
class ShoppingList extends Component {
  renderItems() {
    return (
      <ul>
        {this.state.items.map(item => (
          <li key={item.id}>
            {item.name}:{item.qty}
          </li>
        ))}
      </ul>
    );
  }

}
```

Uncontrolled components

- You will almost never use it
- Some inputs and external libraries require it.

Validation

- Useful for UI
- **Not an alternative to server side validation**
- Formik <https://jaredpalmer.com/formik/docs/overview>

Looking Ahead

Coming Up

- Lifecycle methods
- AJAX with React