React Forms

Springboard

Download Demo Code

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 the function returned to **useState()**.
- 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".

How the Controlled Form Works

- Since value attribute is set on element, displayed value will always be **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.

```
const NameForm = () => {
  // ...

  const handleChange = (evt) => {
    setFullName(evt.target.value);
  }

  // ...
}
```

Thinking about labels

- Our **<label>** tags have an important attribute called **for**
- If we give our label attribute a **for** attribute that matches with an id of an input, we can click on that label and be autofocused in the input
- This is a nice user experience and is very helpful for accessibility
- But there's a problem here!

htmlFor instead

- **for** is a reserved word in JavaScript, just like **class** is!
- The same way we replaced **class** with **className**, we need to replace **for** with **htmlFor**
- You will get warnings in the console if you forget this

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 instructorData = {};
var instructorCode = "elie";
instructorData[instructorCode] = "Elie Schoppik";
```

ES2015

```
let instructorCode = "elie";
let instructorData = {
    // property computed inside the object literal
    [instructorCode]: "Elie Schoppik"
};
```

Application To React Form Components

Instead of making a separate **onChange** handler for every single input, we can make a 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**.

```
const YourComponent = () => {
  // ...

  const handleChange = evt => {
    const { name, value } = evt.target;
    setFormData(fData => ({
      ...fData,
      [name]: value
    }));
  }

  // ...
}
```

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

The state:

```
{ firstName: "", lastName: "" };
```

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

```
  const [formData, setFormData] = useState({
    firstName: "",
    lastName: ""
  });

  const handleChange = evt => {
    const { name, value } = evt.target;
    setFormData(fData => ({
      ...fData,
      [name]: value
    }));
  };
```

Note: Remember the event target

React will forget about the event object after the handler runs, for performance reasons. This can be a problem when you use the callback pattern to set state, since setting state isn't synchronous. This is why we grab what we need from the event first:

```
const { name, value } = evt.target;
```

If you forget to do this and you use the callback pattern, React will throw errors because ***evt.target*** will be null inside of your callback!

For more on this, check out the React docs.

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 calls this method, updating 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

```
  /** Add new item object to cart. */
  const addItem = item => {
    let newItem = { ...item, id: uuid() };
    setItems(items => [...items, newItem]);
  };
```

demo/shopping-list/src/NewListItemForm.js

```
  /** Send {name, quantity} to parent
   *    & clear form. */

  const handleSubmit = evt => {
    evt.preventDefault();
    addItem(formData);
    setFormData(INITIAL_STATE);
  };
```

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 { v4 as uuid } from 'uuid';
  /** Add new item object to cart. */
  const addItem = item => {
    let newItem = { ...item, id: uuid() };
    setItems(items => [...items, newItem]);
  };
```

demo/shopping-list/src/ShoppingList.js

```
const renderItems = () => (
  <ul>
    {items.map(item => (
      <li key={item.id}>
        {item.name}: {item.qty}
      </li>
    ))}
  </ul>
);
```

Uncontrolled components

- If React is *not* in control of the form state, this is called an *uncontrolled component*.
- Some inputs and external libraries require it.

Validation

- Useful for UI
- **Not an alternative to server side validation**
- Formik

Testing Forms

- To test typing in form inputs, we can use ***fireEvent.change***
- When using this, we'll need to mock ***evt.target.value*** - this is how we'll tell React testing library what to place in the input
- For controlled components, state will then automatically update

Testing Forms: An Example

demo/shopping-list/src/ShoppingList.test.js

```
it("can add a new item", function() {
  const { getByLabelText, queryByText } = render(<ShoppingList />);

  // no items yet
  expect(queryByText("ice cream: 100")).not.toBeInTheDocument();

  const nameInput = getByLabelText("Name:");
  const qtyInput = getByLabelText("Qty:");
  const submitBtn = queryByText("Add a new item!")

  // fill out the form
  fireEvent.change(nameInput, { target: { value: "ice cream" }});
  fireEvent.change(qtyInput, { target: { value: 100 }});
  fireEvent.click(submitBtn);

  // item exists!
  expect(queryByText("ice cream: 100")).toBeInTheDocument();
});
```

Looking Ahead

Coming Up

- ***useEffect***
- AJAX with React