BONUS: React Forms Abstraction

(https://github.com/learn-co-curriculum/react-hooks-forms-abstraction) (https://github.com/learn-co-curriculum/react-hooks-forms-abstraction/issues/new)

Learning Goals

- Make our form logic more reusable by creating a dynamic `onChange` event handler

Code Along

If you want to code along there is starter code in the `src` folder. Make sure to run `npm install && npm start` to see the code in the browser.

Form State

Let's talk about the `onChange` event we had set up in the initial version of our `Form` component. If we look at the original code:

```
import React, { useState } from "react";

function Form() {
  const [firstName, setFirstName] = useState("Sylvia");
  const [lastName, setLastName] = useState("Woods");

  function handleFirstNameChange(event) {
    setFirstName(event.target.value);
  }

  function handleLastNameChange(event) {
    setLastName(event.target.value);
  }
```

```
    return (
      <form>
        <input type="text" onChange={handleFirstNameChange} value={firstName} />
        <input type="text" onChange={handleLastNameChange} value={lastName} />
        <button type="submit">Submit</button>
      </form>
    );
  }

  export default Form;
```

We can imagine that adding more input fields to this form is going to get repetitive pretty fast. For every new input field, we'd need to add:

- a new state variable by calling `useState()` to hold the value of that input
- a new `handleChange` function to update that piece of state

As a first refactor, let's use `useState` just once, and make an **object** representing all of our input fields. We will also need to update our `onChange` handlers and the variable names in our JSX accordingly:

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

    function handleFirstNameChange(event) {
      setFormData({
        ...formData,
        firstName: event.target.value,
      });
    }

    function handleLastNameChange(event) {
```

```
      setFormData({
        ...formData,
        lastName: event.target.value,
      });
    }

    return (
      <form>
        <input
          type="text"
          onChange={handleFirstNameChange}
          value={formData.firstName}
        />
        <input
          type="text"
          onChange={handleLastNameChange}
          value={formData.lastName}
        />
      </form>
    );
  }
```

Since our initial state is an *object*, in order to update state in our `onChange` handlers, we have to copy all the key/value pairs from the current version of that object into our new state — that's what the spread operator here is doing:

```
setFormData({
  // formData is an object, so we need to copy all the key/value pairs
  ...formData,
  // we want to overwrite the lastName key with a new value
  lastName: event.target.value,
});
```

Now, we just have one object in state to update whenever an input field changes.

Our change handlers are still a bit verbose, however. Since each one is changing a different value in our state, we've got them separated here. You can imagine that once we've got a more complicated form, this approach may result in a very cluttered component.

Instead of writing separate functions for each input field, we could actually condense this down into one more reusable function. Since `event` is being passed in as the argument, we have access to some of the `event.target` attributes that may be present.

If we give our inputs `name` attributes, we can access them as `event.target.name` :

```
<input
  type="text"
  name="firstName"
  value={formData.firstName}
  onChange={handleFirstNameChange}
/>
<input
  type="text"
  name="lastName"
  value={formData.lastName}
  onChange={handleLastNameChange}
/>
```

As long as the `name` attributes of our `<input>` fields match the keys in our state, we can write a generic `handleChange` function like so:

```
function handleChange(event) {
  // name is the KEY in the formData object we're trying to update
  const name = event.target.name;
  const value = event.target.value;

  setFormData({
    ...formData,
    [name]: value,
  });
}
```

Then, if we connect this new function to both of our `input` s, they will both correctly update state. Why? Because for the first `input`, `event.target.name` is set to `firstName`, while in the second `input`, it is set to `lastName`. Each `input`'s `name` attribute will change which part of state is actually updated!

Now, if we want to add a new input field to the form, we just need to add two things:

- a new key/value pair in our `formData` state, and
- a new `<input>` field where the `name` attribute matches our new key

We can take it one step further, and also handle `checkbox` inputs in our `handleChange` input. Since checkboxes have a `checked` attribute instead of the `value` attribute, we'd need to check what `type` our input is in order to get the correct value in state.

Here's what the final version of our `Form` component looks like:

```
import React, { useState } from "react";

function Form() {
  const [formData, setFormData] = useState({
    firstName: "Sylvia",
    lastName: "Woods",
    admin: false,
  });

  function handleChange(event) {
    const name = event.target.name;
    let value = event.target.value;

    // use `checked` property of checkboxes instead of `value`
    if (event.target.type === "checkbox") {
      value = event.target.checked;
    }

    setFormData({
      ...formData,
```

```
        [name]: value,
      });
    }

    function handleSubmit(event) {
      event.preventDefault();
      console.log(formData);
    }

    return (
      <form onSubmit={handleSubmit}>
        <input
          type="text"
          name="firstName"
          onChange={handleChange}
          value={formData.firstName}
        />
        <input
          type="text"
          name="lastName"
          onChange={handleChange}
          value={formData.lastName}
        />
        <input
          type="checkbox"
          name="admin"
          onChange={handleChange}
          checked={formData.admin}
        />
        <button type="submit">Submit</button>
      </form>
    );
  }
```

```
export default Form;
```

Depending on what input elements you're working with, you might also have to add some additional logic to handle things like number fields (using `parseInt` or `parseFloat`) and other data types to ensure your form state is always in sync with your components.

Conclusion

Working with controlled forms in React involves writing a lot of boilerplate code. We can abstract away some of that boilerplate by making our change handling logic more abstract.

Note: Working with complex forms can get quite challenging! If you're using a lot of forms in your application, it's worth checking out some nice React libraries like **react hook form (https://react-hook-form.com/)** to handle some of this abstraction. You can also use them to add custom client-side validation to your forms.

Resources

- **React Forms (https://reactjs.org/docs/forms.html)**