Forms

HTML form elements work a little bit differently from other DOM elements in React, because form elements naturally keep some internal state. For example, this form in plain HTML accepts a single name:

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

This form has the default HTML form behavior of browsing to a new page when the user submits the form. If you want this behavior in React, it just works. But in most cases, it's convenient to have a JavaScript function that handles the submission of the form and has access to the data that the user entered into the form. The standard way to achieve this is with a technique called "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 typically kept in the state property of components, and only updated with `setState()`.

We can combine the two by making the React state be the "single source of truth". Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a "controlled component".

For example, if we want to make the previous example log the name when it is submitted, we can write the form as a controlled component:

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Try it on CodePen

Since the `value` attribute is set on our form element, the displayed value will always be `this.state.value`, 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 straightforward to modify or validate user input. For example, if we wanted to enforce that names are written with all uppercase letters, we could write `handleChange` as:

```
handleChange(event) {
  this.setState({value: event.target.value.toUpperCase()});
}
```

The textarea Tag

In HTML, a `<textarea>` element defines its text by its children:

```
<textarea>
  Hello there, this is some text in a text area
</textarea>
```

In React, a `<textarea>` uses a `value` attribute instead. This way, a form using a `<textarea>` can be written very similarly to a form that uses a single-line input:

```
class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Essay:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Notice that `this.state.value` is initialized in the constructor, so that the text area starts off with some text in it.

The select Tag

In HTML, `<select>` creates a drop-down list. For example, this HTML creates a drop-down list of flavors:

```
<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>
```

Note that the Coconut option is initially selected, because of the `selected` attribute. React, instead of using this `selected` attribute, uses a `value` attribute on the root `select` tag. This is more convenient in a controlled component because you only need to update it in one place. For example:

```
class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
            <option value="coconut">Coconut</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Try it on CodePen

Overall, this makes it so that `<input type="text">`, `<textarea>`, and `<select>` all work very similarly - they all accept a `value` attribute that you can use to implement a controlled component.

> Note
>
> You can pass an array into the `value` attribute, allowing you to select multiple options in a `select` tag:
>
> ```
> <select multiple={true} value={['B', 'C']}>
> ```

The file input Tag

In HTML, an `<input type="file">` lets the user choose one or more files from their device storage to be uploaded to a server or manipulated by JavaScript via the File API.

```
<input type="file" />
```

Because its value is read-only, it is an **uncontrolled** component in React. It is discussed together with other uncontrolled components later in the documentation.

Handling Multiple Inputs

When you need to handle multiple controlled `input` elements, you can add a `name` attribute to each element and let the handler function choose what to do based on the value of `event.target.name`.

For example:

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };

    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <form>
        <label>
          Is going:
          <input
            name="isGoing"
            type="checkbox"
            checked={this.state.isGoing}
            onChange={this.handleInputChange} />
        </label>
        <br />
        <label>
          Number of guests:
          <input
            name="numberOfGuests"
            type="number"
            value={this.state.numberOfGuests}
            onChange={this.handleInputChange} />
        </label>
      </form>
    );
  }
}
```

Try it on CodePen

Note how we used the ES6 computed property name syntax to update the state key corresponding to the given input name:

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

It is equivalent to this ES5 code:

```
var partialState = {};
partialState[name] = value;
this.setState(partialState);
```

Also, since `setState()` automatically merges a partial state into the current state, we only needed to call it with the changed parts.

Controlled Input Null Value

Specifying the value prop on a controlled component prevents the user from changing the input unless you desire so. If you've specified a `value` but the input is still editable, you may have accidentally set `value` to `undefined` or `null`.

The following code demonstrates this. (The input is locked at first but becomes editable after a short delay.)

```
ReactDOM.render(<input value="hi" />, mountNode);

setTimeout(function() {
  ReactDOM.render(<input value={null} />, mountNode);
}, 1000);
```

Alternatives to Controlled Components

It can sometimes be tedious to use controlled components, because you need to write an event handler for every way your data can change and pipe all of the input state through a React component. This can become particularly annoying when you are converting a preexisting codebase to React, or integrating a React application with a non-React library. In these situations, you might want to check out uncontrolled components, an alternative technique for implementing input forms.

Fully-Fledged Solutions

If you're looking for a complete solution including validation, keeping track of the visited fields, and handling form submission, Formik is one of the popular choices. However, it is built on the same principles of controlled components and managing state — so don't neglect to learn them.