Forms

Form components such as `<input>`, `<textarea>`, and `<option>` differ from other native components because they can be mutated via user interactions. These components provide interfaces that make it easier to manage forms in response to user interactions.

Interactive Props

Form components support a few props that are affected via user interactions:

- `value`, supported by `<input>` and `<textarea>` components.
- `checked`, supported by `<input>` components of type `checkbox` or `radio`.
- `selected`, supported by `<option>` components.

In HTML, the value of `<textarea>` is set via children. In React, you should use `value` instead.

Form components allow listening for changes by setting a callback to the `onChange` prop. The `onChange` prop works across browsers to fire in response to user interactions when:

- The `value` of `<input>` or `<textarea>` changes.
- The `checked` state of `<input>` changes.
- The `selected` state of `<option>` changes.

Like all DOM events, the `onChange` prop is supported on all native components and can be used to listen to bubbled change events.

Controlled Components

An `<input>` with `value` set is a *controlled* component. In a controlled `<input>`, the value of the rendered element will always reflect the `value` prop. For example:

```
    return <input type="text" value="Hello!" />;
  }
```

This will render an input that always has a value of `Hello!`. Any user input will have no effect on the rendered element because React has declared the value to be `Hello!`. If you wanted to update the value in response to user input, you could use the `onChange` event:

Code

```
  getInitialState: function() {
    return {value: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({value: event.target.value});
  },
  render: function() {
    var value = this.state.value;
    return <input type="text" value={value} onChange={this.handleChange} />;
  }
```

In this example, we are simply accepting the newest value provided by the user and updating the `value` prop of the `<input>` component. This pattern makes it easy to implement interfaces that respond to or validate user interactions. For example:

Code

```
  handleChange: function(event) {
    this.setState({value: event.target.value.substr(0, 140)});
  }
```

This would accept user input but truncate the value to the first 140 characters.

Uncontrolled Components

An `<input>` that does not supply a `value` (or sets it to `null`) is an *uncontrolled* component. In an uncontrolled `<input>`, the value of the rendered element will reflect the user's input. For example:

Code

```
render: function() {
  return <input type="text" />;
}
```

This will render an input that starts off with an empty value. Any user input will be immediately reflected by the rendered element. If you wanted to listen to updates to the value, you could use the `onChange` event just like you can with controlled components.

If you want to initialize the component with a non-empty value, you can supply a `defaultValue` prop. For example:

Code

```
render: function() {
  return <input type="text" defaultValue="Hello!" />;
}
```

This example will function much like the Controlled Components example above.

Likewise, `<input>` supports `defaultChecked` and `<option>` supports `defaultSelected`.

Advanced Topics

Why Controlled Components?

Using form components such as `<input>` in React presents a challenge that is absent when

```
<input type="text" name="title" value="Untitled" />
```

This renders an input *initialized* with the value, `Untitled`. When the user updates the input, the node's value *property* will change. However, `node.getAttribute('value')` will still return the value used at initialization time, `Untitled`.

Unlike HTML, React components must represent the state of the view at any point in time and not only at initialization time. For example, in React:

Code

```
render: function() {
  return <input type="text" name="title" value="Untitled" />;
}
```

Since this method describes the view at any point in time, the value of the text input should *always* be `Untitled`.

Why Textarea Value?

In HTML, the value of `<textarea>` is usually set using its children:

Code

```
<!-- counterexample: DO NOT DO THIS! -->
<textarea name="description">This is the description.</textarea>
```

For HTML, this easily allows developers to supply multiline values. However, since React is JavaScript, we do not have string limitations and can use `\n` if we want newlines. In a world where we have `value` and `defaultValue`, it is ambiguous what role children play. For this reason, you should not use children when setting `<textarea>` values:

If you *do* decide to use children, they will behave like `defaultValue`.

← Prev

Next →