REACT FORMS
React Forms

This unit’s final lesson is about forms.

Think about how forms work in a typical, non-React environment. A user types some data into a form’s input fields, and the server doesn’t know about it. The server remains clueless until the user hits a “submit” button, which sends all of the form’s data over to the server simultaneously.

In React, as in many other JavaScript environments, this is not the best way of doing things.

The problem is the period of time during which a form thinks that a user has typed one thing, but the server thinks that the user has typed a different thing. What if, during that time, a *third* part of the website needs to know what a user has typed? It could ask the form or the server and get two different answers. In a complex JavaScript app with many moving, interdependent parts, this kind of conflict can easily lead to problems.

In a React form, you want the server to know about every new character or deletion, *as soon as it happens*. That way, your screen will always be in sync with the rest of your application.

Input onChange

A traditional form doesn’t update the server until a user hits “submit.” But you want to update the server *any time a user enters or deletes any character*.

Instructions

1.

Look at **Input.js**.

Can you find the bug? It’s somewhere in the `render` function. Scroll down once you have a guess.

…

…

…

…

…

…

…

There's a self-closing tag without a forward slash! Find the missing forward slash and fill it in.
Checkpoint 2 Passed

Hint

Find the `<input>` element and make sure it has a closing slash.
2.

Look at `Input`'s render function. View the result on the screen. Try typing into the `<input />` in the browser.

Once `Input` has been set up correctly, then you will be able to change the `<h1></h1>`'s inner text by typing into the `<input />` in the browser.
Checkpoint 3 Passed

Hint

No need to edit code for this checkpoint—just click Run.
3.

You want to respond to any entered or deleted character in the `<input />` field!

The best way to do that is by listening for a "change" event on the `<input />`.

Give `<input />` an `onChange` attribute.
Set `onChange`'s *value* equal to `{this.handleUserInput}`. Don't worry about the fact that there is no `handleUserInput` function yet - you'll make one!
Checkpoint 4 Passed

Hint

Make sure the `<input>` element has an attribute of `onChange={this.handleUserInput}`.

Input.js

```
import React from 'react';
import ReactDOM from 'react-dom';

export class Input extends React.Component {
  render() {
    return (
      <div>
        <input onChange={this.handleUserInput} type="text" /
>
        <h1>I am an h1.</h1>
      </div>
    );
  }
}

ReactDOM.render(
  <Input />,
  document.getElementById('app')
);
```

Write an Input Event Handler

In this exercise, you will define a function that gets called whenever a user enters or deletes any character.

This function will be an *event handler*. It will listen for `change` events. You can see an example of an event handler listening for change events in **Example.js**.

Instructions

1.

Select **Input.js**.

Before `Input`‘s render function, write a new method named `handleUserInput`.

Give this method one *parameter*, named `e`.

Inside of this function’s body, call `this.setState`. Set the state’s `userInput` property equal to `e.target.value`.

`e.target.value` will equal the text in the `<input />` field. You are setting `this.state.userInput` equal to whatever text is currently in `<input />`.

Checkpoint 2 Passed

Hint

You’ll need to write a new method on `Input` that looks like this:

```
handleUserInput(e) {
  // Update the state here
}
```

Example.js

```
import React from 'react';

export class Example extends React.Component {
  constructor(props) {
    super(props);

    this.state = { userInput: '' };

    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(e) {
    this.setState({
      userInput: e.target.value
    });
  }

  render() {
    return (
      <input
        onChange={this.handleChange}
        type="text" />
    );
  }
}
```

Input.js

```
import React from 'react';
import ReactDOM from 'react-dom';

export class Input extends React.Component {


  handleUserInput(e) {
    this.setState({
      userInput: e.target.value
    });
  }
  render() {
    return (
      <div>
        <input onChange={this.handleUserInput} type="text" /
>

        <h1>I am an h1.</h1>
      </div>
    );
  }
}

ReactDOM.render(
  <Input />,
  document.getElementById('app')
);
```

Set the Input's Initial State

Good! Any time that someone types or deletes in `<input />`, the `.handleUserInput()` method will update `this.state.userInput` with the `<input />`'s text.

Since you're using `this.setState`, that means that `Input` needs an initial state! What should `this.state`'s initial *value* be?

Well, `this.state.userInput` will be displayed in the `<input />`. What should the *initial* text in the `<input />` be, when a user first visits the page?

The initial text should be blank! Otherwise it would look like someone had already typed something.

Instructions

1.

Give `Input` a `constructor` function which takes a parameter of `props` and calls `super(props)` on its first line.

Then, still in the constructor, set `state` equal to this object:

```
{ userInput: '' }
```

Feel free to use the example code as a guide.

Checkpoint 2 Passed

Hint

Create the `constructor` function with the `props` argument like below.

```
constructor(props){

}
```

What goes inside the `constructor` function? Refer to the instruction above for more detail and don’t forget to use the `this` keyword to refer to the component’s `state`.

2.

Next, at the end of the constructor, bind `.handleUserInput()` to the current value of `this`.

Checkpoint 3 Passed

Hint

Inside of the instructor, after you set the initial state, bind `this.handleUserInput` to `this`.

If you wanted to bind a method called `myMethod`, you would do something like this:

```
constructor(props) {
  // ...
  this.myMethod = this.myMethod.bind(this);
}
```

Your code will look similar.

Input.js

```
import React from 'react';
import ReactDOM from 'react-dom';

export class Input extends React.Component {
  constructor(props) {
    super(props);
    this.state = { userInput: '' };
    this.handleUserInput = this.handleUserInput.bind(this);

  }

  handleUserInput(e) {
    this.setState({
      userInput: e.target.value
    });
  }
  render() {
    return (
      <div>
        <input onChange={this.handleUserInput} type="text" /
>
        <h1>I am an h1.</h1>
      </div>
    );
  }
}

ReactDOM.render(
  <Input />,
  document.getElementById('app')
);
```

Update an Input's Value

When a user types or deletes in the `<input />`, then that will trigger a *change* event, which will call `handleUserInput`. That's good!

`handleUserInput` will set `this.state.userInput` equal to whatever text is currently in the input field. That's also good!

There's only one problem: you can set `this.state.userInput` to whatever you want, but `<input />` won't care. You need to somehow make the `<input />`'s text responsive to `this.state.userInput`.

Easy enough! You can control an `<input />`'s text by setting its `value` attribute.

Instructions

1.

Inside of `Input`'s render function, give `<input />` the following attribute:

```
value={this.state.userInput}
```
Checkpoint 2 Passed

Hint

Find the `<input />` rendered by the `Input` component. Give it the following attribute:

```
value={this.state.userInput}
```
2.

That should do it! An idiomatically correct React form!

This example doesn't have a server per se, but any time that a user updates `<input />`, the new text is immediately stored in `Input`'s `state`. We could easily connect that `state` with a server. What matters is that the text is sent somewhere to be stored on every character change.

Inside of the `<h1></h1>`, delete the text:

```
I am an h1.
```
Replace it with:

```
{this.state.userInput}
```
Click Run. Try typing into the `<input />` in the browser.

Checkpoint 3 Passed

Hint

Find the `<h1></h1>` and put `{this.state.userInput}` inside.

Input.js

```
import React from 'react';
import ReactDOM from 'react-dom';

export class Input extends React.Component {
  constructor(props) {
    super(props);
    this.state = { userInput: '' };
    this.handleUserInput = this.handleUserInput.bind(this);

  }

  handleUserInput(e) {
    this.setState({
      userInput: e.target.value
    });
  }
  render() {
    return (
      <div>
        <input value={this.state.userInput} onChange={this.handleUserInput} type="text" />
        <h1>{this.state.userInput}</h1>
      </div>
    );
  }
}

ReactDOM.render(
  <Input />,
  document.getElementById('app')
);
```

Controlled vs Uncontrolled

There are two terms that will probably come up when you talk about React forms: *controlled component* and *uncontrolled component*. Like automatic binding, *controlled vs uncontrolled components* is a topic that you should be familiar with, but don’t need to understand deeply at this point.

An *uncontrolled component* is a component that maintains its own internal state. A *controlled component* is a component that does not maintain any internal state. Since a controlled component has no state, it must be *controlled* by someone else.

Think of a typical `<input type='text' />` element. It appears onscreen as a text box. If you need to know what text is currently in the box, then you can ask the `<input />`, possibly with some code like this:

```
let input = document.querySelector('input[type="text"]');

let typedText = input.value; // input.value will be equal to whatever text is currently in the text box.
```

The important thing here is that the `<input />` *keeps track of* its own text. You can ask it what its text is at any time, and it will be able to tell you.

The fact that `<input />` keeps track of information makes it an *uncontrolled component*. It maintains its own internal state, by remembering data about itself.

A *controlled component*, on the other hand, has no memory. If you ask it for information about itself, then it will have to get that information through `props`. Most React components are *controlled*.

In React, when you give an `<input />` a `value` attribute, then something strange happens: the `<input />` BECOMES controlled. It stops using its internal storage. This is a more ‘React’ way of doing things.

You can find more information about controlled and uncontrolled components in the React Forms documentation.

Instructions

Move to the next exercise when ready.

Input.js

```
import React from 'react';
import ReactDOM from 'react-dom';

export class Input extends React.Component {
  render() {
    return (
      <div>
        <input type="text" />
        <h1>I am an h1.</h1>
      </div>
    );
  }
}

ReactDOM.render(
  <Input />,
  document.getElementById('app')
);
```

React Forms Recap

Great work! You just wrote your first React form.

Notice that you didn’t use a submit button. You didn’t even use a `<form>` element! Your “form” was actually just an `<input />`.

That won’t always be the case. You will still sometimes want a `<form>` element and a submit button, especially if you need to differentiate between a finished form and an in-progress form. But in some cases, it’s fine to have a “form” that is really just an input field.

This is because, unlike in the traditional form paradigm, in React you re-send your form on every single character change. That removes the need to ever “submit” anything.

That marks the end of this unit! You’ve learned a wide variety of important techniques: inline styles, separating container and presentational components, stateless functional components, proptypes, and forms. You’ll review all of it in the next course! There is only one major tool still missing from your toolbelt: *lifestyle methods*. We’ll cover those in this course’s final unit.