Chapter 5. Event Handling

When it comes to user interfaces, presenting is only half the equation. The other part is responding to user input, which in JavaScript means handling user-generated events.

React's approach to event handling is to attach event handlers to components, and then to update those components' internal states when the handlers fire. Updating the component's internal state causes it to re-render, so all that is required for the user interface to reflect the consequences of the event is to read from the internal state in the component's render function.

Although it is common to update state based solely on the type of the event in question, it is often necessary to use additional information from the event in order to determine how to update the state. In such a case, the Event Object that is passed to the handler will provide extra information about the event, which the handler can then use to update the internal state.

Between these techniques and React's highly efficient rendering, it becomes easy to respond to user's inputs and to update the user interface based on the consequences of those inputs.

Attaching Event Handlers

At a basic level, React handles the same events you see in regular JavaScript: `MouseEvents` for click handlers, `Change` events when form elements change, and so on. The events have the same names you would see in regular JavaScript, and will trigger under the same circumstances.

React's syntax for attaching event handlers closely resembles HTML's. For example, in the SurveyBuilder, the following code attaches an `onClick` handler to the Save button.

```
<button className="btn btn-save" onClick={this.handleSaveClicked}>Save</button>
```

When the user clicks the button, the button's `handleSaveClicked` method will run. That method

If you are not using JSX, you instead specify the handler as one of the fields in the options object. For example:

```
React.DOM.button({className: "btn btn-save", onClick: this.handleSaveClicked}, "Save");
```

React has first-class support for handling a wide variety of event types, which it lists in the Event System (http://facebook.github.io/react/docs/events.html) page of its documentation.

Most of these work without any extra effort, but touch events must be manually enabled by calling this:

```
React.initializeTouchEvents(true);
```

Events and State

Suppose you want a component to change based on user input, like in the survey editor, where you want to let users drag survey questions in from a menu of question types.

Start with a render function which registers some event handlers based on the HTML5 Drag and Drop API.

```
    var SurveyEditor = React.createClass({
      render: function () {
        return (
          <div className='survey-editor'>
            <div className='row'>
              <aside className='sidebar col-md-3'>
                <h2>Modules</h2>
                <DraggableQuestions />
              </aside>

              <div className='survey-canvas col-md-9'>
                <div
                  className={'drop-zone well well-drop-zone'}
                  onDragOver={this.handleDragOver}
                  onDragEnter={this.handleDragEnter}
                  onDragLeave={this.handleDragLeave}
                  onDrop={this.handleDrop}
                >
                  Drag and drop a module from the left
                </div>
              </div>
            </div>
          </div>
        );
      });
```

The `DraggableQuestions` component will render the menu of question types, and the handler

Rendering Based on State

One thing the handler methods will need to do is to expand some running list of the questions added so far. To do this, you will need to make use of the internal `state` object that every React component contains. It begins as null by default, but can be initialized to something sane with a component's `getInitialState()` method, like so:

```
getInitialState: function () {
  return {
    dropZoneEntered: false,
    title: '',
    introduction: '',
    questions: []
  };
}
```

This establishes a sane baseline for the state: a blank title, a blank introduction, an empty list of questions, and a value of `false` for `dropZoneEntered` - indicating that the user is not currently dragging anything over the drop zone.

From here, you can read from `this.state` in the `render` method in order to display the current values of each of these to the user.

```
render: function () {
  var questions = this.state.questions;

  var dropZoneEntered = '';
  if (this.state.dropZoneEntered) {
    dropZoneEntered = 'drag-enter';
  }

  return (
    <div className='survey-editor'>
      <div className='row'>
        <aside className='sidebar col-md-3'>
          <h2>Modules</h2>
          <DraggableQuestions />
        </aside>

        <div className='survey-canvas col-md-9'>
          <SurveyForm
            title={this.state.title}
            introduction={this.state.introduction}
            onChange={this.handleFormChange}
          />

          <Divider>Questions</Divider>
          <ReactCSSTransitionGroup transitionName='question'>
            {questions}
          </ReactCSSTransitionGroup>

          <div
            className={'drop-zone well well-drop-zone ' + dropZoneEntered}
```

```
              onDragLeave={this.handleDragLeave}
              onDrop={this.handleDrop}
            >
              Drag and drop a module from the left
            </div>

            <div className='actions'>
              <button className="btn btn-save" onClick={this.handleSaveClicked}>Save</bu
            </div>
          </div>
        </div>
      </div>
    );
  }
```

As with `this.props`, the render function can change as little or as much as you like depending on the values of `this.state`. It can render the same elements but with slightly different attributes, or a completely different set of elements altogether. Either works just as well.

Updating State

Since updating a component's internal `state` causes the component to re-render, the next thing to do is to update that state in the drag handler methods. Then `render` will run again, it will read from the current value of `this.state` to display the title, introduction, and questions, and the user will see that everything has been updated properly.

There are two ways to update a component's state: the component's `setState` method and its`replaceState` method. `replaceState` overwrites the entire state object with an entirely new state object, which is rarely what you want. Much more often you will want to use `setState`, which simply merges the object you give it into the existing state object.

For example, suppose you have the following for your current state:

```
  getInitialState: function () {
    return {
      dropZoneEntered: false,
      title: 'Fantastic Survey',
      introduction: 'This survey is fantastic!',
      questions: []
    };
  }
```

In this case, calling `this.setState({title: "Fantastic Survey 2.0"})` only affects the value of`this.state.title`, leaving `this.state.dropZoneEntered`, `this.state.introduction`, and`this.state.questions` unaffected.

Calling `this.replaceState({title: "Fantastic Survey 2.0"})` would instead replace the entire state object with the new object `{title: "Fantastic Survey 2.0"}`, and `this.state.questions` altogether. This would likely break the render function, which would expect `this.state.questions` to be an array instead of `undefined`.

Using `this.setState`, you can now implement the handler methods from earlier.

```
handleFormChange: function (formData) {
  this.setState(formData);
},

handleDragOver: function (ev) {
  // This allows handleDropZoneDrop to be called
  // https://code.google.com/p/chromium/issues/detail?id=168387
  ev.preventDefault();
},

handleDragEnter: function () {
  this.setState({dropZoneEntered: true});
},

handleDragLeave: function () {
  this.setState({dropZoneEntered: false});
},

handleDrop: function (ev) {
  var questionType = ev.dataTransfer.getData('questionType');
  var questions = this.state.questions;
  questions = questions.concat({ type: questionType });

  this.setState({
    questions: questions,
    dropZoneEntered: false
  });
}
```

It's important never to alter the `state` object by any other means than calling `setState` or`replaceState`. Doing something like `this.state.saveInProgress = true` is generally a bad idea, as it will not inform React that it might need to re-render, and might lead to surprising results the next time you call `setState`.

Event Objects

Many handlers simply need to fire in order to serve their purposes, but sometimes you need more information about the user's input.

Take a look at the `AnswerEssayQuestion` class from the SurveyBuilder.

```
var AnswerEssayQuestion = React.createClass({
  handleComplete: function(event) {
    this.callMethodOnProps('onCompleted', event.target.value);
  },
  render: function() {
    return (
            <div className="survey-item-content">
              <textarea className="form-control" rows="3" onBlur={this.handleComplete}/>
            </div>
          </div>
        );
      }
    });
```

React’s event handler functions are always passed an event object, much in the same way that a vanilla JavaScript event listener would be. Here, the `handleComplete` method takes an event object, from which it extracts the current value of the `textarea` by accessing `event.target.value`. Using `event.target.value` in an event handler like this is a common way to get the value from a form input, especially in an `onChange` handler.

Rather than passing the original Event object from the browser directly to the handler, React wraps the original event in a `SyntheticEvent` instance. A `SyntheticEvent` is designed to look and function the same way as the original event the browser created, except with certain cross-browser inconsistencies smoothed over. You should be able to use the `SyntheticEvent` the same way you would a normal event, but in case you need the original event that the browser sent, you can access it via the `SyntheticEvent`’s `nativeEvent` field.

Summary

The steps to reflect changes from user input in the user interface are simple:

1. Attach an event handler to a React component
2. In that event handler, update the component’s internal state. Updating the component’s state will cause it to re-render.
3. Modify the component’s `render` function to incorporate `this.state` as appropriate.

So far you have used a single component to respond to user interactions. Next you will learn how to compose multiple components together, to create an interface that is more than the sum of its parts.

PREV
4. Data Flow

NEXT
6. Composing Components