Chapter 8. DOM Manipulation

For the most part, React's virtual DOM is sufficient to create the user experience you want without having to work directly with the actual underlying DOM at all. By composing components together you can weave complex interactions together into a cohesive whole for the user.

However, in some cases there is no avoiding working with the underlying DOM to accomplish what you need. The most common use cases for this are when you need to incorporate a third-party library that does not use React, or when you need to perform an operation that React does not natively support.

To facilitate this, React provides a system for working with DOM nodes that are being managed by React. They are only accessible during certain parts of the component lifecycle, but using them gives you the power you need to handle these use cases.

Accessing Managed DOM Nodes

To access DOM nodes managed by React, you must first be able to access the components responsible for managing them. Adding a `ref` attribute to child components lets you do this.

```
var DoodleArea = React.createClass({
  render: function() {
    return <canvas ref="mainCanvas" />;
  }
});
```

This will make the `<canvas>` component accessible via `this.refs.mainCanvas`. As you can imagine, the `ref` that you give a child component must be unique among all of a component's children; giving a different child a `ref` of `mainCanvas` as well would not work.

Once we have accessed the child component in question, we can invoke that the `render` method, because the underlying DOM nodes may not be up-to-date (or even created yet!) until `render` completes and React performs its updates.

As such, you cannot invoke the `getDOMNode()` method until the component has been “mounted” - at which point the `componentDidMount` handler will run.

```
var DoodleArea = React.createClass({
  render: function() {
    // We are unmounted inside render(), so this will cause an exception!
    this.getDOMNode();

    return <canvas ref="mainCanvas" />
  },

  componentDidMount: function() {
    var canvasNode = this.refs.mainCanvas.getDOMNode()
    // This will work! We now have access to the HTML5 Canvas node,
    // and can invoke painting methods on it as desired.
  }
});
```

Note that `componentDidMount` is not the only place in which you can call `getDOMNode`. Event handlers also fire after a component has mounted, so you can use it just as easily inside them as you would in a `componentDidMount` handler.

```
var RichText = React.createClass({
  render: function() {
    return <div ref="editableDiv" contentEditable="true" onKeyDown={this.handleKeyDown}>
  },

  handleKeyDown: function() {
    var editor = this.refs.editableDiv.getDOMNode()
    var html = editor.innerHTML;

    // Now we can persist the HTML content the user has entered!
  }
});
```

The above example creates a `div` with `contentEditable` enabled, allowing the user to enter rich text into it.

Although React does not natively provide a way to access a component’s raw HTML contents, the `keyDown` handler can accesses that div’s underlying DOM node, which in turn can access the raw HTML. From there, you can save a copy of what the user has entered so far, compute a word count to display, and so on.

Incorporating Non-React Libraries

There are many useful JavaScript libraries that were not built with React in mind. Some do not keeping their states synchronized with React is critical for successful integration.

Suppose you want to use an autocomplete library which includes the following example code:

```
autocomplete({
  target: document.getElementById("cities"),
  data: [
    "San Francisco",
    "St. Louis",
    "Amsterdam",
    "Los Angeles"
  ],
  events: {
    select: function(city) {
      alert("You have selected the city of " + city);
    }
  }
});
```

This `autocomplete` function needs a target DOM node, a list of strings to use as data, and then some event handlers. To reap the benefits of both React and this library, you can start by building a React component that provides each of these.

```
var AutocompleteCities = React.createClass({
  render: function() {
    return <div id="cities" ref="autocompleteTarget" />
  },

  getDefaultProps: function() {
    return {
      data: [
        "San Francisco",
        "St. Louis",
        "Amsterdam",
        "Los Angeles"
      ]
    };
  },

  handleSelect: function(city) {
    alert("You have selected the city of " + city);
  }
});
```

To finish wrapping this library in React, add a `componentDidMount` handler which connects the two implementations via the underlying DOM node of the `autocompleteTarget` child component.

```
var AutocompleteCities = React.createClass({
  render: function() {
    return <div id="cities" ref="autocompleteTarget" />
  },
      return {
        data: [
          "San Francisco",
          "St. Louis",
          "Amsterdam",
          "Los Angeles"
        ]
      };
    },

    handleSelect: function(city) {
      alert("You have selected the city of " + city);
    },

    componentDidMount: function() {
      autocomplete({
        target: this.refs.autocompleteTarget.getDOMNode(),
        data: this.props.data,
        events: {
          select: this.handleSelect
        }
      });
    }
  });
```

Remember that `componentDidMount` can be called multiple times, and the underlying DOM nodes in question may not have been recreated in between calls. As such, make sure that calling `autocomplete` (in this example) twice on the same node will not have any undesired effects.

Summary

When using the virtual DOM alone is not sufficient, the `ref` attribute allows you to access specific elements and modify their underlying DOM nodes using `getDOMNode` once `componentDidMount` has run.

This allows you to make use of functionality that React does not natively support, or to incorporate third-party libraries that were not designed to interoperate with React.

Next, it's time to look at how you can write tests to ensure your React components are working as expected.

PREV
7. Mixins

NEXT
9. Forms