More About Refs

After returning the structure of your UI from the render method, you may find yourself wanting to "reach out" and invoke methods on component instances returned from render. Often, doing something like this isn't necessary for making data flow through your application, because the Reactive data flow always ensures that the most recent `props` are sent to each child that is output from `render()`. However there are a few cases, where it still might be necessary or beneficial.

Consider the case when you wish to tell an `<input />` element (that exists within your instances sub-hierarchy) to focus after you update its value to be the empty string, `''`.

Code

```
var App = React.createClass({
  getInitialState: function() {
    return {userInput: ''};
  },
  handleKeyUp: function(e) {
    this.setState({userInput: e.target.value});
  },
  clearAndFocusInput: function() {
    this.setState({userInput: ''}); // Clear the input
    // We wish to focus the <input /> now!
  },
  render: function() {
    return (
      <div>
        <div onClick={this.clearAndFocusInput}>
          Click To Focus and Reset
        </div>
        <input
      </div>
    );
  }
});
```

Notice how, in this example, we want to "tell" the input something - something that it cannot infer from it's props over time. In this case we want to "tell" it that it should now become focused. However, there are some challenges. What is returned from `render()`` is not your actual composition of "child" components, it is merely a *description* of the children at a particular instance in time - a snapshot, if you will.

> **Note:**
>
> Remember, what you return from `render()` is not your *actual* rendered children instances. What you return from `render()` is merely a *description* of the children instances in your component's sub-hierarchy at a particular moment in time.

This means that you should never "hold onto" something that you return from `render()` and then expect it to be anything meaningful.

Code

```
// counterexample: DO NOT DO THIS!
render: function() {
  var myInput = <input />;          // I'm going to try to call methods on this
  this.rememberThisInput = myInput; // input at some point in the future! YAY!
  return (
    <div>
      <div>...</div>
      {myInput}
    </div>
  );
```

In this counterexample, the `<input />` is merely a *description* of an `<input />`. This description is used to create a *real* backing instance for the `<input />`.

So how do we talk to the *real* backing instance of the input?

The ref Attribute

React supports a very special property that you can attach to any component that is output from `render()`. This special property allows you to refer to the corresponding backing instance of anything returned from `render()`. It is always guaranteed to be the proper instance, at any point in time.

It's as simple as:

1. Assign a `ref` attribute to anything returned from `render` such as:

Code

```
  <input ref="myInput" />
```

2. In some other code (typically event handler code), access the backing instance via `this.refs` as in:

Code

```
this.refs.myInput
```

Completing the Example

Code

```
var App = React.createClass({
  getInitialState: function() {
    return {userInput: ''};
    this.setState({userInput: e.target.value});
  },
  clearAndFocusInput: function() {
    this.setState({userInput: ''}); // Clear the input
    this.refs.theInput.getDOMNode().focus();   // Boom! Focused!
  },
  render: function() {
    return (
      <div>
        <div onClick={this.clearAndFocusInput}>
          Click To Focus and Reset
        </div>
        <input
          ref="theInput"
          value={this.state.userInput}
          onKeyUp={this.handleKeyUp}
        />
      </div>
    );
  }
});
```

In this example, our render function returns a description of an `<input />` instance. But the true instance is accessed via `this.refs.theInput`. As long as a child component with `ref="theInput"` is returned from render, `this.refs.theInput` will access the proper instance. This even works on higher level (non-DOM) components such as `<Typeahead ref="myTypeahead" />`.

Summary

Refs are a great way to send a message to a particular child instance in a way that would be inconvenient to do via streaming Reactive `props` and `state`. They should, however, not be

Benefits:

- You can define any public method on your component classes (such as a reset method on a Typeahead) and call those public methods through refs (such as `this.refs.myTypeahead.reset()`).
- Performing DOM measurements almost always requires reaching out to a "native" component such as `<input />` and accessing its underlying DOM node via `this.refs.myInput.getDOMNode()` . Refs are one of the only practical ways of doing this reliably.
- Refs are automatically book-kept for you! If that child is destroyed, its ref is also destroyed for you. No worrying about memory here (unless you do something crazy to retain a reference yourself).

Cautions:

- *Never* access refs inside of any component's render method - or while any component's render method is even running anywhere in the call stack.
- If you want to preserve Google Closure Compiler Crushing resilience, make sure to never access as a property what was specified as a string. This means you must access using `this.refs['myRefString']` if your ref was defined as `ref="myRefString"` .
- If you have not programmed several apps with React, your first inclination is usually going to be to try to use refs to "make things happen" in your app. If this is the case, take a moment and think more critically about where `state` should be owned in the component hierarchy. Often, it becomes clear that the proper place to "own" that state is at a higher level in the hierarchy. Placing the state there often eliminates any desire to use `ref`s to "make things happen" - instead, the data flow will usually accomplish your goal.

← Prev Next →

Post to Facebook Posting as Stoyan Stefanov (Change) Comment

Facebook social plugin