Interactivity and Dynamic UIs

You've already learned how to display data with React. Now let's look at how to make our UIs interactive.

A Simple Example

Code

```
/** @jsx React.DOM */

var LikeButton = React.createClass({
  getInitialState: function() {
    return {liked: false};
  },
  handleClick: function(event) {
    this.setState({liked: !this.state.liked});
  },
  render: function() {
    var text = this.state.liked ? 'like' : 'unlike';
    return (
      <p onClick={this.handleClick}>
        You {text} this. Click to toggle.
      </p>
    );
  }
});

React.renderComponent(
  <LikeButton />,
```

Event Handling and Synthetic Events

With React you simply pass your event handler as a camelCased prop similar to how you'd do it in normal HTML. React ensures that all events behave identically in IE8 and above by implementing a synthetic event system. That is, React knows how to bubble and capture events according to the spec, and the events passed to your event handler are guaranteed to be consistent with the W3C spec, regardless of which browser you're using.

If you'd like to use React on a touch device (i.e. a phone or tablet), simply call `React.initializeTouchEvents(true);` to turn them on.

Under the Hood: autoBind and Event Delegation

Under the hood React does a few things to keep your code performant and easy to understand.

Autobinding: When creating callbacks in JavaScript you usually need to explicitly bind a method to its instance such that the value of `this` is correct. With React, every method is automatically bound to its component instance. React caches the bound method such that it's extremely CPU and memory efficient. It's also less typing!

Event delegation: React doesn't actually attach event handlers to the nodes themselves. When React starts up, it starts listening for all events at the top level using a single event listener. When a component is mounted or unmounted, the event handlers are simply added or removed from an internal mapping. When an event occurs, React knows how to dispatch it using this mapping. When there are no event handlers left in the mapping, React's event handlers are simple no-ops. To learn more about why this is fast, see David Walsh's excellent blog post.

Components are Just State Machines

React thinks of UIs as simple state machines. By thinking of a UI as being in various states and rendering those states, it's easy to keep your UI consistent.

In React, you simply update a component's state, and then render a new UI based on this new state. React takes care of updating the DOM for you in the most efficient way.

How State Works

A common way to inform React of a data change is by calling `setState(data, callback)`. This method merges `data` into `this.state` and re-renders the component. When the component finishes re-rendering, the optional `callback` is called. Most of the time you'll never need to provide a `callback` since React will take care of keeping your UI up-to-date for you.

What Components Should Have State?

Most of your components should simply take some data from `props` and render it. However, sometimes you need to respond to user input, a server request or the passage of time. For this you use state.

Try to keep as many of your components as possible stateless. By doing this you'll isolate the state to its most logical place and minimize redundancy, making it easier to reason about your application.

A common pattern is to create several stateless components that just render data, and have a stateful component above them in the hierarchy that passes its state to its children via `props`. The stateful component encapsulates all of the interaction logic, while the stateless components take care of rendering data in a declarative way.

What *Should* Go in State?

stateful component, think about the minimal possible representation of its state, and only store those properties in `this.state`. Inside of `render()` simply compute any other information you need based on this state. You'll find that thinking about and writing applications in this way tends to lead to the most correct application, since adding redundant or computed values to state means that you need to explicitly keep them in sync rather than rely on React computing them for you.

What *Shouldn't* Go in State?

`this.state` should only contain the minimal amount of data needed to represent your UI's state. As such, it should not contain:

- Computed data: Don't worry about precomputing values based on state — it's easier to ensure that your UI is consistent if you do all computation within `render()`. For example, if you have an array of list items in state and you want to render the count as a string, simply render `this.state.listItems.length + ' list items'` in your `render()` method rather than storing it on state.
- React components: Build them in `render()` based on underlying props and state.
- Duplicated data from props: Try to use props as the source of truth where possible. Because props can change over time, it's appropriate to store props in state to be able to know its previous values.

← Prev Next →

Post to Facebook Posting as Stoyan Stefanov (Change) Comment

Facebook social plugin