Reusable Components

When designing interfaces, break down the common design elements (buttons, form fields, layout components, etc) into reusable components with well-defined interfaces. That way, the next time you need to build some UI you can write much less code, which means faster development time, less bugs, and less bytes down the wire.

Prop Validation

As your app grows it's helpful to ensure that your components are used correctly. We do this by allowing you to specify `propTypes`. `React.PropTypes` exports a range of validators that can be used to make sure the data you receive is valid. When an invalid value is provided for a prop, an error will be thrown. Here is an example documenting the different validators provided:

Code

```
React.createClass({
  propTypes: {
    // You can declare that a prop is a specific JS primitive. By default, these
    // are all optional.
    optionalArray: React.PropTypes.array,
    optionalBool: React.PropTypes.bool,
    optionalFunc: React.PropTypes.func,
    optionalNumber: React.PropTypes.number,
    optionalObject: React.PropTypes.object,
    optionalString: React.PropTypes.string,

    // You can ensure that your prop is limited to specific values by treating
    // it as an enum.
    optionalEnum: React.PropTypes.oneOf(['News','Photos']),
    // JS's instanceof operator.
    someClass: React.PropTypes.instanceOf(SomeClass),

    // You can chain any of the above with isRequired to make sure an error is
    // thrown if the prop isn't provided.
    requiredFunc: React.PropTypes.func.isRequired

    // You can also specify a custom validator.
    customProp: function(props, propName, componentName) {
      if (!/matchme/.test(props[propName])) {
        throw new Error('Validation failed!')
      }
    }
  },
  /* ... */
});
```

Default Prop Values

React lets you define default values for your `props` in a very declarative way:

Code

```
var ComponentWithDefaultProps = React.createClass({
  getDefaultProps: function() {
    return {
      value: 'default value'
    };
  }
  /* ... */
});
```

use your props without having to write repetitive and fragile code to handle that yourself.

Transferring Props: A Shortcut

A common type of React component is one that extends a basic HTML in a simple way. Often you'll want to copy any HTML attributes passed to your component to the underlying HTML element to save typing. React provides `transferPropsTo()` to do just this.

Code

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

var CheckLink = React.createClass({
  render: function() {
    // transferPropsTo() will take any props passed to CheckLink
    // and copy them to <a>
    return this.transferPropsTo(<a>{'√ '}{this.props.children}</a>);
  }
});

React.renderComponent(
  <CheckLink href="javascript:alert('Hello, world!');">
    Click here!
  </CheckLink>,
  document.getElementById('example')
);
```

Mixins

Components are the best way to reuse code in React, but sometimes very different components may share some common functionality. These are sometimes called cross-cutting use `setInterval()`, but it's important to cancel your interval when you don't need it anymore to save memory. React provides lifecycle methods that let you know when a component is about to be created or destroyed. Let's create a simple mixin that uses these methods to provide an easy `setInterval()` function that will automatically get cleaned up when your component is destroyed.

Code

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

var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.map(clearInterval);
  }
};

var TickTock = React.createClass({
  mixins: [SetIntervalMixin], // Use the mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // Call a method on the mixin
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
        React has been running for {this.state.seconds} seconds.
      </p>
    );
  }
});

React.renderComponent(
  <TickTock />,
  document.getElementById('example')
);
```

A nice feature of mixins is that if a component is using multiple mixins and several mixins define the same lifecycle method (i.e. several mixins want to do some cleanup when the component is destroyed), all of the lifecycle methods are guaranteed to be called.

← Prev Next →

Add a comment...

Post to Facebook Posting as Stoyan Stefanov (Change) Comment

Facebook social plugin