PREV
3. Component Lifecycle

NEXT
5. Event Handling

Chapter 4. Data Flow

In React, data flows in one direction only: From the parent to the child. This makes components really simple and predictable. They take props from the parent and render. If a prop is changed at the top level component, React will propagate that change all the way down the component tree and re-render all the components that used that property.

Components can also have internal state, which should only be modified within the component.

React components are inherently simple and you can consider them as a function that take `props` and `state` and output HTML.

In this chapter we look will look at:

- What “props” are
- What “state” is
- When to use “props” and when to use “state”

Props

“props”, short for “properties" are passed to a component and can hold any data you’d like.

You can set props on a component during instantiation:

```
var surveys = [{ title: 'Superheroes' }];
<ListSurveys surveys={surveys}/>
```

or via the setProps method on a component instance:

```
var surveys = [{ title: 'Superheroes' }];
var listSurveys = React.renderComponent(
  <ListSurveys/>,
  document.querySelector(‘body’)
);
listSurveys.setProps({ surveys: surveys });
```

You can access props via `this.props`, but you should never write to props that way. A component should never modify its own props.

When used with JSX, props can be set as a string:

```
<a href='/surveys/add'>Add survey</a>
```

It can also be set with the `{}` syntax, which injects JavaScript and allows you to pass in variables of any type:

```
<a href={'/surveys/' + survey.id}>{survey.title}</a>
```

It’s possible with the `transferPropsTo` method to pass all props received to a child component, but it is advised against using this function as it encourages tight coupling. Instead explicitly copy over the props from the parent to the child component:

```
var ListSurveys = React.createClass({
  render: function () {
    return this.transferPropsTo(<SurveyTable/>);
  }
});

// vs the explicit
var ListSurveys = React.createClass({
  render: function () {
    return <SurveyTable surveys={this.props.surveys}/>;
  }
});
```

Props are useful for event handlers as well:

```
var SaveButton = React.createClass({
  render: function () {
    return (
      <a className='button save' onClick={this.handleClick}>Save</a>
    );
  },
  handleClick: function () {
    // ...
  }
});
```

Here we are passing the onClick prop to the anchor tag with the value of the `handleClick` function.

PropTypes

React provides a way to validate your props, through a config object defined on your component:

```
var SurveyTableRow = React.createClass({
  propTypes: {
    survey: React.PropTypes.shape({
      id: React.PropTypes.number.isRequired
    }).isRequired,
    onClick: React.PropTypes.func
  },
  // ...
});
```

If the requirements of the propTypes are not met when the component is instantiated, a `console.warn` will be logged.

For optional props simple leave the `.isRequired` off.

You are not required to use propTypes in your application, but they provide a good way to describe the API of your component.

getDefaultProps

Create the `getDefaultProps` function on your component to provide a default set of properties. This should only be done for props that aren't required.

```
var SurveyTable = React.createClass({
  getDefaultProps: function () {
    return {
      surveys: []
    };
  }
  // ...
});
```

It's important to note that `getDefaultProps` is not called during component instantiation but as soon as `React.createClass` is called to cache the value. This means you can't use any instance specific data in the `getDefaultProps` method.

state

Each component in a React can house state. State differs from props in that it is internal to the component.

State is useful for deciding a view state on an element. Consider the `<AnswerRadioInput/>` component used for collecting survey answers:

```
var AnswerRadioInput = React.createClass({
  getInitialState: function () {
    return {
      checked: !!this.props.checked
    };
  },

  handleChanged: function (e) {
    var checked = e.target.checked;
    this.setState({checked: checked});
  },

  render: function () {
    return (
      <div className="radio">
        <label>
          <input type="radio"
            checked={this.state.checked}
            onChange={this.handleChanged} />
          {this.props.label}
        </label>
      </div>
    );
  }
});
```

Here we use state to track if the checkbox should be checked or not.

State is altered via the `setState` method and can have a default value provided the `getInitialState` method as shown above. When ever `setState` is called within the component the render method is called and if there were changes to the output of the render function, the DOM will be updated and finally the user will see a change in their browser.

State will always make your component more complex, if you isolate your state to certain components you application becomes easier to debug and reason about.

What belongs in state and what belongs in props

Don't store computed values or components in state, instead focus on simple data that is directly required for the component to function, like our checked state from earlier. Without it we couldn't check and uncheck the checkbox. This could also be a boolean for showing the options of a dropdown, or the values of an input field.

Try not to duplicate prop data into state. When possible consider props the source of truth.

PREV
3. Component Lifecycle

NEXT
5. Event Handling