



Chapter 13. Addons

React comes with a set of addons. We use several off them throughout our Sample application.

In this chapter we will go into how each addon works and how they can help you with your React application. We'll cover the following React addons (the full set at the time of the writing):

- `React.addons.CSSTransitionGroup` for Animations
- `React.addons.LinkedStateMixin` to help with two-way data bindings
- `React.addons.classSet` for simple class name manipulation
- `React.addons.TestUtils` for easing your React unit testing
- `React.addons.cloneWithProps` for component cloning
- `React.addons.update` for simple immutability
- `React.addons.PureRenderMixin` for quick performance boosts
- `React.addons.Perf` for diagnosing performance bottlenecks in your application

First though, it's important to note that the default React distribution doesn't come with all the addons. However, when you download the starter kit from <http://facebook.github.io/react/downloads.html> it includes a distribution with all the addons called `react.addons.js`. This file is required instead of the regular `React.js` distribution if you want to use any of the addons.

Animations

React provides a component via it's addons that helps with animations, we can use it to improve the experience of our `SurveyBuilder`.

Let's add an animation when you add or remove any questions for the `SurveyEditor`.

Wrapping our list of EditQuestion modules in the `React.addons.CSSTransitionGroup` component will add a series of CSS classes to modules when ever that list changes

Here's a shorted version of our `<SurveyEditor/>` component with the added `React.addons.CSSTransitionGroup` code.

```
var ReactCSSTransitionGroup = React.addons.CSSTransitionGroup;

var SurveyEditor = React.createClass({
  getInitialState: function () {
    return {
      questions: []
      // ...
    };
  },

  render: function () {
    var questions = this.state.questions.map(function (q, i) {
      return SUPPORTED_QUESTIONS[q.type]({
        key: i,
        onChange: this.handleQuestionChange,
        onRemove: this.handleQuestionRemove,
        question: q
      });
    });
    return (
      <div className='survey-editor'>
        <div className='row'>
          <aside className='sidebar col-md-3'>
            <h2>Modules</h2>
            <DraggableQuestions />
          </aside>

          <div className='survey-canvas col-md-9'>
            <SurveyForm />

            <Divider>Questions</Divider>

            <ReactCSSTransitionGroup transitionName='question'>
              {questions}
            </ReactCSSTransitionGroup>

            // ...
          </div>
        </div>
      </div>
    );
  },

  // ...
});
```

Now this by it self doesn't cause any animations, this simply adds and removes CSS classes. The `transitionName` property we gave the `ReactCSSTransitionGroup` component is used for the

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

(depending whether or not React saw a new element or an element was removed) is `.question-enter`, `.question-enter-active`, `.question-leave` and `.question-leave-active`. We can target these classes in our stylesheet and add an animation when a question is added and an animation when it is removed.

There are 2 classes for elements entering as children to the `ReactCSSTransitionGroup` component and 2 classes for leaving.

The first class that is added is `.question-enter`. This is meant for you to set the stage, prepare the animation if you will.

We want sort of a “drop-in” effect. We achieve this by starting the element at a 1.2 scale and animating it down. We use a cubic-bezier to control the easing so that it feels bouncy:

```
.survey-editor .question-enter {
  transform: scale(1.2);
  transition: transform 0.2s cubic-bezier(.97,.84,.5,1.21);
}
```

On the very next UI tick the `.question-enter-active` class is added. This is our cue to start the animation. We simply reset the scale to 1 which kicks off the animation set in `.question-enter`.

```
.survey-editor .question-enter-active {
  transform: scale(1);
}
```

The same principles apply for when a child leaves `ReactCSSTransitionGroup`.

Here we add a fade animation while the question moves upwards:

```
.survey-editor .question-leave {
  transform: translateY(0);
  opacity: 0;
  transition: opacity 1.2s, transform 1s cubic-bezier(.52,-0.25,.52,.95);
}

.survey-editor .question-leave-active {
  opacity: 0;
  transform: translateY(-100%);
}
```

Two-Way Binding Helpers

To get data from a form field when the user interacts with it we can add an `onChange` handler to the element like so:

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

```
var EditEssayQuestion = React.createClass({
  getInitialState: function () {
    return {
      title: ''
    };
  },
  render: function () {
    return (
      <div>
        <input
          type='text'
          value={this.state.title}
          onChange={this.handleTitleChange}/>
      </div>
    );
  },
  handleTitleChange: function (ev) {
    this.setState({ title: ev.target.value });
  }
});
```

Notice that we set the value of the input field to `this.state.title`. This means that the input is “controlled”, and if we don’t add an `onChange` handler that updates the state, any value the user enters will be discarded. Simply put, with the above example, every time the user enters something into the input field the `handleTitleChange` function is called, we update the state to the new value and the render function gets called again and that finally updates the input field, making the change visible to the user.

Now, building `onChange` handlers like this is simple, but can be made even simpler with the `LinkStateMixin`.

We can re-write the above example like so and it will build the `onChange` handler for us behind the scenes.

```
var EditEssayQuestion = React.createClass({
  mixins: [React.addons.LinkStateMixin],
  getInitialState: function () {
    return { title: '' };
  },
  render: function () {
    return (
      <div>
        <input type='text' valueLink={this.linkState('title')}/>
      </div>
    );
  }
});
```

`React.addons.LinkStateMixin` is especially useful in big form components that need to track a lot of state.

It quickly becomes annoying to manually construct a className string. You can imagine the string being made off of varying state and props.

With the `React.addons.classSet` you can turn:

```
var SurveyRow = React.createClass({
  render: function () {
    var className = 'survey-row';
    if (this.props.survey.isActive) {
      className += ' active';
    }
    return (
      <tr className={className}>
        <td>{this.props.survey.title}</td>
      </tr>
    );
  }
});
```

into this:

```
var SurveyRow = React.createClass({
  render: function () {
    var classSet = React.addons.classSet({
      'survey-row': true,
      'active': this.props.survey.isActive
    });
    return (
      <tr className={classSet}>
        <td>{this.props.survey.title}</td>
      </tr>
    );
  }
});
```

Test Utilities

React provides simple a few test utilities available on `React.addons.TestUtils` - These are all covered in the ‘Testing’ chapter.

Cloning Components

Sometimes it’s useful to clone components. Perhaps you seeded a table with a default row component and want to clone it for adding a new row.

This is how it works:

```
var surveyRowClone = React.addons.cloneWithProps(
  <SurveyRow/>,
  { className: 'survey-row-clone' }
```

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

The second argument is extra props you can pass the clone.

Immutability Helpers

React doesn't force you to choose the kind of data structures you wish to use in your application. Using immutable datastructures can make your `shouldComponentUpdate` functions simpler when you need to compare objects to check for changes.

We can use `React.addons.update` to ensure immutability in our `<SurveyEditor/>` component by updating our change handler functions:

```
var update = React.addons.update;

var SurveyEditor = React.createClass({
  // ...

  handleDrop: function (ev) {
    var questionType = ev.dataTransfer.getData('questionType');
    var questions = update(this.state.questions, {
      $push: [{ type: questionType }]
    });

    this.setState({
      questions: questions,
      dropZoneEntered: false
    });
  },

  handleQuestionChange: function (key, newQuestion) {
    var questions = update(this.state.questions, {
      $splice: [[key, 1, newQuestion]]
    });

    this.setState({ questions: questions });
  },

  handleQuestionRemove: function (key) {
    var questions = update(this.state.questions, {
      $splice: [[key, 1]]
    });

    this.setState({ questions: questions });
  }

  // ...

});
```

`React.addons.update` takes a datastructure and an options hash. You can pass in `$slice`, `$push`, `$unshift`, `$set`, `$merge` and `$apply`.

PureRenderMixin

Find answers on the fly, or master something new. Subscribe today. [See pricing options.](#)

For pure components that always render the same given the same props and state we can add the `React.addons.PureRenderMixin` which gives a performance boost.

The mixin will overwrite `shouldComponentUpdate` to compare the new props and state with the old and return false if they are equal.

A few of our components are this simple, like our `EditEssayQuestion`, which we can use `React.addons.PureRenderMixin` with:

```
var EditEssayQuestion = React.createClass({
  mixin: [React.addons.PureRenderMixin],

  render: function () {
    var description = this.props.question.description || "";

    return (
      <EditQuestion type='essay' onRemove={this.handleRemove}>
        <label>Description</label>
        <input type='text' className='description' value={description} onChange={this.ha
      </EditQuestion>
    );
  },
  // ...
});
```

Performance Tools

As we've mentioned in the last few sections. Adding a custom `shouldComponentUpdate` function to your components can be a great way to optimize your application.

`React.addons.Perf` can help you find the best places to add `shouldComponentUpdate`.

Lets use `React.addons.Perf` to find slowness in our Survey Builder application, specifically the `<SurveyEditor/>` page.

First in the chrome web developer console we run `React.addons.Perf.start();` This will start the snapshot. We'll drag in a few questions and then run `React.addons.Perf.stop();` followed by `React.addons.Perf.printWasted();` in the console.

It gives us this result:

> <code>React.addons.Perf.printWasted()</code>			
(index)	Owner > component	Wasted time (ms)	Instances
0	"ReactTransitionGroup >..."	51.08800007439405	86
1	"SurveyEditor > Draggab..."	19.280999898910522	28
2	"SurveyEditor > SurveyF..."	10.835000139195472	28
3	"DraggableQuestions > M..."	8.496999624185264	84
4	"SurveyEditor > ReactCS..."	7.958000060170889	6
5	"AddSurvey > SurveyEdit..."	3.349000005982816	1
6	"SurveyEditor > Divider"	2.7780000236816704	28
7	"EditMultipleChoiceQues..."	2.5900001055561006	34
8	"SurveyForm > ReactDOMT..."	2.0920000970363617	28
9	"SurveyForm > ReactDOMI..."	1.800999918486923	28
10	"SurveyEditor > ReactD0..."	1.7349999397993088	28
		ReactDefaultPerf.js:99	
Total time: 136.04 ms		ReactDefaultPerf.js:106	

We can't do much with `<ReactTransitionGroup>` since we don't control that component, but a simple fix to `shouldComponentUpdate` on `<DraggableQuestions>` could work wonders. `<DraggableQuestion>` is actually quite a simple component. The way its constructed means it should never need to change. Lets update the `shouldComponentUpdate` function to always return false:

```
var DraggableQuestions = React.createClass({
  render: function () {
    return (
      <ul className="modules list-unstyled">
        <li><ModuleButton text='Yes / No' questionType='yes_no'></li>
        <li><ModuleButton text='Multiple choice' questionType='multiple_choice'></li>
        <li><ModuleButton text='Essay' questionType='essay'></li>
      </ul>
    );
  },

  shouldComponentUpdate: function () {
    return false;
  }
});
```

This removes the component entirely from the list of actions that had wasted time:

> <code>Perf.printWasted()</code>			
(index)	Owner > component	Wasted time (ms)	Instances
0	"ReactTransitionGroup > ReactCS..."	53.026000037789345	57
1	"SurveyEditor > SurveyForm"	14.322000090032816	19
2	"SurveyEditor > ReactCSStansi..."	8.422000042628497	4
3	"SurveyEditor > Divider"	3.816000127699226	19
4	"SurveyForm > ReactDOMInput"	3.748999966774136	19
5	"SurveyForm > ReactDOMTextarea"	2.1199999027885497	19
6	"EditMultipleChoiceQuestion > R..."	2.111000183504075	28
		ReactDefaultPerf.js:99	
Total time: 117.07 ms		ReactDefaultPerf.js:106	



PREV

[12. Server Side Rendering](#)

NEXT



[14. Tools and Debugging](#)