Chapter 10. Animations

Now that you can compose together a complex set of React components and test them, it's time to add some polish. Animation can make user experiences feel more fluid and natural, and React's Transition Groups addon, in conjunction with CSS3, make incorporating animated transitions into your React project a breeze.

Historically, animation in browsers has had a very imperative API. You would take an element and actively move it around or alter its styles in order to animate it. This approach is at odds with React's rendering and re-rendering of components, so instead React takes a more declarative approach.

CSS Transition Groups facilitate applying CSS3 animations to transitions, by strategically adding and removing classes during appropriately timed renders and re-renders. This means the only task you are left with is to specify the appropriate CSS styles for those classes.

Interval Rendering gives you more flexibility and control, at a cost to performance. It requires many more re-renderings, but allows you to animate more than just CSS, such as scroll position and Canvas drawing.

CSS Transition Groups

Take a look at how the Survey Builder renders the list of questions in the survey editor.

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

This `ReactCSSTransitionGroup` component comes from an addon, which is included near the top of the file with `var ReactCSSTransitionGroup =`
`React.addons.CSSTransitionGroup`

Find answers on the fly, or master something new. Subscribe today. See pricing options.

This automatically takes care of re-rendering the component at appropriate times, and altering the transition group's class in order to facilitate styling it situationally based on where it is in the transition.

Styling Transition Classes

By convention, the `transitionName='question'` attribute connects this element to four CSS classes: `question-enter`, `question-enter-active`, `question-leave`, and `question-leave-active`. The CSSTransitionGroup addon will automatically add and remove these classes as child components enter or leave the `ReactCSSTransitionGroup` component.

Here are the transition styles used by the survey editor.

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

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

.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%);
}
```

Note that these `.survey-editor` selectors are not required for the addon, but are simply used to make sure these styles are only applied within the editor.

The Transition Lifecycle

The difference between `question-enter` and `question-enter-active` is that the `question-enter` class is applied as soon as the component is added to the group, and the `question-enter-active` class is applied on the next tick. This allows you to easily specify the beginning style of the transition, the end style of the transition, and how to perform the transition.

For example, when survey editor questions are added to the list, they begin with an inflated scale (1.2) and then transition to a normal scale (1) over the course of 0.2 seconds. This creates the "popping into place" effect you see.

Find answers on the fly, or master something new. Subscribe today. See pricing options.

`{false}` attributes to the component. In addition to giving you control over which of the two you want, you can also use them to situationally disable animations altogether based on a configurable value, like so:

```
<ReactCSSTransitionGroup transitionName='question'
  transitionEnter={this.props.enableAnimations}
  transitionLeave={this.props.enableAnimations}>
  {questions}
</ReactCSSTransitionGroup>
```

Transition Group Pitfalls

There are two important pitfalls to watch out for when using transition groups.

First, the transition group will defer removing child components until animations complete. This means that if you wrap a list of components in a transition group, but do not specify any CSS for the `transitionName` classes, those components can no longer be removed - even if you try stop rendering them!

Second, transition group children must each have a unique `key` attribute set. The transition group uses these values to determine when components are entering or leaving the group, so animations could fail to run and components could become impossible to remove if they are missing their `key` attributes.

Note that even if the transition group only has a single child, it must still have a `key` attribute.

Interval Rendering

You can get great performance and concise code out of CSS3 animations, but they are not always the right tool for the job. Sometimes you have to target older browsers which do not support them. Other times you want to animate something other than a CSS property, such as scroll position or a Canvas drawing. In these cases Interval Rendering will accommodate your needs, but at a cost to performance compared to CSS3 animations.

The basic idea behind Interval Rendering is to periodically trigger a component state update which specifies how far the animation has progressed across its total running time. By incorporating this state value into a component's render function, the component can represent the appropriate stage of the animation each time the state change causes it to re-render.

Since this approach involves many re-renderings, it's typically best to use it in conjunction with `requestAnimationFrame` in order to avoid unnecessary renders. However, in environments where requestAnimationFrame is unavailable or otherwise undesirable, falling back on setTimeout can be a reasonable alternative.

Find answers on the fly, or master something new. Subscribe today. See pricing options.

Suppose you want to move a div across the screen using interval rendering. You could accomplish this by giving it `position: absolute` and then updating its `left` or `top` style attributes as time progressed. Doing this on `requestAnimationFrame`, and basing the amount of the change on how much time has elapsed, should result in a smooth animation.

Here is an example implementation.

```
var Positioner = React.createClass({
  getInitialState: function() { return {position: 0}; },

  resolveAnimationFrame: function() {
    var timestamp = new Date();
    var timeRemaining = Math.max(0, this.props.animationCompleteTimestamp - timestamp);

    if (timeRemaining > 0) {
      this.setState({position: timeRemaining});
    }
  },

  componentWillUpdate: function() {
    if (this.props.animationCompleteTimestamp) {
      requestAnimationFrame(this.resolveAnimationFrame);
    }
  },

  render: function() {
    var divStyle = {left: this.state.position};

    return <div style={divStyle}>This will animate!</div>
  }
});
```

In this example, the component has an `animationCompleteTimestamp` value set in its `props`, which it uses in conjunction with the timestamp returned by `requestAnimationFrame`'s callback to compute how much movement remains. The resulting value is stored as `this.state.position`, which the `render` method then uses to position the `div`.

Since `requestAnimationFrame` is invoked by the `componentWillUpdate` handler, it will be kicked off by any change to the component's props (such as a change to `animationCompleteTimestamp`), which includes the call to `this.setState` inside `resolveAnimationFrame`. This means that once `animationCompleteTimestamp` is set, the component will automatically continue firing successive requestAnimationFrame invocations until the current time exceeds the value of `animationCompleteTimestamp`.

Notice that this logic revolves around timestamps only. A change to `animationCompleteTimestamp` kicks it off, and the value of `this.state.position` depends entirely on the difference between the current time and `animationCompleteTimestamp`. As such, the render method is free to use `this.state.position` for any sort of animation, from setting scroll position to drawing on a canvas, and anything in between.

Find answers on the fly, or master something new. Subscribe today. See pricing options.

Interval Rendering with setTimeout

Although `requestAnimationFrame` is likely to get you the smoothest animation with the least overhead, it is not available in older browsers and may fire more often (or less predictably) than you want it to. In those cases you can use `setTimeout` instead.

```
var Positioner = React.createClass({
  getInitialState: function() { return {position: 0}; },

  resolveSetTimeout: function() {
    var timestamp = new Date();
    var timeRemaining = Math.max(0, this.props.animationCompleteTimestamp - timestamp);

    if (timeRemaining > 0) {
      this.setState({position: timeRemaining});
    }
  },

  componentWillUpdate: function() {
    if (this.props.animationCompleteTimestamp) {
      setTimeout(this.resolveSetTimeout, this.props.timeoutMs);
    }
  },

  render: function() {
    var divStyle = {left: this.state.position};

    return <div style={divStyle}>This will animate!</div>
  }
});
```

Since `setTimeout` takes an explicit time interval, whereas requestAnimationFrame determines that interval on its own, this component has the additional dependency of `this.props.timeoutMs`, which specifies the interval to use.

Summary

Using these animation techniques, you can now:

1. Efficiently animate transitions between states using CSS3 and Transition Groups.
2. Animate outside CSS, such as scroll position and Canvas drawing, using requestAnimationFrame.
3. Fall back on setTimeout in cases where requestAnimationFrame is not a viable option.

Next you'll be ready to take your React renderings to the server!

Find answers on the fly, or master something new. Subscribe today. See pricing options.

PREV
9. Forms

NEXT
11. Performance Tuning

Find answers on the fly, or master something new. Subscribe today. See pricing options.