PREV
6. Composing Components

NEXT
8. DOM Manipulation

Chapter 7. Mixins

What are Mixins?

On the homepage of React we have this example of a timer component:

```
var Timer = React.createClass({
  getInitialState: function() {
    return {secondsElapsed: 0};
  },
  tick: function() {
    this.setState({secondsElapsed: this.state.secondsElapsed + 1});
  },
  componentDidMount: function() {
    this.interval = setInterval(this.tick, 1000);
  },
  componentWillUnmount: function() {
    clearInterval(this.interval);
  },
  render: function() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
});
```

This is good, but we may have multiple components using timers; implementing this exact code. This is where mixins come in. We want to end up with a Timer component that looks like this:

```
var Timer = React.createClass({
  mixins: [IntervalMixin(1000)],
  getInitialState: function() {
    return {secondsElapsed: 0};
    this.setState({secondsElapsed: this.state.secondsElapsed + 1});
  },
  render: function() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
});
```

Mixins are fairly simple. They're objects which are mixed into our component class. React goes a bit further to prevent silently overwriting functions, and allowing multiple mixins which in other systems would conflict. For example:

```
React.createClass({
  mixins: [{
    getInitialState: function(){ return {a: 1} }
  }],
  getInitialState: function(){ return {b: 2} }
});
```

You have getInitialState defined in both the mixin and component class. The resulting initial state is {a: 1, b: 2}. If a key exists in both of these, an error will be thrown to alert you to the problem.

The lifecycle methods starting with "component", e.g. "componentDidMount", are called in the order of the mixin array, and finally the componentDidMount on the component class itself, if it exists.

So back to our original example, we need to implement IntervalMixin. Sometimes a simple object suffices, but other times we need a function which returns that object. In this case, you want to specify the interval.

```
var IntervalMixin = function(interval) {
  return {
    componentDidMount: function() {
      this.__interval = setInterval(this.onTick, interval);
    },
    componentWillUnmount: function() {
      clearInterval(this.__interval);
    }
  };
};
```

This is pretty good, but it has some limitations. You can't have multiple intervals, and uoi can't choose the function that handles the interval, and we can't terminate the interval manually without using the internal __interval property. To solve this, our mixin can have a public api.

Here's an example where we simply want to show the number of seconds since January 1st 2014. This mixin is a bit more code, but more flexible and powerful.

```
var IntervalMixin = {
  setInterval: function(callback, interval){
    var token = setInterval(callback, interval);
    this.__intervals.push(token);
    return token;
  },
  componentDidMount: function() {
    this.__intervals = [];
  },
  componentWillUnmount: function() {
    this.__intervals.map(clearInterval);
  }
};

var Since2014 = React.createClass({
  mixins: [IntervalMixin],
  componentDidMount: function(){
    this.setInterval(this.forceUpdate.bind(this), 1000);
  },
  render: function() {
    var from = Number(new Date(2014, 0, 1));
    var to = Date.now();
    return (
      <div>{Math.round((to-from) / 1000)}</div>
    );
  }
});
```

Mixins are one of the most powerful tools for eliminating code repetition, and keeping your components focused on what makes them special. They allow you to use powerful abstractions, and some problems can't be solved elegantly without them. Some noteworthy examples:

- a mixin which listens for events, and applies them to state (e.g. flux store mixin)
- an upload mixin which handles XHR uploads, and applies the status and progress of uploads to state
- 'layers' mixin, which facilitates rendering children to the end of </body> (e.g. for modals)

Testing

Now that you've learned how a mixin can be used in our component, an interesting question becomes -- how do we unit test it? Turns out there are three options for testing a mixin:

1. Test the mixin object directly
2. Test the mixin which is included in a fake component
3. Test the mixin which is included in the real component using a shared behavior spec

Testing the mixin directly

any React.js methods called from within a function. Let's give this a try by starting with the simplest function in our mixin, `componentDidMount`:

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

var IntervalMixin = require ('../../../client/testing_examples/interval_mixin');

describe("IntervalMixin", function(){
  describe("testing the mixin directly", function(){
    var subject;

    beforeEach(function(){
      // WARNING: DON'T DO THIS!! IT WILL CAUSE ISSUES WHICH WE WILL DISCUSS BELOW
      subject = IntervalMixin;
    });

    describe("componentDidMount", function(){
      it("should set an empty array called __intervals on the instance", function(){
        expect(subject.__intervals).toBeUndefined();

        subject.componentDidMount();

        expect(subject.__intervals).toEqual([]);
      });
    });
  });
});
```

When we run this tests it passes fine. You might notice that you are using the `IntervalMixin` as your subject, this is not a good idea because the `componentDidMount` function is setting a variable on `this`, `subject`, which in this case will be the `IntervalMixin` object. So when the next test runs, the `IntervalMixin` object has been polluted and will cause weird failures. To see this in action, modify the spec to this:

```
...

describe("IntervalMixin", function(){
  describe("testing the mixin directly", function(){
    var subject;

    beforeEach(function(){
      // WARNING: DON'T DO THIS!! IT WILL CAUSE ISSUES WHICH WE WILL DISCUSS BELOW
      subject = IntervalMixin;
    });

    describe("componentDidMount", function(){
      it("should set an empty array called __intervals on the instance", function(){
        expect(subject.__intervals).toBeUndefined();

        subject.componentDidMount();

        expect(subject.__intervals).toEqual([]);
      });
```

```
        subject.componentDidMount();

        expect(subject.__intervals).toEqual([]);
      });
    });
  });
});
```

And you will see the second test fails on the `expect(subject.__intervals).toBeUndefined();` line, because the test before had executed:

```
Chrome 37.0.2062 (Mac OS X 10.8.2) IntervalMixin testing the mixin directly componentDid
```

To fix this, we need to use a fresh copy of our mixin each time, so we'll switch our `beforeEach` to use `Object.create` and both tests will start passing:

```
...
    beforeEach(function(){
      subject = Object.create(IntervalMixin);
    });
...
```

Now that you've got a test passing for `componentDidMount`, lets work on testing `setInterval`. The `setInterval` function has three responsibilities:

1. be a pass-through to the real setInterval function
2. save off the interval id number into an array (so it can be cleared out later)
3. return the interval id

So those tests would look something like this:

```
...
    describe("setInterval", function(){
      var fakeIntervalId;
      beforeEach(function(){
        fakeIntervalId = 555;
        spyOn(window, "setInterval").andReturn(fakeIntervalId);
        // NOTE: how we have to call componentDidMount before we call setInterval, so th
        //   this.__intervals is defined.  This is a drawback of calling functions direc
        //   a mixin object
        subject.componentDidMount();
      });

      it("should call window.setInterval with the callback and the interval", function()
        expect(window.setInterval.callCount).toBe(0);

        subject.setInterval(function(){}, 500);
      });

      it("should store the setInterval id in the this.__intervals array", function(){
        subject.setInterval(function(){}, 500);

        expect(subject.__intervals).toEqual([fakeIntervalId]);
      });

      it("should return the setInterval id", function(){
        var returnValue = subject.setInterval(function(){}, 500);

        expect(returnValue).toBe(fakeIntervalId);
      });
    });

...
```

STUB OUT REACT.JS

Notice how the above tests don't need any React.js specific functionality, so they will be vanilla jasmine tests. If your mixin functions start calling methods which are provided by React.js (like `this.setState({})`), it is usually recommended to `spyOn(subject, "setState")` to mock out the React.js specific functions which are used. This allows you to keep your mixin test isolated to just the mixin object.

// TODO: tests for componentWillUnmount and explanation for it

You might have noticed that testing the mixin directly results in tests which are very fine grained. Sometimes this can be helpful when the behavior starts to grow in complexity, but sometimes it can lead to a path where you are testing the implementation and not the functionality. It also requires you to call any functions on the mixin in a specific order to mimic React.js lifecycle callbacks, like how we had to call `subject.componentDidMount();` in the `setInterval` test. The next section will show a way to test a mixin without these drawbacks.

Test the mixin which is included in a fake component

To test a mixin with a fake component, i.e. a "faux" component, you will need to define a React component in your test suite. The fact the component is defined in the test suite makes it very clear that this component exists for the sole purpose of testing and can't be used in the production application. Below is an example of our faux component. Notice how the functionality is simple so that we can keep the tests simple which will help keep the intent of each tests clear. The only bit of "cruft" is the `render` function which is required by React.

```
describe("testing the mixin via a faux component", function(){
  var FauxComponent;
```

```
      // Notice how the faux component is defined in the jasmine spec file.  This is
      //  intentional.  This expresses the intent that this react component exists
      //  for the sole purpose of testing this mixin.
      FauxComponent = React.createClass({
        mixins: [IntervalMixin],
        render: function(){
          return (<div>Faux components are all the rage!</div>);
        },
        myFakeMethod: function(){
          this.setInterval(function(){}, 500);
        }
      });
    });

  });
```

Now that you've got your faux component, let's write some tests:

```
...
    describe("setInterval", function(){
      var subject;
      beforeEach(function(){
        spyOn(window, "setInterval");
        subject = TestUtils.renderIntoDocument(<FauxComponent />);
      });

      it("should call window.setInterval with the callback and the interval", function()
        expect(window.setInterval.callCount).toBe(0);

        subject.myFakeMethod();

        expect(window.setInterval.callCount).toBe(1);
      });
    });

    describe("unmounting", function(){
      var subject;

      beforeEach(function(){
        spyOn(window, "setInterval").andReturn(555);
        spyOn(window, "clearInterval");
        subject = TestUtils.renderIntoDocument(<FauxComponent />);
        subject.myFakeMethod();
      });

      it("should clear any setTimeout's", function(){
        expect(window.clearInterval.callCount).toBe(0);
        React.unmountComponentAtNode(subject.getDOMNode().parentNode);
        expect(window.clearInterval.callCount).toBe(1);
      });
    });

...
```

The first major difference with these specs, which render a faux component, versus the specs which against it to make your test assertions. While this difference is glaring, there is a more subtle difference which can have a large effect on your tests: “Faux” has tests for `setInterval` and `unmounting`, while “direct” has tests for `setInterval`, `componentDidMount`, and `componentWillUnmount`.

That doesn’t sound like such a big deal, so who cares? For the answer for this, look at the `componentDidMount` function. Setting up `this.__intervals` provides no value in of itself, the value is only used with the other functions. In the “direct mixin” spec, we are testing the implementation of the function by asserting on `this.__intervals` which is an implementation detail, not functionality. In the “faux component” tests, we don’t need to test the implementation of `componentDidMount` because it is implicitly tested in the `setInterval` test when the component is rendered into the document.

Why is it important one describe block is called “unmounting” versus “componentWillUnmount”? Because we don’t actually care how the `clearInterval` code is called, we just care that it is called when the component is unmounted. So instead of calling `subject.componentWillUnmount` in the “direct” spec, we just unmount the component and let React.js call the appropriate callbacks in the correct order.

WHICH ONE TO CHOOSE?

In the two methods we have gone over so far, “direct” and “faux”, one is not better than the other. It depends on the complexity and behavior of your mixin for which one is better. One recommendation is to start with writing the “direct” spec, because it’s the most focused and then if it becomes a pain to write, then switch it to a faux spec (or do both). Don’t be afraid to just pick one option and let your tests tell you that you are wrong.

Shared behavior spec

The “faux” and “direct” approaches didn’t involve any of the components from your real application, only the mixin itself. The last approach will test the mixin via the real world components which will actually use that mixin -- this approach is called the “shared behavior” spec. The first big difference with this approach is that our tests are no longer located in the `interval_mixin_spec.js`, because the specs will be run by the `since_2014_spec.js`. Let’s start there:

```
/** @jsx React.DOM */
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;

var Since214 = require ('../../../client/testing_examples/since_2014');

describe("Since2014", function(){
});
```

We’ve got our boilerplate for a `Since2014` spec, so now let’s add our “shared example spec”:

```
...
describe("Since2014", function(){
  describe("shared examples", function(){
    IntervalMixinSharedExamples();
  });
});
...
```

If we run these tests they will fail because `IntervalMixinSharedExamples is not defined`, so let’s define that function:

```
...
var Since2014 = require ('../../../client/testing_examples/since_2014');
var IntervalMixinSharedExamples = require('../shared_examples/interval_mixin_shared_exam
...
```

and then in the interval_mixin_shared_examples.js file, put the boilerplate for a shared example spec:

```
/** @jsx React.DOM */
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;

var SetIntervalSharedExamples = function(attributes){

  var componentClass;

  beforeEach(function(){
    componentClass = attributes.componentClass;
  });

  describe("SetIntervalSharedExamples", function(){
  });
};

module.exports = SetIntervalSharedExamples;
```

If you look at this code carefully you will notice that `SetIntervalSharedExamples` is just a function which has a bunch of jasmine tests inside of it. So the `Since2014` specs call the `IntervalMixinSharedExamples();` function and then those tests will run.

Another important part of the “shared behavior” boilerplate is the `attributes.componentClass` part. This allows you to use dependency injection to pass in the component under test, in this example `Since2014`:

```
...
```

```
    IntervalMixinSharedExamples({componentClass: Since2014});
  });
});
```

Now let’s write the `SetIntervalSharedExamples` specs:

```
/** @jsx React.DOM */
var React = require("react/addons");
var TestUtils = React.addons.TestUtils;

var SetIntervalSharedExamples = function(attributes){

  var componentClass;

  beforeEach(function(){
    componentClass = attributes.componentClass;
  });

  describe("SetIntervalSharedExamples", function(){
    describe("setInterval", function(){

      var subject, fakeFunction;

      beforeEach(function(){
        spyOn(window, "setInterval");
        subject = TestUtils.renderIntoDocument(<componentClass />);
        fakeFunction = function(){};
      });

      it("should call window.setInterval with the callback and the interval", function()
        expect(window.setInterval).not.toHaveBeenCalledWith(fakeFunction, jasmine.any(Nu

        subject.setInterval(fakeFunction, 100);

        expect(window.setInterval).toHaveBeenCalledWith(fakeFunction, jasmine.any(Number
      });
    });

    describe("unmounting", function(){
      var subject, fakeFunction;

      beforeEach(function(){
        fakeFunction = function(){};

        spyOn(window, "setInterval").andCallFake(function(func, interval){
          // we want to make sure we are only asserting against the setInterval calls
          //  which come from this spec (not the ones which come from our components
          //  which use this mixin).  So we force the setInterval calls for our "fakeFun
          //  to return a different id number.
          if(func === fakeFunction){
            return 444;
          } else {
            return 555;
          }
        });
        spyOn(window, "clearInterval");
        subject = TestUtils.renderIntoDocument(<componentClass />);
      });

      it("should clear any setTimeout's", function(){
        expect(window.clearInterval).not.toHaveBeenCalledWith(444);

        React.unmountComponentAtNode(subject.getDOMNode().parentNode);

        expect(window.clearInterval).toHaveBeenCalledWith(444);
      });
    });
  });
};

module.exports = SetIntervalSharedExamples;
```

SHARED SPECS

One important distinction to keep in mind, is that behavior which is specific to `Since2014` should stay in the `since_2014_spec.js`, but functionality which is made avaialbe to `Since2014` by the `IntervalMixin` mixin should be tested in the `interval_mixin_shared_examples.js` spec file.

If you look at the specs for the shared behavior you will notice they are similar in nature to the specs for the faux component, but they are slightly more complex. In the “faux” example, we can do this in the `unmounting` spec:

```
spyOn(window, "setInterval").andReturn(555);
```

But in the “shared example” example, we have todo this in the `unmounting` spec:

```
        spyOn(window, "setInterval").andCallFake(function(func, interval){
          if(func === fakeFunction){
            return 444;
          } else {
            return 555;
          }
        });
```

This extra complexity is due to fact that the `Since2014` code will be calling `setInterval` in addition the to calls to `setInterval` in the shared behavior spec, so we have to be able to distinguish between them. Otherwise our shared behavior spec might not be testing the mixin correctly. This means that a “shared behavior” spec will have more noise and complexity than a comparable “faux spec”. This complexity can be worth it in certain cases: - If you mixin requires the React component to have certain functions/behavior defined in the component, the shared behavior spec can validate that the component has those functions/behavior which the mixin needs might get easily overridden or messed up by the component, a shared behavior spec can be a way to assert that doesn’t happen.

With the three available options to test a mixin (direct, faux, and shared behavior), each one comes with it’s own pros and cons. When testing a mixin, don’t feel afraid to try one option and switch if it isn’t working out. And it’s possible that a combination of different solutions covering different parts of the mixin might be the best approach!

PREV
6. Composing Components

NEXT
8. DOM Manipulation