React Advanced: Immutability & Performance

Trayan Iliev

IPT – Intellectual Products & Technologies
e-mail: tiliev@iproduct.org
web: http://www.iproduct.org

Agenda

1. Immutability and performance
2. Using addons - PureRenderMixin
3. shouldComponentUpdate() component method
4. shallowCompare()
5. Immutability Helpers addon
6. Cloning ReactElements.
7. Inversion of Control (IoC) principle and Dependency Injection (DI) pattern
8. Using React component Context
9. Injecting services using Context

Advanced Performance with React.js

- Will be React application as fast and responsive as an equivalent non-React version?
- What's the cost of re-rendering an entire sub-tree of components in response to every state change?
- React decides whether an actual DOM update is necessary by constructing a new virtual DOM and comparing it to the old one. Only in the case they are not equal, will React **reconcile** the DOM, applying as few mutations as possible.
- We can boost performance even further by mannually deciding when should component update – combined with **immutability** it could be really easy.

How to Boost Performance with React

- Use the production build
- Avoiding reconciling the DOM – React provides a component lifecycle function, **shouldComponentUpdate**, which is triggered before the re-rendering process starts (virtual DOM comparison and possible eventual DOM reconciliation), giving the developer the ability to **short circuit** this process. Default:

```
shouldComponentUpdate: function(nextProps, nextState) {
    return true;
}
```

- React invokes **shouldComponentUpdate** often -should be fast
- Use **immutability** for comparisons to be efficient

Performance with shouldComponentUpdate

```
React.createClass({
  propTypes: {
    value: React.PropTypes.string.isRequired
  },

  shouldComponentUpdate: function(nextProps, nextState) {
    return this.props.value !== nextProps.value;
  },

  render: function() {
    return <div>{this.props.value}</div>;
  }
});
```

Immutable.js to the Rescue

- **Immutable-js** is a JavaScript collections library written by Lee Byron, which Facebook recently open-sourced. It provides **immutable persistent collections via structural sharing**:
  - **Immutable**: once created, a collection cannot be altered
  - **Persistent**: new collections can be created from a previous collection and a mutation such as set. The original collection is still valid after the new collection is created.
  - **Structural Sharing**: new collections are created using as much of the same structure as the original collection as possible, reducing copying to a minimum to achieve space efficiency and acceptable performance. If the new collection is equal to the original, the original is often returned.

Using Immutable.js

```
var SomeRecord = Immutable.Record({ foo: null });
var x = new SomeRecord({ foo: 'bar'  });
var y = x.set('foo', 'baz');
x === y; // false
```

Using Immutable.js with Data Stores (1)

```
var User = Immutable.Record({
  id: undefined,
  name: undefined,
  email: undefined
});

var Message = Immutable.Record({
  timestamp: new Date(),
  sender: undefined,
  text: ''
});
```

Using Immutable.js with Data Stores (2)

```
this.users = Immutable.List();
this.messages = Immutable.List();

this.messages = this.messages.push(new Message({
  timestamp: payload.timestamp,
  sender: payload.sender,
  text: payload.text
});
```

Using Addons

React add-ons are collection of utility modules for React apps:

- **LinkedStateMixin** - simplifies coordination between user's form input data and the component's state
- **TransitionGroup** and **CSSTransitionGroup** - animations and transitions on events such as components adding and removal
- **createFragment** – create a set of externally-keyed children
- **update** – helper function dealing with immutable data
- **PureRenderMixin** – performance booster in certain situations
- **shallowCompare** – helper function that performs a shallow comparison for props and state
- **TestUtils** – simple helpers for writing test cases
- **Perf** – performance profiling tool for React code optimization

Boosting Performance: PureRenderMixin

```
var PureRenderMixin = require('react-addons-pure-render-mixin');
React.createClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

PureRenderMixin with ES6 classes

```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate =
          PureRenderMixin.shouldComponentUpdate.bind(this);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Shallow Compare for ES6 Classes

```
var shallowCompare = require('react-addons-shallow-compare');
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Immutability Helpers Addon

```
var update = require('react-addons-update');

var newData = update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});
```

Immutability Helpers - Commands

- **{$push: array}** - push() all the items in array on the target
- **{$unshift: array}** - unshift() all the items in array on the target
- **{$splice: array of arrays}** - for each item in arrays call splice() on the target with the parameters provided by the item
- **{$set: any}** - replace the target entirely
- **{$merge: object}** - merge the keys of object with the target
- **{$apply: function}** - passes in the current value to the function and updates it with the new returned value

Immutability Helpers - Examples

```
var collection = [1, 2, {a: [12, 17, 15]}];
var newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13,
14]]}}});
// => [1, 2, {a: [12, 13, 14, 15]}]

var obj = {a: 5, b: 3};
var newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});
// => {a: 5, b: 6}
// This is equivalent, but gets verbose for deeply nested collections:
var newObj2 = update(obj, {b: {$set: obj.b * 2}});

var obj = {a: 5, b: 3};
var newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```

Using React Component Context

- **React props** allow to track data-flow easy between componets
- React Context is alternative if you want to pass data through the component tree without having to pass the props down manually at every level.
- Inversion of Control (IoC) principle and Dependency Injection (DI) pattern
- React's "context" feature lets you do this. are collection of utility modules for React apps - Example:

```
TestList.contextTypes = {
    testService: React.PropTypes.object,
    router: React.PropTypes.object
};
```

How to Provide React Context (React 15)

```
getChildContext() {
  return {
      testService: this.testServiceSingleton,
      userService: this.userServiceSingleton,
      localeService: this.localeServiceSingleton
  };
}

onLocaleChange() {
  this.setState({});
}

IPTKnowledgeTester.childContextTypes = {
  testService: React.PropTypes.object,
  userService: React.PropTypes.object,
  localeService: React.PropTypes.object
};
```

React Context Usage Example (React 15)

```
handleAddTest() {
    const path = { pathname: '/test',
                   query: { controls: true, edit: true } };
    this.context.router.push(path);
 }

 componentDidMount() {
    this.context.testService.getTests().then((tests) => {
      this.setState({ tests: tests });
  });
 }
```

Referencing Context in Lifecycle Methods

```
void componentWillReceiveProps(
  object nextProps, object nextContext
)

boolean shouldComponentUpdate(
  object nextProps, object nextState, object nextContext
)

void componentWillUpdate(
  object nextProps, object nextState, object nextContext
)

void componentDidUpdate(
  object prevProps, object prevState, object prevContext
)
```

Using Context with Functional Components

```
const Button = ({children}, context) =>
<button style={{background: context.color}}>
    {children}
</button>;

Button.contextTypes = {color: React.PropTypes.string};
```

Thanks for Your Attention!

Questions?