Table of Contents

React Bits

A compilation of React Patterns, techniques, tips and tricks.

Gitbook format: https://vasanthk.gitbooks.io/react-bits

> Your contributions are heartily ♡ welcome. (✿◠‿◠)
>
> **Translations by community:**
>
> - 中文版 (Chinese): react-bits-cn

Conditionals in JSX

Instead of

```
const sampleComponent = () => {
  return isTrue ? <p>True!</p> : <none/>
};
```

Use short-circuit evaluation

```
const sampleComponent = () => {
  return isTrue && <p>True!</p>
};
```

For Complex scenarios with too many ternaries:

```
// Y soo many ternary??? :-/
const sampleComponent = () => {
  return (
    <div>
      {flag && flag2 && !flag3
        ? flag4
        ? <p>Blah</p>
        : flag5
        ? <p>Meh</p>
        : <p>Herp</p>
        : <p>Derp</p>
      }
    </div>
  )
};
```

- Best approach: Move logic to sub-components
- Alternate hacky approach: Use IIFE

There are some libraries that solve this problem (JSX-Control Statements), but rather than introduce another dependency use an IIFE and return values by using if-else statement inside

```
const sampleComponent = () => {
  return (
    <div>
      {
        (() => {
          if (flag && flag2 && !flag3) {
            if (flag4) {
              return <p>Blah</p>
            } else if (flag5) {
              return <p>Meh</p>
            } else {
              return <p>Herp</p>
            }
          } else {
            return <p>Derp</p>
          }
        })()
      }
    </div>
  )
};
```

With an appropiate transpiler you can take advantage of the upcoming do expression which is currently on stage-1

```
const sampleComponent = () => {
  return (
    <div>
      {
        do => {
          if (flag && flag2 && !flag3) {
            if (flag4) {
              <p>Blah</p>
            } else if (flag5) {
              <p>Meh</p>
            } else {
              <p>Herp</p>
            }
          } else {
            <p>Derp</p>
          }
        }
      }
    </div>
  )
};
```

Or alternatively simply use early returns

```
const sampleComponent = () => {
  const basicCondition = flag && flag2 && !flag3;
  if (!basicCondition) return <p>Derp</p>;
  if (flag4) return <p>Blah</p>;
  if (flag5) return <p>Meh</p>;
  return <p>Herp</p>
}
```

Related links:

- https://engineering.musefind.com/our-best-practices-for-writing-react-components-dec3eb5c3fc8
- Conditional rendering

Async Nature Of setState()

On the Async nature of setState()

Gist:

React batches updates and flushes it put once per frame (perf optimization) However, in some cases React has no control over batching, hence updates are made synchronously eg. eventListeners, Ajax, setTimeout and similar Web APIs

Main Idea

setState() does not immediately mutate this.state but creates a pending state transition. Accessing this.state after calling this method can potentially return the existing value. There is no guarantee of synchronous operation of calls to setState and calls may be batched for performance gains.

Run the below code and you will make the following observations:

You can see that in every situation (addEventListener, setTimeout or AJAX call) the state before and the state after are different. And that render was called immediately after triggering the setState method. But why is that? Well, it turns out React does not understand and thus cannot control code that doesn't live inside the library. Timeouts or AJAX calls for example, are developer authored code that executes outside of the context of React.

So why does React synchronously updated the state in these cases? Well, because it's trying to be as defensive as possible. Not being in control means it's not able to do any perf optimisations so it's better to update the state on spot and make sure the code that follows has access to the latest information available.

```
class TestComponent extends React.Component {
  constructor(...args) {
    super(...args);
    this.state = {
      dollars: 10
    };
    this._saveButtonRef = (btn => { this._btnRef = btn });
    [
      '_onTimeoutHandler',
      '_onMouseLeaveHandler',
      '_onClickHandler',
      '_onAjaxCallback',
    ].forEach(propToBind => {
      this[propToBind] = this[propToBind].bind(this);
    });
  }

  componentDidMount() {
    // Add custom event via `addEventListener`
    //
    // The list of supported React events does include `mouselea
ve`
    // via `onMouseLeave` prop
    //
    // However, we are not adding the event the `React way` - th
is will have
    // effects on how state mutates
    //
    // Check the list here - https://facebook.github.io/react/do
cs/events.html
    this._btnRef.addEventListener('mouseleave', this._onMouseLea
veHandler);

    // Add JS timeout
    //
    // Again,outside React `world` - this will also have effects
 on how state
    // mutates
    setTimeout(this._onTimeoutHandler, 10000);

    // Make AJAX request
    fetch('https://api.github.com/users')
      .then(this._onAjaxCallback);
  }
  render() {
    console.log('State in render: ' + JSON.stringify(this.state)
);

    return (
       <button
         ref={this._saveButtonRef}
         onClick={this._onClickHandler}>
         'Click me'
      </button>
    );
  }

  _onClickHandler() {
    console.log('State before (_onClickHandler): ' + JSON.string
ify(this.state));
    this.setState({
      dollars: this.state.dollars + 10
    });
    console.log('State after (_onClickHandler): ' + JSON.stringi
fy(this.state));
  }

  _onMouseLeaveHandler() {
    console.log('State before (mouseleave): ' + JSON.stringify(t
his.state));
    this.setState({
      dollars: this.state.dollars + 20
    });
    console.log('State after (mouseleave): ' + JSON.stringify(th
is.state));
  }

  _onTimeoutHandler() {
    console.log('State before (timeout): ' + JSON.stringify(this
.state));
    this.setState({
      dollars: this.state.dollars + 30
    });
    console.log('State after (timeout): ' + JSON.stringify(this.
state));
  }

  _onAjaxCallback(response) {
    if (response.status !== 200) {
      console.log('Error in AJAX call: ' + response.statusText);
      return;
    }
    console.log('State before (AJAX call): ' + JSON.stringify(th
is.state));
    this.setState({
      dollars: this.state.dollars + 40
    });
    console.log('State after (AJAX call): ' + JSON.stringify(this
.state));
  }
};

// Render to DOM
ReactDOM.render(
  <TestComponent />,
  document.getElementById('app')
);
```

Possible solution?

We're used to calling setState with one parameter only, but actually, the method's signature support two. The second argument that you can pass in is a callback function that will always be executed after the state has been updated (whether it's inside React's known context or outside of it).

An example might be:

```
_onClickHandler: function _onClickHandler() {
   console.log('State before (_onClickHandler): ' + JSON.stringi
fy(this.state));
   this.setState({
   dollars: this.state.dollars + 10
   }, () => {
   console.log('Here state will always be updated to latest vers
ion!');
   console.log('State after (_onClickHandler): ' + JSON.stringif
y(this.state));
   });
}
```

A note on the async nature of setstate

To be politically correct, setState, as a method, is always synchronous. It's just a function that calls something behind the scenes - enqueueState or enqueueCallback on updater.

In fact, here's setState taken directly from React source code:

```
ReactComponent.prototype.setState = function(partialState, callb
ack) {
  invariant(
    typeof partialState === 'object' ||
    typeof partialState === 'function' ||
    partialState == null,
    'setState(...): takes an object of state variables to update
 or a ' +
    'function which returns an object of state variables.'
  );
  this.updater.enqueueSetState(this, partialState);
  if (callback) {
    this.updater.enqueueCallback(this, callback, 'setState');
  }
};
```

What's actually sync or async are the effects of calling setState in a React application - the reconciliation algorithm, doing the VDOM comparisons and calling render to update the real DOM.

Related links:

- https://medium.com/@wereHamster/beware-react-setstate-is-asynchronous-ce87ef1a9cf3#.jhdhncws3
- https://www.bennadel.com/blog/2893-setstate-state-mutation-operation-may-be-synchronous-in-reactjs.htm

Dependency Injection

In React the need of dependency injection is easily visible. Let's consider the following application tree:

```
// Title.jsx
export default function Title(props) {
  return <h1>{ props.title }</h1>;
}
```

```
// Header.jsx
import Title from './Title.jsx';
export default function Header() {
  return (
    <header>
      <Title />
    </header>
  );
}
```

```
// App.jsx
import Header from './Header.jsx';
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = { title: 'React Dependency Injection' };
  }
  render() {
    return <Header />;
  }
}
```

The string "React Dependency Injection" should somehow reach the Title component. The direct way of doing this is to pass it from App to Header and then Header to pass it to Title. However, this may work for these three components but what happens if there are multiple properties and deeper nesting. Lots of components will have to mention properties that they are not interested in. It is clear that most React components receive their dependencies via props but the question is how these dependencies reach that point.

One way to achieve dependency injection is by using higher-order component to inject data.

```
// inject.jsx
var title = 'React Dependency Injection';
export default function inject(Component) {
  return class Injector extends React.Component {
    render() {
      return (
        <Component
          {...this.state}
          {...this.props}
          title={ title }
        />
      )
    }
  };
}
```

```
// Title.jsx
export default function Title(props) {
  return <h1>{ props.title }</h1>;
}
```

```
// Header.jsx
import inject from './inject.jsx';
import Title from './Title.jsx';

var EnhancedTitle = inject(Title);
export default function Header() {
  return (
    <header>
      <EnhancedTitle />
    </header>
  );
}
```

The title is hidden in a middle layer (higher-order component) where we pass it as a prop to the original Title component. That's all nice but it solves only half of the problem. Now we don't have to pass the title down the tree but how this data will reach the enhance.jsx helper.

Using React's context

React has the concept of context. The context is something that every component may have access to. It's something like an event bus but for data. A single model which we can access from everywhere.

a place where we'll define the context

```
var context = { title: 'React in patterns' };
class App extends React.Component {
  getChildContext() {
    return context;
  }
  // ...
}

App.childContextTypes = {
  title: PropTypes.string
};
```

A place where we need data

```
class Inject extends React.Component {
  render() {
    var title = this.context.title;
  // ...
  }
}
Inject.contextTypes = {
  title: PropTypes.string
};
```

Related links:

- What is Dependency Injection?
- The Basics of Dependency Injection
- Dependency injection in JavaScript
- DI In React

Context Wrapper

It is a good practice that our context is not just a plain object but it has an interface that allows us to store and retrieve data. For example:

```
// dependencies.js
export default {
  data: {},
  get(key) {
    return this.data[key];
  },
  register(key, value) {
    this.data[key] = value;
  }
}
```

Then, if we go back to our example, the very top App component may look like that:

```
import dependencies from './dependencies';
dependencies.register('title', 'React in patterns');

class App extends React.Component {
  getChildContext() {
    return dependencies;
  }
  render() {
    return <Header />;
  }
}

App.childContextTypes = {
  data: PropTypes.object,
  get: PropTypes.func,
  register: PropTypes.func
};
```

And our Title component gets it's data through the context:

```
// Title.jsx
export default class Title extends React.Component {
  render() {
    return <h1>{ this.context.get('title') }</h1>
  }
}
Title.contextTypes = {
  data: PropTypes.object,
  get: PropTypes.func,
  register: PropTypes.func
};
```

Ideally we don't want to specify the contextTypes every time when we need an access to the context. This detail may be wrapped in a higher-order component. And even more, we may write an utility function that is more descriptive and helps us declare the exact wiring. ie. instead of accessing the context directly with this.context.get('title') we ask the higher-order component to get what we need and to pass it as a prop to our component. For example:

```
// Title.jsx
import wire from './wire';

function Title(props) {
  return <h1>{ props.title }</h1>;
}

export default wire(Title, ['title'], function resolve(title) {
  return { title };
});
```

The wire function accepts first a React component, then an array with all the needed dependencies (which are registered already) and then a function which I like to call mapper. It receives what's stored in the context as a raw data and returns an object which is the actual React props for our component (Title). In this example we just pass what we get - a title string variable. However, in a real app this could be a collection of data stores, configuration or something else. So, it's nice that we pass exactly what we need and don't pollute the components with data that they don't need.

Here is how the wire function looks like:

```
export default function wire(Component, dependencies, mapper) {
  class Inject extends React.Component {
    render() {
      var resolved = dependencies.map(this.context.get.bind(this
.context));
      var props = mapper(...resolved);

      return React.createElement(Component, props);
    }
  }
  Inject.contextTypes = {
    data: PropTypes.object,
    get: PropTypes.func,
    register: PropTypes.func
  };
  return Inject;
};
```

Inject is a higher-order component that gets access to the context and retrieves all the items listed under dependencies array. The mapper is a function receiving the context data and transforms it to props for our component.

Non-context alternative

Use a singleton to register/fetch all dependencies

```
var dependencies = {};

export function register(key, dependency) {
  dependencies[key] = dependency;
}

export function fetch(key) {
  if (key in dependencies) return dependencies[key];
  throw new Error(`"${ key } is not registered as dependency.`);
}

export function wire(Component, deps, mapper) {
  return class Injector extends React.Component {
    constructor(props) {
      super(props);
      this._resolvedDependencies = mapper(...deps.map(fetch));
    }
    render() {
      return (
        <Component
          {...this.state}
          {...this.props}
          {...this._resolvedDependencies}
        />
      );
    }
  };
}
```

We'll store the dependencies in dependencies global variable (it's global for our module, not at an application level).

We then export two functions register and fetch that write and read entries.

It looks a little bit like implementing setter and getter against a simple JavaScript object.

Then we have the wire function that accepts our React component and returns a higher-order component.

In the constructor of that component we are resolving the dependencies and later while rendering the original component we pass them as props.

We follow the same pattern where we describe what we need (deps argument) and extract the needed props with a mapper function.

Having the di.jsx helper we are again able to register our dependencies at the entry point of our application (app.jsx) and inject them wherever (Title.jsx) we need.

```
// app.jsx
import Header from './Header.jsx';
import { register } from './di.jsx';

register('my-awesome-title', 'React in patterns');

class App extends React.Component {
  render() {
    return <Header />;
  }
}
```

```
// Header.jsx
import Title from './Title.jsx';

export default function Header() {
  return (
    <header>
      <Title />
    </header>
  );
}
```

```
// Title.jsx
import { wire } from './di.jsx';

var Title = function(props) {
  return <h1>{ props.title }</h1>;
};

export default wire(Title, ['my-awesome-title'], title => ({ tit
le }));
```

If we look at the `Title.jsx` file we'll see that the actual component and the wiring may live in different files. That way the component and the mapper function become easily unit testable.

Event Handlers

Binding event handlers in the constructor.

Most of the times we handle DOM events in the component that contains the elements dispatching the events. Like in the example below, we have a click handler and we want to run a function or method of the same component:

```
class Switcher extends React.Component {
  render() {
    return (
      <button onClick={ this._handleButtonClick }>
        click me
      </button>
    );
  }
  _handleButtonClick() {
    console.log('Button is clicked');
  }
}
```

That's all fine because `_handleButtonClick` is a function and we indeed pass a function to the onClick attribute.

The problem is that as it is the code doesn't keep the scope. So, if we have to use `this` inside `_handleButtonClick` we'll get an error.

```
class Switcher extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'React in patterns' };
  }
  render() {
    return (
      <button onClick={ this._handleButtonClick }>
        click me
      </button>
    );
  }
  _handleButtonClick() {
    console.log(`Button is clicked inside ${ this.state.name }`)
;
    // leads to
    // Uncaught TypeError: Cannot read property 'state' of null
  }
}
```

What we normally do is to use bind like so:

```
<button onClick={ this._handleButtonClick.bind(this) }>
  click me
</button>
```

However, this means that the bind function is called again and again because we may render the button many times. A better approach would be to create the bindings in the constructor of the component:

```
class Switcher extends React.Component {
  constructor(props) {
    super(props);
    this.state = { name: 'React in patterns' };
    this._buttonClick = this._handleButtonClick.bind(this);
  }
  render() {
    return (
      <button onClick={ this._buttonClick }>
        click me
      </button>
    );
  }
  _handleButtonClick() {
    console.log(`Button is clicked inside ${ this.state.name }`)
;
  }
}
```

The other alternative is to use arrow functions for the onClick prop function assignment. Arrow functions don't affect the context at invocation time (`this` value from the surrounding scope is used).

Facebook by the way recommend the same technique while dealing with functions that need the context of the same component. The binding in the constructor may be also useful if we pass callbacks down the tree.

A short hand example using arrow functions and avoid having to use the constructor:

```
class Switcher extends React.Component {
  state = { name: 'React in patterns' };

  render() {
    return (
      <button onClick={ this._handleButtonClick }>
        click me
      </button>
    );
  }
  _handleButtonClick = () => {
    console.log(`Button is clicked inside ${ this.state.name }`)
;
  }
}
```

Flux pattern for data handling

Simple dispatcher

```
var Dispatcher = function () {
  return {
    _stores: [],
    register: function (store) {
      this._stores.push({store: store});
    },
    dispatch: function (action) {
      if (this._stores.length > 0) {
        this._stores.forEach(function (entry) {
          entry.store.update(action);
        });
      }
    }
  }
};
```

We expect the store to have an update method(), so let's modify register to expect it.

```
function register(store) {
  if (!store || !store.update || typeof store.update !== 'functi
on') {
    throw new Error('You should provide a store that has an upda
te method');
  } else {
    this._stores.push({store: store});
  }
}
```

Full blown Dispatcher

```
var Dispatcher = function () {
  return {
    _stores: [],
    register: function (store) {
      if (!store || !store.update) {
        throw new Error('You should provide a store that has an
`update` method.');
      } else {
        var consumers = [];
        var change = function () {
          consumers.forEach(function (l) {
            l(store);
          });
        };
        var subscribe = function (consumer, noInit) {
          consumers.push(consumer);
          !noInit ? consumer(store) : null;
        };

        this._stores.push({store: store, change: change});
        return subscribe;
      }
      return false;
    },
    dispatch: function (action) {
      if (this._stores.length > 0) {
        this._stores.forEach(function (entry) {
          entry.store.update(action, entry.change);
        });
      }
    }
  }
};

module.exports = {
  create: function () {
    var dispatcher = Dispatcher();

    return {
      createAction: function (type) {
        if (!type) {
          throw new Error('Please, provide action\'s type.');
        } else {
          return function (payload) {
            return dispatcher.dispatch({type: type, payload: pay
load});
          }
        }
      },
      createSubscriber: function (store) {
        return dispatcher.register(store);
      }
    }
  }
};
```

Related links:

- https://github.com/krasimir/react-in-patterns/tree/master/patterns/flux

One way data flow

One-way direction data flow eliminates multiple states and deals with only one which is usually inside the store. To achieve that our Store object needs logic that allows us to subscribe for changes:

```
var Store = {
  _handlers: [],
  _flag: '',
  onChange: function (handler) {
    this._handlers.push(handler);
  },
  set: function (value) {
    this._flag = value;
    this._handlers.forEach(handler => handler())
  },
  get: function () {
    return this._flag;
  }
};
```

Then we will hook our main App component and we'll re-render it every time when the Store changes its value:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    Store.onChange(this.forceUpdate.bind(this));
  }

  render() {
    return (
      <div>
        <Switcher
          value={ Store.get() }
          onChange={ Store.set.bind(Store) }/>
      </div>
    );
  }
}
```

Notice that we are using forceUpdate which is not really recommended.

Normally a high-order component is used to enable the re-rendering. We used forceUpdate just to keep the example simple.

Because of this change the Switcher becomes really simple. We don't need the internal state:

```
class Switcher extends React.Component {
  constructor(props) {
    super(props);
    this._onButtonClick = e => {
      this.props.onChange(!this.props.value);
    }
  }

  render() {
    return (
      <button onClick={ this._onButtonClick }>
        { this.props.value ? 'lights on' : 'lights off' }
      </button>
    );
  }
}
```

The benefit that comes with this pattern is that our components become dummy representation of the Store's data. It's really easy to think about the React components as views (renderers). We write our application in a declarative way and deal with the complexity in only one place.

Related links:

- https://www.startuprocket.com/articles/evolution-toward-one-way-data-flow-a-quick-introduction-to-redux

Presentational and Container components

Problem

Data and logic together.

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {time: this.props.time};
    this._update = this._updateTime.bind(this);
  }

  render() {
    var time = this._formatTime(this.state.time);
    return (
      <h1>{ time.hours } : { time.minutes } : { time.seconds }</
h1>
    );
  }

  componentDidMount() {
    this._interval = setInterval(this._update, 1000);
  }

  componentWillUnmount() {
    clearInterval(this._interval);
  }

  _formatTime(time) {
    var [ hours, minutes, seconds ] = [
      time.getHours(),
      time.getMinutes(),
      time.getSeconds()
    ].map(num => num < 10 ? '0' + num : num);

    return {hours, minutes, seconds};
  }
  _updateTime() {
    this.setState({time: new Date(this.state.time.getTime() + 10
00)});
  }
}

ReactDOM.render(<Clock time={ new Date() }/>, ...);
```

Solution

Let's split the component into two parts - container and presentation.

Container Component

Containers know about data, it's shape and where it comes from. They know details about how the things work or the so called business logic. They receive information and format it so it is easy to use by the presentational component. Very often we use higher-order components to create containers. Their render method contains only the presentational component.

```
// Clock/index.js
import Clock from './Clock.jsx'; // <-- that's the presentationa
l component

export default class ClockContainer extends React.Component {
  constructor(props) {
    super(props);
    this.state = {time: props.time};
    this._update = this._updateTime.bind(this);
  }

  render() {
    return <Clock { ...this._extract(this.state.time) }/>;
  }

  componentDidMount() {
    this._interval = setInterval(this._update, 1000);
  }

  componentWillUnmount() {
    clearInterval(this._interval);
  }

  _extract(time) {
    return {
      hours: time.getHours(),
      minutes: time.getMinutes(),
      seconds: time.getSeconds()
    };
  }

  _updateTime() {
    this.setState({time: new Date(this.state.time.getTime() + 10
00)});
  }
};
```

Presentational component

Presentational components are concerned with how the things look. They have the additional markup needed for making the page pretty. Such components are not bound to anything and have no dependencies. Very often implemented as a stateless functional components they don't have internal state.

```
// Clock/Clock.jsx
export default function Clock(props) {
  var [ hours, minutes, seconds ] = [
    props.hours,
    props.minutes,
    props.seconds
  ].map(num => num < 10 ? '0' + num : num);

  return <h1>{ hours } : { minutes } : { seconds }</h1>;
};
```

The nice things about containers is that they encapsulate logic and may inject data into different renderers. Very often a file that exports a container is not sending out a class directly but a function. For example, instead of using

```
import Clock from './Clock.jsx';
export default class ClockContainer extends React.Component {
  render() {
    return <Clock />;
  }
}
```

We may export a function that accepts the presentational component:

```
export default function (Component) {
  return class Container extends React.Component {
    render() {
      return <Component />;
    }
  }
}
```

Using this technique our container is really flexible in rendering its result. It will be really helpful if we want to switch from digital to analog clock representation.

Related links:

- https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0#.mbglcakmp
- https://github.com/krasimir/react-in-patterns/tree/master/patterns/presentational-and-container
- https://medium.com/@learnreact/container-components-c0e67432e005

Third Party Integration

Mixing 3rd party integrations/libraries with React

In this example we'll see how to mix React and jQuery's UI plugin. We pick tag-it jQuery plugin for the example. It transforms an unordered list to input field for managing tags:

```
<ul>
  <li>JavaScript</li>
  <li>CSS</li>
</ul>
```

To make it work we have to include jQuery, jQuery UI and the tag-it plugin code. It works like that:

```
$('<dom element selector>').tagit();
```

We select a DOM element and call tagit().

The very first thing that we have to do is to force a single-render of the Tags component. That's because when React adds the elements in the actual DOM we want to pass the control of them to jQuery. If we skip this both React and jQuery will work on same DOM elements without knowing for each other. To achieve a single-render we have to use the lifecycle method `shouldComponentUpdate`

Let's say that we want to programmatically add a new tag to the already running tag-it field. Such action will be triggered by the React component and needs to use the jQuery API. We have to find a way to communicate data to Tags component but still keep the single-render approach. To illustrate the whole process we will add an input field to the App class and a button which if clicked will pass a string to Tags component.

```
class App extends React.Component {
  constructor(props) {
    super(props);

    this._addNewTag = this._addNewTag.bind(this);
    this.state = {
      tags: ['JavaScript', 'CSS'],
      newTag: null
    };
  }

  _addNewTag() {
    this.setState({newTag: this.refs.field.value});
  }

  render() {
    return (
      <div>
        <p>Add new tag:</p>
        <div>
          <input type='text' ref='field'/>
          <button onClick={ this._addNewTag }>Add</button>
        </div>
        <Tags tags={ this.state.tags } newTag={ this.state.newTa
g }/>
      </div>
    );
  }
}
```

We use the internal state as a data storage for the value of the newly added field. Every time when we click the button we update the state and trigger re-rendering of Tags component. However, because of shouldComponentUpdate we update nothing. The only one change is that we get a value of the newTag prop which may be captured via another lifecycle method - componentWillReceiveProps

```
class Tags extends React.Component {
  componentDidMount() {
    this.list = $(this.refs.list);
    this.list.tagit();
  }

  shouldComponentUpdate() {
    return false;
  }

  componentWillReceiveProps(newProps) {
    this.list.tagit('createTag', newProps.newTag);
  }

  render() {
    return (
      <ul ref='list'>
        { this.props.tags.map((tag, i) => <li key={ i }>{ tag }
</li>) }
      </ul>
    );
  }
}
```

Related links:

- https://github.com/krasimir/react-in-patterns/tree/master/patterns/third-party

Passing a function to setState

Here's the dirty secret about setState—it's actually asynchronous. React batches state changes for performance reasons, so the state may not change immediately after setState is called. That means you should not rely on the current state when calling setState—since you can't be sure what that state will be! Here's the solution—pass a function to setState, with the previous state as an argument. Doing so avoids issues with the user getting the old state value on access (due to the asynchrony of setState)

Problem

```
// assuming this.state.count === 0
this.setState({count: this.state.count + 1});
this.setState({count: this.state.count + 1});
this.setState({count: this.state.count + 1});
// this.state.count === 1, not 3
```

Solution

```
this.setState((prevState, props) => ({
  count: prevState.count + props.increment
}));
```

Variations

```
// Passing object
this.setState({ expanded: !this.state.expanded });

// Passing function
this.setState(prevState => ({ expanded: !prevState.expanded }));
```

Related links:

- setState() Gate
- Do I need to use setState(function) overload in this case?
- Functional setState is the future of React

Decorators

Decorators (*supported by Babel, in Stage 2 proposal as of 03/17*)

If you’re using something like mobx, you can decorate your class components — which is the same as passing the component into a function. Decorators are flexible and readable way of modifying component functionality.

Non-decorators approach

```
class ProfileContainer extends Component {
  // Component code
}
export default observer(ProfileContainer)
```

With decorators

```
@observer
export default class ProfileContainer extends Component {
  // Component code
}
```

Article:

- Enhancing React components with Decorators

Related:

- Decorators != higher ordered components
- React Decorator example - Module
- What is the use of Connect(decorator in react-redux)
- Decorators with React Components
- Exploring ES7 decorators
- Understanding Decorators

Feature Flags

Enabling Feature flags in React using Redux

```
// createFeatureFlaggedContainer.js
import React from 'react';
import { connect } from 'react-redux';
import { isFeatureEnabled } from './reducers'

export default function createFeatureFlaggedContainer({
  featureName,
  enabledComponent,
  disabledComponent
  }) {
  function FeatureFlaggedContainer({ isEnabled, ...props }) {
    const Component = isEnabled ? enabledComponent : disabledCom
ponent;

    if (Component) {
      return <Component {...props} />;
    }

    // `disabledComponent` is optional property
    return null;
  }

  // Having `displayName` is very useful for debugging.
  FeatureFlaggedContainer.displayName = `FeatureFlaggedContainer
(${ featureName })`;

  return connect((store) => {
    isEnabled: isFeatureEnabled(store, featureName)
  })(FeatureFlaggedContainer);
}
```

```
// EnabledFeature.js
import { connect } from 'react-redux';
import { isFeatureEnabled } from './reducers'

function EnabledFeature({ isEnabled, children }) {
  if (isEnabled) {
    return children;
  }

  return null;
}

export default connect((store, { name }) => {
  isEnabled: isFeatureEnabled(store, name)
})(EnabledFeature);
```

```
// featureEnabled.js
import createFeatureFlaggedContainer from './createFeatureFlagge
dContainer'

// Decorator for "Page" components.
// usage: enabledFeature('unicorns')(UnicornsPage);
export default function enabledFeature(featureName) {
  return (Component) => {
    return createFeatureFlaggedContainer({
      featureName,
      enabledComponent: Component,
      disabledComponent: PageNotFound, // 404 page or something
similar
    });
  };
};
```

```
// features.js
// This is quite simple reducer, containing only an array of fea
tures.
// You can attach this data to a `currentUser` or similar reduce
r.

// `BOOTSTAP` is global action, which contains the initial data
for a page
// Features access usually don't change during user usage of a p
age
const BOOTSTAP = 'features/receive';

export default function featuresReducer(state, { type, payload }
) {
  if (type === BOOTSTAP) {
    return payload.features || [];
  }

  return state || [];
}

export function isFeatureEnabled(features, featureName) {
  return features.indexOf(featureName) !== -1;
}
```

```
// reducers.js
// This is your main reducer.js file
import { combineReducers } from 'redux';

import features, { isFeatureEnabled as isFeatureEnabledSelector
} from './features';
// ...other reducers

export default combineReducers({
  features
  // ...other reducers
});

// This is the important part, access to `features` reducer shou
ld only happens
// via this selector.
// Then you can always change where/how the features are stored.
export function isFeatureEnabled({ features }, featureName) {
  return isFeatureEnabledSelector(features, featureName);
}
```

Related links:

- Feature flags in React
- Gist

The switching component

A switching component is a component that renders one of many components.

Use an object to map prop values to components

```
import HomePage from './HomePage.jsx';
import AboutPage from './AboutPage.jsx';
import UserPage from './UserPage.jsx';
import FourOhFourPage from './FourOhFourPage.jsx';

const PAGES = {
  home: HomePage,
  about: AboutPage,
  user: UserPage
};

const Page = (props) => {
  const Handler = PAGES[props.page] || FourOhFourPage;

  return <Handler {...props} />
};

// The keys of the PAGES object can be used in the prop types to
 catch dev-time errors.
Page.propTypes = {
  page: PropTypes.oneOf(Object.keys(PAGES)).isRequired
};
```

Related links:

- https://hackernoon.com/10-react-mini-patterns-c1da92f068c5

Reaching into a Component

Accessing a component from the parent. eg. Autofocus an input (controlled by parent component)

Child Component

An input component with a focus() method that focuses the HTML element

```
class Input extends Component {
  focus() {
    this.el.focus();
  }

  render() {
    return (
      <input
        ref={el=> { this.el = el; }}
      />
    );
  }
}
```

Parent Component

In the parent component, we can get a reference to the Input component and call its focus() method.

```
class SignInModal extends Component {
  componentDidMount() {
    // Note that when you use ref on a component, it’s a referen
ce to
    // the component (not the underlying element), so you have a
ccess to its methods.
    this.InputComponent.focus();
  }

  render() {
    return (
      <div>
        <label>User name:</label>
        <Input
          ref={comp => { this.InputComponent = comp; }}
        />
      </div>
    )
  }
}
```

Reference:

- https://hackernoon.com/10-react-mini-patterns-c1da92f068c5

Lists Components

Lists and other things that are almost components

Instead of making a separate component for lists I can then generate the results like:

```
const SearchSuggestions = (props) => {
  // renderSearchSuggestion() behaves as a pseudo SearchSuggesti
on component
  // keep it self contained and it should be easy to extract lat
er if needed
  const renderSearchSuggestion = listItem => (
    <li key={listItem.id}>{listItem.name} {listItem.id}</li>
  );

  return (
    <ul>
      {props.listItems.map(renderSearchSuggestion)}
    </ul>
  );
};
```

If things get more complex or you want to use this component elsewhere, you should be able to copy/paste the code out into a new component. Don’t prematurely componentize.

Related links:

- https://hackernoon.com/10-react-mini-patterns-c1da92f068c5

Components for formatting text

Instead of formatting text by calling helper functions inside render, we can create a separate component that handles this.

With Component

Render function is lot cleaner to comprehend as it is just simple component composition.

```
const Price = (props) => {
  // toLocaleString is not React specific syntax - it is a nativ
e JavaScript function used fo formatting
  // https://developer.mozilla.org/en/docs/Web/JavaScript/Refere
nce/Global_Objects/Number/toLocaleString
  const price = props.children.toLocaleString('en', {
    style: props.showSymbol ? 'currency' : undefined,
    currency: props.showSymbol ? 'USD' : undefined,
    maximumFractionDigits: props.showDecimals ? 2 : 0
  });

  return <span className={props.className}>{price}</span>
};

Price.propTypes = {
  className: PropTypes.string,
  children: PropTypes.number,
  showDecimals: PropTypes.bool,
  showSymbol: PropTypes.bool
};

Price.defaultProps = {
  children: 0,
  showDecimals: true,
  showSymbol: true,
};
```

```
const Page = () => {
  const lambPrice = 1234.567;
  const jetPrice = 999999.99;
  const bootPrice = 34.567;

  return (
    <div>
      <p>One lamb is <Price className="expensive">{lambPrice}</P
rice></p>
      <p>One jet is <Price showDecimals={false}>{jetPrice}</Price
></p>
      <p>Those gumboots will set ya back
        <Price
        showDecimals={false}
        showSymbol={false}>
        {bootPrice}
        </Price>
        bucks.
      </p>
    </div>
  );
};
```

Without Component

Less code: But render looks less clean. (Debatable, yeah I understand)

```
function numberToPrice(num, options = {}) {
  const showSymbol = options.showSymbol !== false;
  const showDecimals = options.showDecimals !== false;

  return num.toLocaleString('en', {
    style: showSymbol ? 'currency' : undefined,
    currency: showSymbol ? 'USD' : undefined,
    maximumFractionDigits: showDecimals ? 2 : 0
  });
}

const Page = () => {
  const lambPrice = 1234.567;
  const jetPrice = 999999.99;
  const bootPrice = 34.567;

  return (
    <div>
      <p>One lamb is <span className="expensive">{numberToPrice(
lambPrice)}</span></p>
      <p>One jet is {numberToPrice(jetPrice, { showDecimals: fal
se })}</p>
      <p>Those gumboots will set ya back
        {numberToPrice(bootPrice, { showDecimals: false, showSym
bol: false })}
        bucks.</p>
    </div>
  );
};
```

Reference:

- 10 React Mini Patterns

Share Tracking Logic

Using HOC to share tracking logic across various UX components

eg. Adding analytics tracking across various components.

- Helps to keep it DRY (Do not Repeat Yourself)
- Removing tracking logic etc. from the presentational component makes it well testable, which is key.

```
import tracker from './tracker.js';

// HOC
const pageLoadTracking = (ComposedComponent) => class HOC extends
 Component {
  componentDidMount() {
    tracker.trackPageLoad(this.props.trackingData);
  }

  componentDidUpdate() {
    tracker.trackPageLoad(this.props.trackingData);
  }

  render() {
    return <ComposedComponent {...this.props} />
  }
};

// Usage
import LoginComponent from "./login";

const LoginWithTracking = pageLoadTracking(LoginComponent);

class SampleComponent extends Component {
  render() {
    const trackingData = {/** Nested Object **/};
    return <LoginWithTracking trackingData={trackingData}/>
  }
}
```

Anti-patterns

Familiarizing ourselves with common anti-patterns will help us understand how React works and describe useful forms of refactoring our code.

Props in Initial State

From docs:

> Using props to generate state in getInitialState often leads to duplication of "source of truth", i.e. where the real data is. This is because getInitialState is only invoked when the component is first created.

The danger is that if the props on the component are changed without the component being 'refreshed', the new prop value will never be displayed because the constructor function (or getInitialState) will never update the current state of the component. The initialization of state from props only runs when the component is first created.

Bad

```
class SampleComponent extends Component {
  // constructor function (or getInitialState)
  constructor(props) {
    super(props);
    this.state = {
      flag: false,
      inputVal: props.inputValue
    };
  }

  render() {
    return <div>{this.state.inputVal && <AnotherComponent/>}</div
>
  }
}
```

Good

```
class SampleComponent extends Component {
  // constructor function (or getInitialState)
  constructor(props) {
    super(props);
    this.state = {
      flag: false
    };
  }

  render() {
    return <div>{this.props.inputValue && <AnotherComponent/>}</
div>
  }
}
```

Related links:

- React Anti-Patterns: Props in Initial State
- Why is passing the component initial state a prop an anti-pattern?

Refs over findDOMNode()

Use callback refs over findDOMNode()

Note: React also supports using a string (instead of a callback) as a ref prop on any component, although this approach is mostly legacy at this point.

- More about refs
- Why ref-string is legacy?

findDOMNode(this)

Before:

```
class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this).scrollIntoView();
  }

  render() {
    return <div />
  }
}
```

After

```
class MyComponent extends Component {
  componentDidMount() {
    this.node.scrollIntoView();
  }

  render() {
    return <div ref={node => this.node = node}/>
  }
}
```

findDOMNode(stringDOMRef)

Before

```
class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this.refs.something).scrollIntoView();
  }

  render() {
    return (
      <div>
        <div ref='something'/>
      </div>
    )
  }
}
```

After

```
class MyComponent extends Component {
  componentDidMount() {
    this.something.scrollIntoView();
  }

  render() {
    return (
      <div>
        <div ref={node => this.something = node}/>
      </div>
    )
  }
}
```

findDOMNode(childComponentStringRef)

Before:

```
class Field extends Component {
  render() {
    return <input type='text'/>
  }
}

class MyComponent extends Component {
  componentDidMount() {
    findDOMNode(this.refs.myInput).focus();
  }

  render() {
    return (
      <div>
        Hello,
        <Field ref='myInput'/>
      </div>
    )
  }
}
```

After

```
class Field extends Component {
  render() {
    return (
      <input type='text' ref={this.props.inputRef}/>
    )
  }
}

class MyComponent extends Component {
  componentDidMount() {
    this.inputNode.focus();
  }

  render() {
    return (
      <div>
        Hello,
        <Field inputRef={node => this.inputNode = node}/>
      </div>
    )
  }
}
```

Related links:

- ESLint Rule proposal: warn against using findDOMNode()
- Refs and the DOM

Use Higher order components over Mixins

Simple Example

```
// With Mixin
var WithLink = React.createClass({
  mixins: [React.addons.LinkedStateMixin],
  getInitialState: function () {
    return {message: 'Hello!'};
  },
  render: function () {
    return <input type="text" valueLink={this.linkState('message
')}/>;
  }
});

// Move logic to a HOC
var WithLink = React.createClass({
  getInitialState: function () {
    return {message: 'Hello!'};
  },
  render: function () {
    return <input type="text" valueLink={LinkState(this,'message
')}/>;
  }
});
```

Detailed Example

```
// With Mixin
var CarDataMixin = {
  componentDidMount: {
    // fetch car data and
    // call this.setState({carData: fetchedData}),
    // once data has been (asynchronously) fetched
  }
```

```
};

var FirstView = React.createClass({
  mixins: [CarDataMixin],
  render: function () {
    return (
      <div>
        <AvgSellingPricesByYear country="US" dataset={this.state
.carData}/>
        <AvgSellingPricesByYear country="UK" dataset={this.state
.carData}/>
        <AvgSellingPricesByYear country="FI" dataset={this.state
.carData}/>
      </div>
    )
  }
});

// With HOC
var bindToCarData = function (Component) {
  return React.createClass({
    componentDidMount: {
      // fetch car data and
      // call this.setState({carData: fetchedData}),
      // once data has been (asynchronously) fetched
    },

    render: function () {
      return <Component carData={ this.state.carData }/>
    }
  });
};

// Then wrap your component when you define it.
var FirstView = bindToCarData(React.createClass({
  render: function () {
    return (
      <div>
        <AvgSellingPricesByYear country="US" dataset={this.props
.carData}/>
```

```
        <AvgSellingPricesByYear country="UK" dataset={this.props
.carData}/>
        <AvgSellingPricesByYear country="FI" dataset={this.props
.carData}/>
      </div>
    )
  }
}));
```

Related links:

- Mixins are dead - Long live higher ordercomponents
- Mixins are considered harmful
- Stackoverflow: Using mixins vs components for code reuse
- Stackoverflow: Composition instead of mixins in React

setState() in componentWillMount()

Avoid async initialization in `componentWillMount()`

`componentWillMount()` is invoked immediately before mounting occurs. It is called before `render()`, therefore setting state in this method will not trigger a re-render. Avoid introducing any side-effects or subscriptions in this method.

Make async calls for component initialization in `componentDidMount` instead of `componentWillMount`

```
function componentDidMount() {
  axios.get(`api/messages`)
    .then((result) => {
      const messages = result.data
      console.log("COMPONENT WILL Mount messages : ", messages);
      this.setState({
        messages: [...messages.content]
      })
    })
}
```

Mutating State without setState()

- Causes state changes without making component re-render.
- Whenever setState gets called in future, the mutated state gets applied.

Bad

```
class SampleComponent extends Component {
  constructor(props) {
    super(props);

    this.state = {
      items: ['foo', 'bar']
    };

    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // BAD: We mutate state here
    this.state.items.push('lorem');

    this.setState({
      items: this.state.items
    });
  }

  render() {
    return (
      <div>
        {this.state.items.length}
        <button onClick={this.handleClick}>+</button>
      </div>
    )
  }
}
```

Good

```
class SampleComponent extends Component {
  constructor(props) {
    super(props);

    this.state = {
      items: ['foo', 'bar']
    };

    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // We update using setState() - concat return new array afte
r appending new item.
    this.setState({
      items: this.state.items.concat('lorem')
    });
  }

  render() {
    return (
      <div>
        {this.state.items.length}
        <button onClick={this.handleClick}>+</button>
      </div>
    )
  }
}
```

Related links:

React Design Patterns and best practices by Michele Bertoli.

Using indexes as keys

Keys should be stable, predictable, and unique so that React can keep track of elements.

Bad

In this snippet each element's key will be based on ordering, rather than tied to the data that is being represented. This limits the optimizations that React can do.

```
{todos.map((todo, index) =>
  <Todo
    {...todo}
    key={index}
  />
)}
```

Good

Assuming `todo.id` is unique to this list and stable, React would be able to reorder elements without needing to reevaluate them as much.

```
{todos.map((todo) =>
  <Todo {...todo}
    key={todo.id} />
)}
```

@Reference:

- React docs
- Lin Clark's code cartoon

Spreading props on DOM elements

When we spread props we run into the risk of adding unknown HTML attributes, which is a bad practice.

Bad

This will try to add the unknown HTML attribute `flag` to the DOM element.

```
const Sample = () => (<Spread flag={true} className="content"/>)
;
const Spread = (props) => (<div {...props}>Test</div>);
```

Good

By creating props specifically for DOM attribute, we can safely spread.

```
const Sample = () => (<Spread flag={true} domProps={{className: "
content"}}/>);
const Spread = (props) => (<div {...props.domProps}>Test</div>);
```

Or alternatively we can use prop destructuring with `...rest` :

```
const Sample = () => (<Spread flag={true} className="content"/>)
;
const Spread = ({ flag, ...domProps }) => (<div {...domProps}>Te
st</div>);
```

Note

In scenarios where you use a PureComponent, when an update happens it re-renders the component even if `domProps` did not changed. This is because PureComponent only shallowly compares the objects.

Related links:

- React Design Patterns and best practices by Michele Bertoli.
- In React, children are just props: Kent C. Dodds' Tweet

Handling UX variations for multiple brands and apps

Slides from my talk

A few general coding principles which help write reusable React components

Single Responsibility Principle

In React

Components/Containers code must essentially deal with only one chunk of the UI feature/functionality.

- Eg. Shipping Address component
- Address container (Only has address related fields), Name container (first and last name), Phone component, State, City and Zip code container

In Redux

All API related call go into Redux thunks/other async handling sections (redux-promise, sagas etc)

- The thunks are responsible only for the dispatching action on AJAX begin/fail and complete.
- Any routing has to be dealt with the receiving component via a promise.

Keep it Simple Stupid (KISS)

- Essentially, if the component needs no state - use stateless functions.
- Perf matters: **Stateless fns > ES6 class components > React.createClass()**

- Don’t pass any more props than required {...this.props} only if the list is big -- if not pass individual props.
- Too much flows of control (If-else variations) inside the component is usually a red-flag. This most likely means - need to split up the component or create a separate variation.
- Don’t optimize prematurely - Making the current component reusable with current variations known.

Articles

Building React Components for Multiple Brands and Applications

Using Composition to handle UX variations

Combining smaller reusable components to build a bigger UI blocks.

How do we make sure components are reusable?

- By ensuring our UI components are pure presentational components (dumb)

What does reusable mean?

- No data fetching within the component (do it in Redux).
- If data is required from API - goes into Redux Via redux-thunk API calls are isolated away from the redux containers that deal with the data obtained and pass it on to the dumb component.

If we have a bunch of renderBla() functions within the component which are used in the main component render()

- It's better to move it to separate components. That way it is reusable.

Example

Login page variations

UX variations toggle features + add in additional links/markup.

If the UX variations are involved toggling features within a component + adding minor markup around it

```
import React, { Component } from "react";
import PropTypes from 'prop-types';
import SignIn from "./sign-in";

class MemberSignIn extends Component {
  _renderMemberJoinLinks() {
    return (
      <div className="member-signup-links">
        ...
      </div>
    );
  }

  _routeTo() {
    // Routing logic here
  }

  render() {
    const {forgotEmailRoute,forgotPwdRoute, showMemberSignupLink
s} = this.props;
    return (
      <div>
        <SignIn
          onForgotPasswordRequested={this._routeTo(forgotPwdRout
e)}
          onForgotEmailRequested={this._routeTo(forgotEmailRoute
)}>
          {this.props.children}
          {showMemberSignupLinks && this._renderMemberJoinLinks(
)}
        </SignIn>
      </div>
    );
  }
}

export default MemberSignIn;
```

Related links:

- Slides from my talk: Building Multi-tenant UI with React

Toggle UI Elements

Handling minor UX variations in the component by toggling ON/OFF features.

Modify the component to take in a prop to control it’s behavior.

Gotcha:

Easy to overuse this idea by adding props for every variation.

- Only add in props for features specific to the current feature that the component.
- Basically, not violate the Single Responsibility Principle.

Example

Show/Hide password feature in Login form

```
class PasswordField extends Component {
  render() {
    const {
      password,
      showHidePassword,
      showErrorOnTop,
      showLabels,
      shouldComplyAda
    } = this.props;
    return (
      <div>
        <Password
          field={password}
          label="Password"
          showErrorOnTop={showErrorOnTop}
          placeholder={shouldComplyAda ? "" : "Password"}
          showLabel={showLabels}
          showHidePassword={showHidePassword}
        />
      </div>
    );
  }
}
```

Related links:

- Slides from my talk: Building Multi-tenant UI with React

HOC for Feature Toggles

Using Higher order components (HOC) for UX variations

eg. Toggling features On/Off

```
// featureToggle.js
const isFeatureOn = function (featureName) {
  // return true or false
};

import { isFeatureOn } from './featureToggle';

const toggleOn = (featureName, ComposedComponent) => class HOC e
xtends Component {
  render() {
    return isFeatureOn(featureName) ? <ComposedComponent {...thi
s.props} /> : null;
  }
};

// Usage
import AdsComponent from './Ads'
const Ads = toggleOn('ads', AdsComponent);
```

Higher Order Component - Props proxy

This basically helps to add/edit props passed to the Component.

```
function HOC(WrappedComponent) {
  return class Test extends Component {
    render() {
      const newProps = {
        title: 'New Header',
        footer: false,
        showFeatureX: false,
        showFeatureY: true
      };

      return <WrappedComponent {...this.props} {...newProps} />
    }
  }
}
```

Wrapper Components

Using Wrappers to handle UX/style variations

For Handling Wrapper `<div>` 's and other markup around component, use composition!

When you create a React component instance, you can include additional React components or JavaScript expressions between the opening and closing tags. Parent can read its children by accessing the special this.props.children prop.

```
const SampleComponent = () => {
  <Parent>
    <Child />
  </Parent>
};

const Parent = () => {
  // You can use class 'bla' or any other classes to handle any
style variations for the same markup.
  <div className="bla">
    {this.props.children}
  </div>
};
```

FYI - Wrapper component can also be made accept a tag name and then used to generate the HTML element. However, usually this is not recommended because you can't add attributes/props to it.

```
const SampleComponent = () => {
  <Wrap tagName="div" content="Hello World" />
};

const Wrap = ({ tagName, content }) => {
  const Tag = `${tagName}`     // variable name must begin with
capital letters
  return <Tag>{content}</Tag>
}
```

Related links:

- Slides from my talk: Building Multi-tenant UI with React

Display UI elements in different order

Use a prop to specify order – Map through ReactElements and render it based on order prop.

```
class PageSections extends Component {
  render() {
    const pageItems = this.props.contentOrder.map(
      (content) => {
        const renderFunc = this.contentOrderMap[content];
        return (typeof renderFunc === 'function') ? renderFunc()
 : null;
      }
    );

    return (
      <div className="page-content">
        {pageItems}
      </div>
    )
  }
}
```

Related links:

- Slides from my talk: Building Multi-tenant UI with React

Perf Tips

Key Ideas

- Avoid Reconciliation with shouldComponentUpdate() check
- Use Immutable Data Structures
- Use Production Build
- Profile Components with Chrome Timeline
- Defer computationally expensive tasks in componentWillMount/componentDidMount by using setTimeout and/or requestAnimationFrame

Articles

Optimizing Performance: Docs

Performance Engineering with React

Tips to optimise rendering of a set of elements in React

React.js Best Practices for 2016

shouldComponentUpdate() check

`shouldComponentUpdate` check to avoid expensive re-renders

React Components re-render every time their props or state change. So imagine having to render the entire page every time there in an action. That takes a big load on the browser. That's where ShouldComponentUpdate comes in, whenever React is rendering the view it checks to see if shouldComponentUpdate is returning false/true. So whenever you have a component that's static do yourself a favor and return false. Or if is not static check to see if the props/state has changed.

Bad

```
const AutocompleteItem = (props) => {
  const selectedClass = props.selected === true ? "selected" : ""
;
  var path = parseUri(props.url).path;
  path = path.length <= 0 ? props.url : "..." + path;

  return (
    <li
      onMouseLeave={props.onMouseLeave}
      className={selectedClass}>
      <i className="ion-ios-eye"
         data-image={props.image}
         data-url={props.url}
         data-title={props.title}
         onClick={props.handlePlanetViewClick}/>
      <span
        onMouseEnter={props.onMouseEnter}
      >
        <div className="dot bg-mint"/>
        {path}
      </span>
    </li>
  );
};
```

Good

```
export default class AutocompleteItem extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    if (
      nextProps.url !== this.props.url ||
      nextProps.selected !== this.props.selected
    ) {
      return true;
    }
    return false;
  }

  render() {
    const {props} = this;
    const selectedClass = props.selected === true ? "selected" :
"";
    var path = parseUri(props.url).path;
    path = path.length <= 0 ? props.url : "..." + path;

    return (
      <li
        onMouseLeave={props.onMouseLeave}
        className={selectedClass}>
        <i className="ion-ios-eye"
           data-image={props.image}
           data-url={props.url}
           data-title={props.title}
           onClick={props.handlePlanetViewClick}/>
        <span
          onMouseEnter={props.onMouseEnter}>
          <div className="dot bg-mint"/>
          {path}
        </span>
      </li>
    );
  }
}
```

Related links:

- React Performance optimization
- React rendering misconception

Using Pure Components

Pure Components do shallow equality checks in `shouldComponentUpdate` by default. This is intended to prevent renders when neither props nor state have changed.

Recompose offers a Higher Order Component called `pure` for this purpose and React added `React.PureComponent` in v15.3.0.

Bad

```
export default (props, context) => {
  // ... do expensive compute on props ...
  return <SomeComponent {...props} />
}
```

Good

```
import { pure } from 'recompose';
// This won't be called when the props DONT change
export default pure((props, context) => {
  // ... do expensive compute on props ...
  return <SomeComponent someProp={props.someProp}/>
})
```

Better

```
// This is better mainly because it uses no external dependencie
s.
import { PureComponent } from 'react';

export default class Example extends PureComponent {
  // This won't re-render when the props DONT change
  render() {
    // ... do expensive compute on props ...
    return <SomeComponent someProp={props.someProp}/>
  }
}
})
```

Related links:

- Recompose
- Higher Order Components with Functional Patterns Using Recompose
- React: PureComponent
- Pure Components
- Top 5 Recompose HOCs

Using reselect

Use Reselect in Redux connect(mapState) -- to avoid frequent re-render.

Bad

```
let App = ({otherData, resolution}) => (
  <div>
    <DataContainer data={otherData}/>
    <ResolutionContainer resolution={resolution}/>
  </div>
);

const doubleRes = (size) => ({
  width: size.width * 2,
  height: size.height * 2
});

App = connect(state => {
  return {
    otherData: state.otherData,
    resolution: doubleRes(state.resolution)
  }
})(App);
```

In this above case every time otherData in the state changes both DataContainer and ResolutionContainer will be rendered even when the resolution in the state does not change. This is because the doubleRes function will always return a new resolution object with a new identity. If doubleRes is written with Reselect the issue goes away: Reselect memoizes the last result of the function and returns it when called until new arguments are passed to it.

Good

```
import { createSelector } from 'reselect';
const doubleRes = createSelector(
  r => r.width,
  r => r.height,
  (width, height) => ({
    width: width * 2,
    height: height * 2
  })
);
```

Related links:

- React
- Computing Derived Data: Docs

Styling in React

Here we look into some ideas around using CSS in JS.

If you are pondering over why to use CSS in JS, I highly recommend this talk by Vjeux

Articles

Patterns for style composition in React

Inline style vs stylesheet performance

Styling in stateless UI components

Keep styles separated from the parts of the app that are tied to state. That means routes, views, containers, forms, layouts, etc. should not have any styling or classes in them. Instead, these heavy-lifting components should be composed of primarily stateless functional UI components.

Form component (with no styles/classNames) - just pure composed components

```
class SampleComponent extends Component {
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <Heading children='Sign In'/>
        <Input
          name='username'
          value={username}
          onChange={this.handleChange}/>
        <Input
          type='password'
          name='password'
          value={password}
          onChange={this.handleChange}/>
        <Button
          type='submit'
          children='Sign In'/>
      </form>
    )
  }
}

// Presentational component (with Styles)
const Button = ({
  ...props
  }) => {
  const sx = {
    fontFamily: 'inherit',
    fontSize: 'inherit',
    fontWeight: 'bold',
    textDecoration: 'none',
    display: 'inline-block',
    margin: 0,
    paddingTop: 8,
    paddingBottom: 8,
    paddingLeft: 16,
    paddingRight: 16,
    border: 0,
    color: 'white',
    backgroundColor: 'blue',
    WebkitAppearance: 'none',
    MozAppearance: 'none'
  }

  return (
    <button {...props} style={sx}/>
  )
}
```

Styles module

Generally, hard coding styles values in a component should be avoided. Any values that are likely to be used across different UI components should be split into their own module.

```
// Styles module
export const white = '#fff';
export const black = '#111';
export const blue = '#07c';

export const colors = {
  white,
  black,
  blue
};

export const space = [
  0,
  8,
  16,
  32,
  64
];

const styles = {
  bold: 600,
  space,
  colors
};

export default styles
```

Usage

```jsx
// button.jsx
import React from 'react'
import { bold, space, colors } from './styles'

const Button = ({
  ...props
  }) => {
  const sx = {
    fontFamily: 'inherit',
    fontSize: 'inherit',
    fontWeight: bold,
    textDecoration: 'none',
    display: 'inline-block',
    margin: 0,
    paddingTop: space[1],
    paddingBottom: space[1],
    paddingLeft: space[2],
    paddingRight: space[2],
    border: 0,
    color: colors.white,
    backgroundColor: colors.blue,
    WebkitAppearance: 'none',
    MozAppearance: 'none'
  };

  return (
    <button {...props} style={sx}/>
  )
};
```

Style Functions

Since we’re using JavaScript, we can also employ helper functions for styling elements.

Example 1

A function to create rgba values of black

```
const darken = (n) => `rgba(0, 0, 0, ${n})`;
darken(1 / 8); // 'rgba(0, 0, 0, 0.125)'

const shade = [
  darken(0),
  darken(1 / 8),
  darken(1 / 4),
  darken(3 / 8),
  darken(1 / 2),
  darken(5 / 8),
  darken(3 / 4),
  darken(7 / 8),
  darken(1)
];
// So now,
// shade[4] is 'rgba(0, 0, 0, 0.5)'
```

Example 2

Creating a scale for margin and padding to help keep visual rhythm consistent

```
// Modular powers of two scale
const scale = [
  0,
  8,
  16,
  32,
  64
];

// Functions to get partial style objects
const createScaledPropertyGetter = (scale) => (prop) => (x) => {
  return (typeof x === 'number' && typeof scale[x] === 'number')
    ? {[prop]: scale[x]}
    : null
};
const getScaledProperty = createScaledPropertyGetter(scale);

export const getMargin = getScaledProperty('margin');
export const getPadding = getScaledProperty('padding');
// Style function usage
const Box = ({
  m,
  p,
  ...props
  }) => {
  const sx = {
    ...getMargin(m),
    ...getPadding(p)
  };

  return <div {...props} style={sx}/>
};

// Component usage
const Box = () => (
  <div>
    <Box m={2} p={3}>
      A box with 16px margin and 32px padding
    </Box>
  </div>
);
```

Using npm modules

For more complex color/style transformation logic, it's always good to use it from a separate npm module (or) create one.

Example

For darkening scales in CSS you can use `chroma-js` module

```
import chroma from 'chroma-js'

const alpha = (color) => (a) => chroma(color).alpha(a).css();

const darken = alpha('#000');

const shade = [
  darken(0),
  darken(1 / 8),
  darken(1 / 4)
  // More...
];

const blueAlpha = [
  alpha(blue)(0),
  alpha(blue)(1 / 4),
  alpha(blue)(1 / 2),
  alpha(blue)(3 / 4),
  alpha(blue)(1)
];
```

Base Component

Using a Base Component

There is tremendous amount of flexibility when it comes to composition in React – since components are essentially just functions. By changing style details in a component to props we can make it more reusable.

The color and backgroundColor properties have been moved up to the component’s props. Additionally, we’ve added a big prop to adjust the padding top and bottom.

```
const Button = ({
  big,
  color = colors.white,
  backgroundColor = colors.blue,
  ...props
  }) => {
  const sx = {
    fontFamily: 'inherit',
    fontSize: 'inherit',
    fontWeight: bold,
    textDecoration: 'none',
    display: 'inline-block',
    margin: 0,
    paddingTop: big ? space[2] : space[1],
    paddingBottom: big ? space[2] : space[1],
    paddingLeft: space[2],
    paddingRight: space[2],
    border: 0,
    color,
    backgroundColor,
    WebkitAppearance: 'none',
    MozAppearance: 'none'
  };

  return (
    <button {...props} style={sx}/>
  )
};
```

Usage

```
const Button = () => (
  <div>
    <Button>
      Blue Button
    </Button>
    <Button big backgroundColor={colors.red}>
      Big Red Button
    </Button>
  </div>
);

// By adjusting the props API of the base Button component,
// an entire set of button styles can be created.
const ButtonBig = (props) => <Button {...props} big/>;
const ButtonGreen = (props) => <Button {...props} backgroundColo
r={colors.green}/>;
const ButtonRed = (props) => <Button {...props} backgroundColor=
{colors.red}/>;
const ButtonOutline = (props) => <Button {...props} outline/>;
```

Layout component

We can extend the idea of Base components to create Layout components.

Example

```
const Grid = (props) => (
  <Box {...props}
    display='inline-block'
    verticalAlign='top'
    px={2}/>
);

const Half = (props) => (
  <Grid {...props}
    width={1 / 2}/>
);

const Third = (props) => (
  <Grid {...props}
    width={1 / 3}/>
);

const Quarter = (props) => (
  <Grid {...props}
    width={1 / 4}/>
);

const Flex = (props) => (
  <Box {...props}
    display='flex'/>
);

const FlexAuto = (props) => (
  <Box {...props}
    flex='1 1 auto'/>
);
```

Usage

```
const Layout = () => (
  <div>
    <div>
      <Half>Half width column</Half>
      <Half>Half width column</Half>
    </div>
    <div>
      <Third>Third width column</Third>
      <Third>Third width column</Third>
      <Third>Third width column</Third>
    </div>
    <div>
      <Quarter>Quarter width column</Quarter>
      <Quarter>Quarter width column</Quarter>
      <Quarter>Quarter width column</Quarter>
      <Quarter>Quarter width column</Quarter>
    </div>
  </div>
);
```

Related links:

- Github: React Layout components
- Leveling Up With React: Container Components
- Container Components and Stateless Functional Components in React

Typography Component

We can extend the idea of Base components to create Typography components this pattern helps ensure consistency and keep your styling DRY.

Example

```
import React from 'react';
import { alternateFont, typeScale, boldFontWeight } from './styl
es';

const Text = ({
  tag = 'span',
  size = 4,
  alt,
  center,
  bold,
  caps,
  ...props
  }) => {
  const Tag = tag;
  const sx = {
    fontFamily: alt ? alternateFont : null,
    fontSize: typeScale[size],
    fontWeight: bold ? boldFontWeight : null,
    textAlign: center ? 'center' : null,
    textTransform: caps ? 'uppercase' : null
  };

  return <Tag {...props} style={sx}/>
};

const LeadText = (props) => <Text {...props} tag='p' size={3}/>;
const Caps = (props) => <Text {...props} caps/>;
const MetaText = (props) => <Text {...props} size={5} caps/>;
const AltParagraph = (props) => <Text {...props} tag='p' alt/>;

const CapsButton = ({ children, ...props }) => (
  <Button {...props}>
    <Caps>
      {children}
    </Caps>
  </Button>
);
```

Usage

```
const TypographyComponent = () => (
  <div>
    <LeadText>
      This is a lead with some<Caps>all caps</Caps>.
    It has a larger font size than the default paragraph.
    </LeadText>
    <MetaText>
      This is smaller text, like form helper copy.
    </MetaText>
  </div>
);
```

Using HOC for styling

Sometimes there are isolated UI components that only require a minimal amount of state for interaction, and using them as standalone components is sufficient

eg. Interactions in a Carousel

Example: Carousel

The HOC will have a current slide index and have previous and next methods.

```
// Higher order component
import React from 'react'
// This could be named something more generic like Counter or Cy
cle
const CarouselContainer = (Comp) => {
  class Carousel extends React.Component {
    constructor() {
      super();
      this.state = {
        index: 0
      };
      this.previous = () => {
        const { index } = this.state;
        if (index > 0) {
          this.setState({index: index - 1})
        }
      };

      this.next = () => {
        const { index } = this.state;
        this.setState({index: index + 1})
      }
    }

    render() {
      return (
        <Comp
          {...this.props}
          {...this.state}
          previous={this.previous}
          next={this.next}/>
      )
    }
  }
  return Carousel
};
export default CarouselContainer;
```

Using the HOC

```
// UI component
const Carousel = ({ index, ...props }) => {
  const length = props.length || props.children.length || 0;

  const sx = {
    root: {
      overflow: 'hidden'
    },
    inner: {
      whiteSpace: 'nowrap',
      height: '100%',
      transition: 'transform .2s ease-out',
      transform: `translateX(${index % length * -100}%)`
    },
    child: {
      display: 'inline-block',
      verticalAlign: 'middle',
      whiteSpace: 'normal',
      outline: '1px solid red',
      width: '100%',
      height: '100%'
    }
  };

  const children = React.Children.map(props.children, (child, i)
 => {
    return (
      <div style={sx.child}>
        {child}
      </div>
    )
  });

  return (
    <div style={sx.root}>
      <div style={sx.inner}>
        {children}
      </div>
    </div>
  )
};

// Final Carousel component
const HeroCarousel = (props) => {
  return (
    <div>
      <Carousel index={props.index}>
        <div>Slide one</div>
        <div>Slide two</div>
        <div>Slide three</div>
      </Carousel>
      <Button
        onClick={props.previous}
        children='Previous'/>
      <Button
        onClick={props.next}
        children='Next'/>
    </div>
  )
};

// Wrap the component with the functionality from the higher ord
er component
export default CarouselContainer(HeroCarousel)
```

By keeping the styling separate from the interactive state, any number of carousel variations can be created from these reusable parts.

Usage

```
const Carousel = () => (
  <div>
    <HeroCarousel />
  </div>
);
```

Gotchas

React is intuitive for the most part, but there are quite a few stumbling points which might catch you by surprise. We'll discuss more on them here.

Articles

React Gotchas

Top 5 React Gotchas

Pure Render Checks

In order to preserve performance one needs to consider the creation of new entities in the render method.

Pure render?

With React.js pure render I mean components that implement the shouldComponentUpdate method with shallow equality checks.

Examples of this are the React.PureComponent, PureRenderMixin, recompose/pure and many others.

Case 1

Bad

```
class Table extends PureComponent {
  render() {
    return (
      <div>
        {this.props.items.map(i =>
          <Cell data={i} options={this.props.options || []}/>
        )}
      </div>
    );
  }
}
```

The issue is with `{this.props.options || []}` - it caused all the cells to be re-rendered even for a single cell change. Why?

You see the options array was passed deep down in the Cell elements. Normally this would not be an issue. The other Cell elements would not be re-rendered because they can do the cheap shallow equality check and skip the render entirely but in this case the options prop was null and the default array was used. As you should know the array literal is the same as new Array() which creates a new array instance. This completely destroyed every pure render optimization inside the Cell elements. In Javascript different instances have different identities and thus the shallow equality check always produces false and tells React to re-render the components.

Good

```
const defaultval = [];  // <---  The fix (defaultProps could als
o have been used).
class Table extends PureComponent {
  render() {
    return (
      <div>
        {this.props.items.map(i =>
          <Cell data={i} options={this.props.options || defaultv
al}/>
        )}
      </div>
    );
  }
}
```

Case 2

Similar issue with using functions in render() as well

BAD

```
class App extends PureComponent {
  render() {
    return <MyInput
      onChange={e => this.props.update(e.target.value)}/>;
  }
}
```

Bad again

```
class App extends PureComponent {
  update(e) {
    this.props.update(e.target.value);
  }

  render() {
    return <MyInput onChange={this.update.bind(this)}/>;
  }
}
```

^^In both cases a new function is created with a new identity. Just like with the array literal. We need to bind the function early

Good

```
class App extends PureComponent {
  constructor(props) {
    super(props);
    this.update = this.update.bind(this);
  }

  update(e) {
    this.props.update(e.target.value);
  }

  render() {
    return <MyInput onChange={this.update}/>;
  }
}
```

Bad

```
class Component extends React.Component {
  state = {clicked: false};

  onClick() {
    this.setState({clicked: true})
  }

  render() {
    // Options object created each render if not set
    const options = this.props.options || {test: 1};

    return <Something
      options={options}
      // New function created each render
      onClick={this.onClick.bind(this)}
      // New function & closure created each render
      onTouchTap={(event) => this.onClick(event)
    />
  }
}
```

Good

```
class Component extends React.Component {
  state = {clicked: false};
  options = {test: 1};

  onClick = () => {
    this.setState({clicked: true})
  };

  render() {
    // Options object created once
    const options = this.props.options || this.options;

    return <Something
      options={options}
      onClick={this.onClick} // Function created once, bound once

      onTouchTap={this.onClick} // Function created once, bound
once
    />
  }
}
```

Related links:

- https://medium.com/@esamatti/react-js-pure-render-performance-anti-pattern-fb88c101332f
- https://github.com/nfour/js-structures/blob/master/guides/react-anti-patterns.md#pure-render-immutability
- Optimizing React Rendering

Synthetic events in React

Inside React event handlers, the event object is wrapped in a SyntheticEvent object. These objects are pooled, which means that the objects received at an event handler will be reused for other events to increase performance. This also means that accessing the event object’s properties asynchronously will be impossible since the event’s properties have been reset due to reuse.

The following piece of code will log null because event has been reused inside the SyntheticEvent pool:

```
function handleClick(event) {
  setTimeout(function () {
    console.log(event.target.name);
  }, 1000);
}
```

To avoid this you need to store the event’s property you are interested in inside its own binding:

```
function handleClick(event) {
  let name = event.target.name;
  setTimeout(function () {
    console.log(name);
  }, 1000);
}
```

Related links:

- React/Redux: Best practices & gotchas
- React events in depth w/ Kent C. Dodds, Ben Alpert, & Dan Abramov

Related Links

- React in Patterns by krasimir
- React Patterns by planningcenter
- reactpatterns.com
- 10 React Mini-patterns