React

Wednesday, August 26, 2015 5:07 PM

General structure of applications (header + content + footer)

3:58 HiDeoo: Adam13531 From what I remember, Bot Land is closer to example #1 where the way I'm used to is more of the example #2 where the child is never directly doing something related to the layout or if it should, it's done via composition of elements provided by the parent.
https://bpaste.net/show/248b23fda519

Example 1 - each component needs <Header> and <Footer>

App
<Route path="/news" component={News} />
<Route path="/store" component={Store} />

News
<Header />
<div>content news</div>
<Footer />

Store
<Header />
<div>content store</div>
<Footer />

Example 2 - the app has <Header> and <Footer> so that each component doesn't need it

App
<Header />
<Route path="/news" component={News} />
<Route path="/store" component={Store} />
<Footer />

News
<div>content news</div>

Store
<div>content store</div>

Note: if you wanted one page to have a Header or a Footer but not all pages, you could use functional composition so that the children can choose what they want, but they don't need to handle the implementation details or even import those files (see this section of the note you're reading).

Extending PureComponent instead of Component

Explanation by HiDeoo:

> *Ho no no, PureComponent is used with props & state.*
>
> *You use it basically when all props & all state changes and only that should trigger a re-render aka if given the same props & state, it always render the same thing.*
>
> *So if you want your component to only re-render when something change in the props or state, it's easier to extends PureComponent vs implementing shouldComponentupdate and checking everything.*

What PureComponent does, it just don't call shouldComponentUpdate of the component and just do a shallow comparison of the state & nextState and props & nextProps. If the shallow comparison return true, it re-renders, if not, it does not re-render. That's it. Nothing more.

setState with callback

If your setState call depends on the previous state, specify a callback as the argument so that you don't run into errors around scheduled updates.

```
this.setState((prevState) => ({
  showPopup: !prevState.showPopup
}));
```

Advanced React lectures (reference)

UPDATE: never mind, didn't realize this cost $250. I still wrote some notes for the only real free video of the set.

Tabs: the biggest thing this lecture addresses is when you keep adding props to a component to control how it will render when that's something you should be controlling yourself. The example he covers is that instead of adding a prop to a tablist like "renderTabsOnBottom={true}" or "disableTabIndices={[1, 3]}", you can change it so that you're the one controlling where tabs render and you can just put "isDisabled={true}" on a tab that should be disabled instead of on its parent.

For a more in-depth example taken from the lecture:

```
// Here's our structure. <Tabs> should keep track of which tab is selected so that we don't need to keep passing props all over the place. <Tabs> has it in its state, but how do we get that to the children without having to specify anything as a caller?
<Tabs>
    <TabList>
        <Tab>Blah</Tab>
        …more tabs
    </TabList>
</Tabs>

// Solution: do this from <Tabs>
const children = React.Children.map(this.props.children, (child) => {
  return React.cloneElement(child, {
    activeIndex: this.state.activeIndex
  });
});
```

Simple stuff

Counting children (reference)

In React, use React.Children.count(this.props.children) instead of something like this.props.children.length.

Passing a bunch of extra props to a component

Let's say you have a Button class that you made just so that you don't have to specify className='foo' everywhere. So you have this:

```
const Button = (props) => {
    return <button className='foo'>props.text</button>;
}
export default Button;
```

You could use with code like this:

```
<Button text='Click me'/>
```

However suppose you wanted to specify an "extra" class name, or whether it's disabled, or just some other props that you don't feel like having Button check. You could change it to something like this:

```
const Button = (props) => {
    return <button className='foo' {...props.extraProps}>
    props.text</button>;
}
```

Now you don't have to worry about maintaining each individual prop.

Setting ref from child

(^I couldn't come up with a better title for this, so just read this description)

Description of the problem: in Bot Land, I had a CosmeticsInventory and a HardwareInventory and I wanted to refactor them into a generic Inventory class. The Inventory would be responsible for scrolling particular items into view (e.g. if you'd filtered to that item or the tutorial wanted you to select a specific one). However, CosmeticsInventory or HardwareInventory would construct specific items for the Inventory to use, but I couldn't figure out how to pass a ref from HardwareInventory to Inventory.

Code showing the problem:

```
class HardwareInventory {
    makeItem() {
        return <HardwareInvItem id='lasers' />;
    }

    render() {
        return <Inventory item={this.makeItem()}/>;
    }
}

class Inventory {
    constructor() {
        this.refToItem = null; // I want to set this to the
<HardwareInvItem> that is passed as this.props.item
    }

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

class HardwareInvItem extends Component {
    constructor(props) {
        super(props);

        this.draggable = null;
    }
```

```
        getDraggableDomElement() {
            return this.draggable;
        }

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

The solution was something that HiDeoo had suggested (original bpaste):
Summary: pass a function to the child that will set the ref in the parent class.

```
    class Inventory {
        constructor() {
            this.refToItem = null;
        }

        setInnerRef(innerRef) {
            this.refToItem = innerRef;
        }

        render() {
            // Pass a function that will allow the child to set a reference
    in this
            // class.
            return <div>
                {React.cloneElement(this.props.item, { setInnerRef:
    this.setInnerRef })}
            </div>;
        }
    }

    class HardwareInventory {
        makeItem() {
            return <HardwareInvItem id='lasers' />;
        }

        render() {
            return <Inventory item={this.makeItem()}/>
        }
    }

    class HardwareInvItem extends Component {
        constructor(props) {
            super(props);

            this.draggable = null;
        }

        componentDidMount() {
            this.props.setInnerRef(this.draggable);
        }

        getDraggableDomElement() {
            return this.draggable;
        }

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

Basic composition

Here's a really simple class [that doesn't need to be a class and shouldn't have a useless constructor] that simply renders all children inside a div with a particular CSS class:

```
class LoginSection extends Component {
    constructor(props) {
        super(props);
    }

    render() {
        const className = this.props.isTopSection
            ? styles.loginViewTopPart
            : styles.loginViewBottomPart;

        return (
            <div
                className={className}
                >
                {this.props.children}
            </div>
        );
    }
}

const enhance = onlyUpdateForKeys([
    'isTopSection',

    // If we don't do this, then we won't detect when the children
change.
    'children'
]);

LoginSection.defaultProps = {
    isTopSection: true
};

export default enhance(LoginSection);
```

As you can see, any children inside a <LoginSection> will get inserted inside of the div, so a caller simply has to do something like this:

```
<LoginSection isTopSection={true}>
    <div>I'm on top</div>
</LoginSection>
<LoginSection isTopSection={false}>
    <div>I'm on bottom</div>
</LoginSection>
```

This is powerful because you don't need special classes for top vs. bottom. You simply compose components as you wish. Unit testing these should be pretty easy too since it can all be contained to a single class/file.

React patterns, techniques, tips and tricks

https://github.com/vasanthk/react-bits

A different composition pattern: "function as child" (render callback)

FYI: this is what downshift uses
Around July 2017, I was coding on-stream and I wasn't totally sure how to solve a particular problem. HiDeoo suggested something that I ended up not using because it didn't match my use-case exactly, but I wanted to write it down at least.

HiDeoo: Adam13531 Rough & quick mockup, the idea, make it easily customizable by using a function as a React children https://bpaste.net/show/57af8eff3f28
HiDeoo: The idea is to provide all login elements available in the parent and you pick what you want, build your input like you want and render it
HiDeoo: In the end it's also easier to unit test because you only have basically to test the individual components, not the whole mess trying to mix things together
[later] HiDeoo: Well, you have to import everything in the parent whereas in the other case, the component responsible for showing a LandingNavEntry provide everything that you can use to render it properly. It's isolated.

[UPDATE: the actual code I ended up implementing is at the bottom of this section]

```
------------------------------

USAGE

------------------------------

Only a input text

<LoginInput>
  {({ LoginTextField }) =>
    <div>
      <LoginTextField value="blabla" onChange={_.noop} />
    </div>}
</LoginInput>

Only a button
```

```
<LoginInput>
  {({ LoginButton }) =>
    <div>
      <LoginButton title="Go go go" onClick={_.noop} />
    </div>}
</LoginInput>

Secure Text Field + button

<LoginInput>
  {({ LoginTextField, LoginButton }) =>
    <div>
      <LoginTextField secure value="blabla" onChange={_.noop} />
      <LoginButton title="Go go go" onClick={_.noop} />
    </div>}
</LoginInput>

------------------------------

CODE

------------------------------


/**
 * LoginButton Component.
 */
const LoginButton = ({ onClick, title }) =>
  <Flex>
    <input type="button" value={title} className={style.button}
onClick={onClick} {...otherProps} />;
  </Flex>

export default LoginButton;

/**
 * LoginTextField Component.
 */
const LoginTextField = ({ value, onChange, secure, ...otherProps }) =>
  <Flex>
    <input type={secure ? 'password' : 'text'} value={value}
className={style.input} onChange={onChange} {...otherProps} />;
  </Flex>

export default LoginTextField;

/**
 * LoginInput Component.
 */
const LoginInput = ({ children }) => {
    return (
      <Flex>
        {children({ LoginTextField, LoginButton })}
      </Flex>
    );
}

/**
 * React Props.
 * @type {Object}
 */
LoginInput.propTypes = {
  children: PropTypes.function.isRequired,
};

export default LoginInput;
```

Here's how it works:
LoginInput takes in a function that will be passed the possible classes that you can render, so as a user of LoginInput, you can use or ignore whichever classes you want

> Note: the reason "children" was used and expected to be a function is so that callers can specify that function like this:
>
> ```
> <LoginInput>
>     FUNCTION GOES HERE
> </LoginInput>
> ```

This is helpful if you have lots of optional components or you want different locations in the DOM, e.g.

```
<LandingNavLogEntry>
  {({LandingNavTitleBar, LandingNavContent}) => {
    return (
      <div>
        <LandingNavTitleBar />    <--- This could be below the content if we wanted
        <LandingNavContent />
      </div>
    );
  }}
</LandingNavLogEntry>
```

I did use this in Bot Land for CosmeticSlot, HardwareSlot, and MiniHardwareSlot. There's a Slot class that looks something like this:

```
const ContentTitle = ({ children, ...otherProps }) =>
    <div {...otherProps}>
        {children}
    </div>;

const Header = ({ children }) =>
    <Flex>
        <div className={styles.slotHeader}>
            {children}
        </div>
        <div className={styles.slotHeaderExtra}/>
    </Flex>;

class Slot extends Component {
    render() {
        return (
            <div
                onClick={this.props.onClick}
```

```
                className={this.props.className}>
                {this.props.children({ Header, ContentTitle })}
            </div>
        );
    }
}

export default Slot;
```

Then, to use this, I did something like this for HardwareSlot

```
class HardwareSlot extends Component {
    render() {
        return (
            <Slot>
                {({ Header, ContentText }) =>
                    <div>
                        <Header>
                            {this.props.displayName}
                        </Header>
                        <ContentText>
                            {this.props.description}
                        </ContentText>
                    </div>
                }
            </Slot>
        );
    }
}
```

Notes:

- If you want to use parent state in the children, just pass it to the children. The reason I explicitly mention this is because at one point, I had a child binding functions in such a way that made it look like this wouldn't be the correct way of passing state around.

Free React training

https://reacttraining.com/online/react-fundamentals

Costs nothing to register. It was written by the guys who make React-Router.

Resources on React lifecycle methods

https://engineering.musefind.com/react-lifecycle-methods-how-and-when-to-use-them-2111a1b692b1
http://imgh.us/react-lifecycle.svg

Only render a component based on a boolean

There are a bunch of different techniques that I've tried for this:

Conditionally render from a function

```
renderComponent() {
    if (!this.props.componentIsVisible) {
        return null;
    }

    return (
        <SomeComponent/>
    );
}

render() {
    return (
        <div>
            {this.renderComponent()}
        </div>
    );
}
```

The problem here is that if "SomeComponent" shows up from multiple parent components, each parent needs this logic to control visibility of the child.

Make a logical AND around rendering

```
render() {
    return (
        <div>
            {this.props.componentIsVisible && <SomeComponent/>}
        </div>
    );
}
```

I've found this to be a bit messy when reading through the render function due to how tabbing works.

Let the component return null if it's not visible

```
// parent.js
render() {
    return (
        <div>
            {<SomeComponent isVisible={this.props.componentIsVisible}/>}
        </div>
    );
}

// somecomponent.js
render() {
    if (!this.props.isVisible) {
        return null;
    }

    // return regular elements
}
```

The benefit to this is that the parent is very easy to read, and multiple parent components can use SomeComponent without having to worry about a conditional in the parent logic; they just pass the prop regardless and the child figures out how to conditionally render.

Bypassing writing a component with the same props everywhere by using HoC (reference)

I ran into a problem where there was a component that was used from many different containers. For example, I have a header component that has functions like "settings", "store", and "logout", and I didn't want to have to do this from every single page:

```
<Route path='landingPage'
   component={LandingPage}
   onLogout={logout}
   />
<Route path='news'
   component={News}
   onLogout={logout}
   />
<Route path='somethingElse'
   component={SomethingElse}
   onLogout={logout}
   />
```

HiDeo suggested that I do this:

```
/**
 * PackOpener Higher-Order Component.
 */
export const withPacks = (ComposedComponent) => {
  const composition = props => <ComposedComponent {...props} />;

  composition.displayName = `withPacks(${ComposedComponent.displayName ||
ComposedComponent.name})`;

  return connect(null, { openPackUI, openPack, sellPack })(composition); // note: connect comes
from "react-redux"
};
```

Anywhere you can use a decorator to get this new actions as props

```
@withPackOpener
export default class Thingy extends Component {
  ....
}
```

Or the basic way by wrapping

```
withPackOpener(Thingy);
```

HiDeoo: I use that a lot for example for notifications, I have a Notifications HoC and in any component that need to raise a notificaiton, I use the @withNotifications decorator and can use this.props.addError('something went wrong') in the component

I asked about the situation where I have exactly the same header in each page:

HiDeoo: Either have its own container or decorators + HoC but if you have 30+ header, forgetting the HoC wrapping in one may screw everything up ^^

HiDeoo: And instead of rendering <Header /> you render <HeaderContainer /> which then render <Header /> with all the proper props

shouldComponentUpdate

A good article on shouldComponentUpdate here

Conclusions:

- Don't just define shouldComponentUpdate by default. Instead, think "*do* I need to override this?" The article says "[only] Use shouldComponentUpdate once you've measured that it provides a perceivable performance improvement."
- shouldComponentUpdate becomes another maintenance point. Speaking from experience, I've been burned a few times because I renamed a prop but forgot to update "shouldUpdateComponent" (in the form of onlyUpdateForKeys), so the component didn't update when this renamed prop changed and I didn't notice it for a while since I didn't have tests.

Intro

Getting React up and running was easy thanks to this.

I used Gulp to build the JSX files and Express as my framework:

First:

```
npm install --save-dev gulp-jsx
```

Then, modify your gulpfile.js:

```
var gulp = require('gulp');
var jsx = require('gulp-jsx');

gulp.task('jsx', function() {
  return gulp.src('views/**/*.js', {base: './'})
    .pipe(jsx({
      factory: 'React.createElement'
    }))
    .pipe(gulp.dest('dist'));
});
```

Then, in my index.jade, I included this in the head

```
script(type='text/javascript', src='https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react.js')
```

Then I put this in the body

```
  div#example
  //- This has to come after #example exists.
  script(type='text/javascript', src='dist/views/helloworld.js')
```

In the end, my index.jade looked like this:

```
doctype html
html
  head
    title Learning ES6
    script(type='text/javascript', src='https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react.js')
    //- script(type='text/javascript', src='https://cdnjs.cloudflare.com/ajax/libs/react/0.13.3/react-
with-addons.js')
    script(type='text/javascript', src='dist/src/main.js')
  body(onload='main.start()' style="overflow: hidden;")
    div#example
    //- This has to come after #example exists.
    script(type='text/javascript', src='dist/views/helloworld.js')
```

Then I created views/helloworld.js:

```
React.render(
  <h1>Hello, world!</h1>,
  document.getElementById('example')
);
```

Next, I did "gulp jsx" to build, which put helloworld.js into dist/views/helloworld.js. I had already set up app.js to use "/dist" explicitly with this line:

```
app.use('/dist', express.static(path.join(__dirname, 'dist')));
```

React basics

Component communication (parent/child, parent/parent): http://andrewhfarmer.com/component-communication/

Case-sensitivity: JSX elements must start with a capital letter.

JSX does this to distinguish between JSX and HTML elements.

"this.props" is owned by the parent. They are immutable. YOU HEAR THAT? Don't pass things in as props unless you want them existing forever in that object.

"this.props" being immutable means that you can't add/remove to it. HOWEVER, the parent can at some point change the props entirely, and that would trigger a re-render.

"this.state" is mutable state that is owned by a particular component.

Note: make sure you only change it through this.setState()! That way React knows to trigger a re-render.

"this.refs" refers to any child components you added that have ref="something". They're basically IDs. For example:

```
var CommentForm = React.createClass({
  handleSubmit: function(e) {
    e.preventDefault();
    var text = React.findDOMNode(this.refs.myRefExample).value.trim();
  },
  render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input type="text" placeholder="Say something..." ref="myRefExample" />
        <input type="submit" value="Post" />
      </form>
    );
  }
});
```

The way you would pass information from a child to a parent is using a callback as a prop. In short: the parent passes a callback to the child, which the child stores in this.props.myCallback. When the child wants to communicate, it calls the callback.

1. In the parent, I render something like <CommentForm onCommentSubmit={this.handleCommentSubmit} />
2. I also define the handleCommentSubmit function in the parent with any arguments I may want.
3. In the child, I can simply call this.props.onCommentSubmit now with any arguments in order to call back to the parent.

NOTE: this methodology of passing callbacks around is also used in Redux.

Make sure that you are calling setState on your data instead of manually updating parts of it! For example, if you have a state like

```
{
    comments:
        [
            {name: 'Adam', message: 'hi'},
            {name: 'Adam', message: 'hi again'},
        ]
}
```

...don't just do comments.push(newMessage). Instead, do a setState on the updated array.

In their own tutorial, Facebook suggests doing it this way:

```
var comments = this.state.data;
var newComments = comments.concat([comment]); // not totally sure why they don't just do a .push
this.setState({data: newComments});
```

For lifecycle information, read here. For example, componentDidMount is called only once, only on the client, immediately after the initial rendering occurs. At this point in the lifecycle, the component has a DOM representation, meaning all of its children already had their componentDidMount functions called.

The lifecycle is useful because sometimes you'll want to use, say, a JQuery plug-in without having to rebuild it in idiomatic React. That's when you would hook into lifecycle functions.

Bind functions at creation time, not at call time. For example:

```
// Note: I adapted this code from the redux\examples\real-world\containers\App.js code.
  constructor(props) {
    super(props);
    this.handleAdd = this.handleAdd.bind(this); // Yes, they name their variable the same as their function
  }

handleAdd(someNewItem) {
  // code goes here to handle adding.
}

  render() {
    return (
      <button onClick={this.handleAdd}>Add Item</button>    <-- GOOD. You bound it in the constructor
      <button onClick={this.handleAdd.bind(this)}>Add Item</button>    <-- BAD. You're binding at call time (note: for you to even consider this code, "this.handleAdd = this.handleAdd.bind(this)" wouldn't have appeared in the constructor)
    );
  }
```

To style components, you can use CSS inline (although this is the worst way to do it... you should look up online how to do this better, perhaps with an NPM module like https://github.com/blakeembrey/free-style).

```
class ChatLog extends Component {
    render() {
        const divStyle = {
            backgroundColor: '#ccc',
            height: this.props.height + 'px',
            overflow: 'auto'
        };
        return (
            <div style={divStyle}>
            </div>
        );
    }
}
```

To pass in values (e.g. widths and heights), have the parent pass it in via props.

Context

If you're sick of passing props from main --> Root --> App --> Container --> Component, you can use context, although it is still experimental (1/12/2016):
https://facebook.github.io/react/docs/context.html

React-redux

https://github.com/rackt/react-redux/

As I can see from redux\examples, I should have folders like:
actions - just ActionCreators, e.g. addTodo or deleteTodo
components - dumb React components
constants - literally just a whole bunch of   export const ADD_TODO = 'ADD_TODO';
containers - smart React components
reducers - the big switch/case blobs. Because these are pure functions, you can always have them on their own.

"Dumb" components are middle and leaf components. They only read from props and invoke callbacks from props. They don't need Redux at all.
For example, a Picker may be a dumb component. When you create it, just pass an onChange handler, e.g. <Picker value={selectedOption} onChange={this.handleChange} options={['Strawberry', 'Cherry']} />
"Smart" components are top-level or route-handling components. They actually make use of React-Redux, meaning they subscribe to the Redux state, and they dispatch Redux actions to change data.

Note that sometimes you'll want dumb components to be able to modify the Redux state while maintaining the "dumb" property. I.e. the dumb component can't know Redux exists, yet it should still be able to modify Redux state. That's where bindActionCreators comes in; all it does is take in an object of ActionCreators and binds them to the dispatch. This technically turns the dumb component INTO a smart component (even according to the official documentation), but the component can still be used as a dumb component.

Recompose (reference)

syntonic8 told me about this. It's like lodash for React.

Recompose + refs

I never figured out how to have a parent be able to access the functions of a child through a ref when "onlyUpdateForKeys" is involved. It involves "toClass" and "hoistStatics" probably, but I just ended up getting rid of onlyUpdateForKeys altogether. I could have also written my own shouldComponentUpdate.

Barring all of that, it's maybe possible to use withHandlers (reference).

HiDeoo: @Adam13531 Tried it over breakfast (aka over coffee), the solution is indeed withHandlers() to mix recompose + refs. Got an example working if you want to check it / add it to your Recompose notes https://codesandbox.io/s/Z4r2G6n45 (I console.log the ref in the example to make sure it works so toggle the dev tools)

```
import React, { Component } from 'react';
import { compose, onlyUpdateForKeys, withHandlers } from 'recompose';

class Child extends Component {
  componentDidMount() {
    this.props.setInnerRef(this.getInnerDiv());
  }

  getInnerDiv() {
    return this.innerDiv;
  }

  render() {
    return (
      <div>
        <div ref={(div) => { this.innerDiv = div; }}>
          Child i:{this.props.i} - j: {this.props.j}
        </div>
      </div>
    );
  }
}

const enhance = compose(
  withHandlers(() => {
    let innerRef;

    return {
      setInnerRef: () => (ref) => innerRef = ref,
      getInnerRef: () => () => innerRef,
    }
  }),
  onlyUpdateForKeys(['j'])
)

export default enhance(Child);
```

Then, in the parent, get a ref and call this.child.handlers.getInnerRef();

Animations

I was trying to accomplish something pretty specific for Bot Land in June 2016. I wanted a layout that had content like "[A][B][C][D][E]", but only three of those "blocks" were visible at any given time. By pressing buttons on the page, you could show either "ABC", "BCD", or "CDE". I wanted them to animate the sliding.

EveryJuan posted this example:

```
<!-- EJ here with another amazing snippet -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <style>
      /*All of this is just boilerplate to get the divs in the right place*/
      html, body {
        height: 100%;
        width: 100%;
        padding: 0;
        margin: 0;
        overflow: hidden;
      }
      .l-header, .l-footer {
        width: 100%;
        height: 10%;
        background: red;
      }
      .l-content {
        height: 80%;
        width: 100%;
        overflow: hidden;
      }
      .l-content div {
        float: left;
        height: 100%;
        text-align: center;
      }
      .p-left-opt, .p-right-opt {
        background: orange;
        width: 20%;
        height: 80%;
        top: 10%;
      }
      .p-left, .p-right {
        background: blue;
        width: 20%;
      }
      .p-center {
        left: 20%;
        background: green;
        width: 60%;
        height: 80%;
      }
      .l-footer {
        position: absolute;
        bottom: 0;
      }
    </style>
    <style>
      /*ok here we go*/
      .p-left-opt {
        position: absolute;
        left: -20%;
      }
      .p-right-opt {
        position: absolute;
        right: -20%;
      }
      .l-content, .p-left-opt, .p-right-opt {
        transition: all 1s;
      }
      body.t-left .l-content {
        transform: translateX(20%);
      }
      body.t-left .p-left-opt {
        transform: translateX(100%);
      }
      body.t-right .l-content {
        transform: translateX(-20%);
      }
      body.t-right .p-right-opt {
        transform: translateX(-100%);
      }
    </style>
    <title>How to botland</title>
  </head>
  <body>
    <div class="l-header"></div>
    <div class="p-left-opt"></div>
    <div class="p-right-opt"></div>
    <div class="l-content"><!--
    --><div class="p-left"><button onclick="left()">Left Toggle</button></div><!--
    --><div class="p-center"></div><!--
    --><div class="p-right"><button onclick="right()">Right Toggle</button></div><!--
    --></div>
    <div class="l-footer"></div>
  </body>
</html>

<script>
```

```
  var hasLeft = false;
  var hasRight = false;

  var left = function(){

    if(hasLeft){
      hasLeft = false;
      return document.body.className = '';
    }

    document.body.className = 't-left';
    hasLeft = true;

  };

  var right = function(){

    if(hasRight){
      hasRight = false;
      return document.body.className = '';
    }

    document.body.className = 't-right';
    hasRight = true;

  };

</script>
```

Utrolig suggested something very similar: https://jsfiddle.net/ju8uke5t/

The reason why they did the above is that you can't do a CSS transition on an element unless it exists in the DOM, so you can't just create the hidden panel on the fly and then transition it in (unless you did so in two discrete steps).

Also, "transform: translateX(500px)" does not push all other elements to the right unless they ALSO have the transform on them. This means that if you have this layout: <div style="display:inline-block">A</div><div style="display:inline-block">B</div> that "B" won't be pushed over if you give "A" transform:translateX(50%).

Alagwin made a solution using just a flexbox (reference)

LESS CSS:

```
html,body {
 height:100%;
 min-height:100%;
}

.frame {
 display:flex;
 overflow:hidden;
 flex-direction:column;
 min-height:100%;

 .top, .bottom {
  height:30px;
  background-color:orange;

 }
}

.inner-content {
 display:flex;
 flex:1 1 auto;
 align-items: stretch;
 align-content: stretch;
 flex-direction: row;
 height:100%;
 border:1px solid black;



 div {
  flex: auto;
  width:0;
  flex-basis: auto;
  white-space:nowrap;
  transition: all 0.5s;
 }


 .far_left {
  background-color:red;
  /* opacity:0; */
  flex-grow:0;
  width:0;
```

```
    img {
     width:150px;
    }
  }
  .left {
    background-color:green;
  }
  .center{
    background-color:purple;
    flex: 2
  }
  .right {
    background-color:yellow;
  }
  .far_right {
    flex-grow:0;
    width:0;
    background-color:tan;
  }


}
```

JavaScript:

```
$('.left').on('click', () => {
  $('.far_left').css('flexGrow','1');
//   $('.far_left').css('opacity','1');

  $('.right').css('flexGrow','0');
//   $('.right').css('opacity','0');
});


$('.far_left').on('click', () => {
  $('.far_left').css('flexGrow','0');
//   $('.far_left').css('opacity','0');

  $('.right').css('flexGrow','1');
//   $('.right').css('opacity','1');
});


$('.right').on('click', () => {
  $('.far_right').css('flexGrow','1');
//   $('.far_left').css('opacity','1');

  $('.left').css('flexGrow','0');
//   $('.right').css('opacity','0');
});


$('.far_right').on('click', () => {
  $('.far_right').css('flexGrow','0');
//   $('.far_left').css('opacity','0');

  $('.left').css('flexGrow','1');
//   $('.right').css('opacity','1');
});
```

HTML:

```
<!DOCTYPE html>
<html>
<head>
<script src="https://code.jquery.com/jquery-2.2.4.js"></script>
<script src="https://fb.me/react-with-addons-15.1.0.js"></script>
<script src="https://fb.me/react-dom-15.1.0.js"></script>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>JS Bin</title>
</head>
<body>

  <div class="frame">
      <div class="top">Top</div>

    <div class="inner-content">
      <div class="far_left">
        <p>Far Left</p>
        <img src="http://media.vogue.com/r/c_1,h_4000,w_3280/2015/06/01/beauty-mark-kate-upton.jpg" />
      </div>
      <div class="left">Left</div>
      <div class="center">Center</div>
      <div class="right">Right</div>
      <div class="far_right">Far Right</div>
    </div>
    <div class="bottom">Bottom</div>
  </div>

</body>
</html>
```

- Facebook page on animation: https://facebook.github.io/react/docs/animation.html
- I could use react-motion with CSS3 transforms/transitions
- Look at this premade sidebar component: https://github.com/balloob/react-sidebar
- Consider something like this: https://daneden.github.io/animate.css/
- For page transitions, look into this: http://tympanus.net/Development/PageTransitions/

Troubleshooting

Image with null source

On 2/28/2017, I got an error saying "Failed to load resource: the server responded with a status of 404 (Not Found)". Chrome's network tab showed "http://localhost:3000/null" and an initiator of a random script. Commenting out that script showed another script as the initiator (and so on and so forth). It turns out I was trying to use an image with a null source, which I didn't realize until I looked at the network tab in Firefox (which correctly showed the source as "img" as opposed to some JavaScript).

Component isn't updating when it should

3/31/2017
The first thing to test is whether any of the React lifecycle methods are even being called. For example, add a "componentWillReceiveProps" function that just has a "debugger;" statement in it.

> If the lifecycle methods aren't being hit, then it means that nothing is even trying to render your component. When this happened to me, it turned out that a parent component's "shouldComponentUpdate" was preventing *its* render method from being called again, so the child was never hitting any of its lifecycle methods.
>
> If the lifecycle methods *are* being hit, then it means that you have bad logic in one of them or an early return or something. For example, maybe you want your component to update when the colors of something change, but your shouldComponentUpdate only checks to see if the ID changed.

setState isn't working

Don't call setState from your constructor, instead, just write "this.state = …whatever".