SB$_{C}omponents$

Release 1.0.0

SB$_{C}omponents$

Apr 16, 2018

Contents

CHAPTER 1

Project Setup

1.1 Development Environment

- install node, npm included, here
- install vscode here
- Checkout VsCode for extensions

1.2 Cloning and Branching

1. Clone the project `git clone https://github.com/osu-cass/sb-components`
2. Checkout dev `git checkout dev`
3. Create a feature branch `git checkout -b feat/{your-branch-here}`
4. Create a PR!

1.3 Setup

1. run `npm i` to install dependencies
2. run `npm run storybook` to start development server

1.4 Development

1.4.1 Storybook

Each component is developed independently of the DOM using storybook. You'll need to write a .storybook file to develop your component.

CHAPTER 2

Style Guide

2.1 General style

We follow AirBnb style guide listed [here] and Microsoft contribution library rules.(https://github.com/airbnb/javascript/blob/master/react/README.md) Follow basic A11y guidelines

2.2 Cheatsheet

- Directories pascalcase `DirectoryName`
- File names pascalcase 'FileName'
- File extension for react is `.tsx`
- File extension for non-react is `.ts`
- Styles names are lowercase 'filename.less'
- constants camelcase
- interface pascalcase
- interface members are camelcase
- No models except for props and state in tsx
- Component names **don't** have a suffix "component"
- Containers names **do** have suffix "Container"
- Models have suffix "Model"
- One react component per file unless stateless

CHAPTER 3

Visual Studio Code Environment

3.1 Development environment

- install node, npm included, here
- install vscode here

3.2 Extensions

3.2.1 Required

- Add Document This here
- TSLint here
- Spell Checker here
- Prettier - Code Formatter here
- Jest here

3.2.2 Optional

- TypeScript Hero here
- Typescript React code snippets here
- JavaScript (ES6) code snippets here
- Git History here
- Git Lens here
- Visual Studio Keymap here

CHAPTER 4

React Best Practices

A collection of best practice guidelines for ReactJS. Prepared by Rob Caldecott

4.1 Contents

4.2 JavaScript

A collection of JavaScript tips.

4.2.1 Stop using `var`

Use `const` and `let` instead. They have proper block scoping, unlike `vars` which are hoisted to the top of the function.

```
let name = "John Doe";
name = "Someone else";
...
const name = "John Doe";
// This will fail
name = "Someone else";
```

4.2.2 Use object shorthand notation

```
state = { name: "" };

onChangeName = e => {
  const name = e.target.value;
  this.setState({ name });
  // this.setState({ name: name });
};
```

4.2.3 Use string templates

```
const name = "John Doe";
const greeting = `Hello ${name}, how are you?`;
```

4.3 ESNext

You can take advantage of some next-generation JavaScript syntax right now, including:

- Async/await (ES7, ratified in June 2017)
  - Useful when using promises and `window.fetch`
- `Object` rest/spread (stage 3 proposal)
  - Destructure and make shallow copies of objects
- Class fields and `static` properties (stage 2 proposal)
  - Initialise component state
  - Auto-bound event handlers

4.3.1 async/await

Simplify your promise handling.

Before

```
const fetchIp = () => {
  window
    .fetch("https://api.ipify.org?format=json")
    .then(response => response.json())
    .then(({ ip }) => {
      // We're done, here in this handler
      this.setState({ ip });
    })
    .catch(({ message }) => {
      // Special catch handler syntax
      this.setState({ error: message })
    });
};
```

After

```
const fetchIp = async () => {
  try {
    const response = await window.fetch("https://api.ipify.org?format=json");
    const { ip } = await response.json();
    // We're done: the code looks synchronous
    this.setState({ ip });
  } catch ({ message }) {
    // Standard try/catch
    this.setState({ error: message });
  }
};
```

4.3.2 `Object` spread/rest

Use this to pull out useful properties from an object or make a shallow copy.

```
const { text, show } = this.props;
const { text, ...other } = this.props;
const copy = { ...data, additional: "value" };
```

4.3.3 Class fields and static properties

Make your React classes more readable:

```
class MyComponent extends React.Component {
  state = { name: "" };

  onChangeName = e => {
```

```
    this.setState({ name: e.target.value });
  }
}
```

4.3.4 Component class skeleton

A typical component class using ESNext syntax looks like this:

```
class MyComponent extends React.Component {
  static propTypes = { ... };

  static defaultProps = { ... };

  state = { ... };

  onEvent = () => { ... };

  classMethod() { ... }
}
```

4.4 prop-types

Always declare your props. Simply install the `prop-types` module from npm:

```
npm install --save prop-types
```

And then import the module into your component:

```
import PropTypes from "prop-types";
```

Using prop types ensures:

- Consumers of your component can see exactly what props are supported.
- Console warning are displayed when the wrong prop type is used.
- Props can be documented for use with `react-styleguidist`.

4.4.1 Example (class using ESNext static property support)

The following component supports two props: `onClick` which is a function and `name` which is a string and is a mandatory prop.

```
import React from "react";
import PropTypes from "prop-types";

export default class Greeting extends React.Component {
  static propTypes = {
    onClick: PropTypes.func,
    name: PropTypes.string.isRequired
  };

  render() {
```

```
    return (
      <h1 onClick={this.props.onClick}>
        Hello, {this.props.name}
      </h1>
    );
  }
}
```

4.4.2 Example (stateless functional component)

When using a stateless functional component you need to declare prop types on the function object itself:

```
import React from "react";
import PropTypes from "prop-types";

const Greeting = props =>
  <h1 onClick={props.onClick}>
    Hello,{props.name}
  </h1>;

Greeting.propTypes = {
  onClick: PropTypes.func,
  name: PropTypes.string.isRequired
};

export default Greeting;
```

4.4.3 Use destructuring to import specific prop types

You can also use destructuring to import just the prop types you need. This can save typing, especially when using props of the same type.

For example, here is the above stateless functional component example rewritten:

```
import React from "react";
import { func, string } from "prop-types";

const Greeting = props =>
  <h1 onClick={props.onClick}>
    Hello,{props.name}
  </h1>;

Greeting.propTypes = {
  onClick: func,
  name: string.isRequired
};

export default Greeting;
```

4.4.4 Using object shapes and arrays

You can also specify the *shape* of an object prop or the shape of arrays.

For example, the following component expects an array of objects. Each object requires `id` and `name` string properties.

```
import React from "react";
import { arrayOf, shape, string } from "prop-types";

const Stores = ({ stores }) =>
  <ul>
    {stores.map(store =>
      <li key={store.id}>
        {store.name}
      </li>
    )}
  </ul>;

Stores.propTypes = {
  stores: arrayOf(
    shape({
      id: string.isRequired,
      name: string.isRequired
    })
  )
};

export default Stores;
```

4.4.5 Custom prop types: sharing your shapes

You can easily share custom prop types by adding them to a file and exporting them for use in your project. For example:

```
// customProps.js
import { string, shape } from "prop-types";

export const store = shape({
  id: string.isRequired,
  name: string.isRequired
});

// Stores.js
import React from "react";
import { arrayOf } from "prop-types";
import { store } from "./customProps.js";

const Stores = ({ stores }) =>
  <ul>
    {stores.map(store =>
      <li key={store.id}>
        {store.name}
      </li>
    )}
  </ul>;

Stores.propTypes = {
  stores: arrayOf(stores).isRequired
};
```

```
export default Stores;
```

4.4.6 Specifying default prop values

You can also declare default values for props by declaring the `defaultProps` object on the component class or function. For example:

```
import React from "react";
import PropTypes from "prop-types";

export default class Heading extends React.Component {
  static propTypes = {
    backgroundColor: PropTypes.string,
    color: PropTypes.string,
    children: PropTypes.node.isRequired
  };

  static defaultProps = {
    backgroundColor: "black",
    color: "white"
  };

  render() {
    const style = {
      backgroundColor: this.props.backgroundColor,
      color: this.props.color
    };

    return (
      <h1 style={style}>
        {this.props.children}
      </h1>
    );
  }
}
```

This can be especially useful for `func` props as it stops a potential crash if an optional function prop is not supplied. For example:

```
import React from "react";
import { func } from "prop-types";

const Button = ({ onClick }) =>
  <div className="btn" onClick={onClick}>
    Button
  </div>;

Button.propTypes = {
  onClick: func
};

Button.defaultProps = {
  onClick: () => {}
};
```

```
export default Button;
```

Using default prop values in stateless functional components

Rather than declaring `defaultProps` for stateless functional components, you can use a combination of **destructuring** and **default parameter values** instead. For example:

```
import React from "react";
import { string, node } from "prop-types";

const Heading = ({ children, backgroundColor = "black", color = "white" }) => {
  const style = {
    backgroundColor,
    color
  };

  return (
    <h1 style={style}>
      {children}
    </h1>
  );
};

Heading.propTypes = {
  backgroundColor: string,
  color: string,
  children: node.isRequired
};

export default Heading;
```

4.5 Stateless Functional Components

Stateless functional components are React components as JavaScript functions. They can be used for components that do not use any lifecycle methods other than render and do not use any state.

- Concerned with *how things look*.
- AKA as *presentational* or *dumb* components.
- Functional programming paradigm: **stateless function components are pure functions of their props**.
- Props passed as the first function parameter.
- Simply return the component JSX: the same as the class `render` method.
- No state, no lifecycle methods.
- Easy to test.
- Easy to re-use/share.
- Make no assumptions about application state or the data source.
- Can be combined with container components (which may have state and may know about the data source).

4.5.1 Example

A simple greeting component: it displays a name and calls a prop when clicked.

> Note that ES6 arrow functions are preferred.

```
import React from "react";
import { func, string } from "prop-types";

const Greeting = ({ onClick, name }) =>
  <h1 onClick={onClick}>
    Hello, {name}
  </h1>;

Greeting.propTypes = {
  onClick: func,
  name: string.isRequired
};

export default Greeting;
```

4.5.2 Simple snapshot testing

You can quickly test a simple component like this using **snapshot testing**. For example:

```
import React from "react";
import renderer from "react-test-renderer";
import Greeting from "../Greeting";

it("renders", () => {
  expect(renderer.create(<Greeting name="The name" />)).toMatchSnapshot();
});
```

4.6 Containers

Containers are combinations of *state* and *presentational components*.

- Concerned with *how things work*.
- Usually ES6 class components with state.
- Render Re-usable stateless functional components.
- Knowledge about the data source and/or the application state.
- Commonly used with `react-redux`.
- Often generated using *higher order components* (HOCs).

4.6.1 Example

Here is a simple presentational component that renders a styled IP address:

```
import React from "react";
import { string } from "prop-types";

const IPAddress = ({ ip }) => {
  const styles = {
    container: {
      textAlign: "center"
    },
    ip: {
      fontSize: "20px",
      fontWeight: "bold"
    }
  };
  return (
    <div style={styles.container}>
      <div style={styles.ip}>
        {ip}
      </div>
    </div>
  );
};

IPAddress.propTypes = {
  ip: string
};

export default IPAddress;
```

And here is an example container for this component. The container knows about the data source (in this case how to fetch the current IP.)

Notice the following characteristics:

- It is an ES6 class.
- It has state.
- It is using component lifecycle (`componentDidMount`).
- It is acting as a wrapper for the `IPAddress` component.

```
import React from "react";
import IPAddress from "./IPAddress";

export default class IPAddressContainer extends React.Component {
  state = { ip: "" };

  componentDidMount() {
    window
      .fetch("https://api.ipify.org?format=json")
      .then(response => response.json())
      .then(response => {
        this.setState({ ip: response.ip });
      });
  }

  render() {
    return <IPAddress ip={this.state.ip} />;
  }
}
```

4.7 Higher Order Components

Higher Order Components (or HOCs) are used to transform a component into another component.

- A HOC **is a function that takes a component and returns a new component**.
- Made possible due to the compositional nature of React.
- Often used to inject additional props into an existing component.
- Useful for creating containers.
- A popular example is the `react-redux` `connect` function.
- Can be used to re-use code, hijack the `render` method and to manipulate existing props.

4.7.1 Example: IP address

The following HOC function will fetch the IP address and inject a prop called `ip` into **any** component. This is an example of using a `class` as the container.

```
import React from "react";

const withIPAddress = Component => {
  return class extends React.Component {
    state = { ip: "" };

    componentDidMount() {
      window
        .fetch("https://api.ipify.org?format=json")
        .then(response => response.json())
        .then(response => {
          this.setState({ ip: response.ip });
        });
    }

    render() {
      return <Component ip={this.state.ip} {...this.props} />;
    }
  };
};

export default withIPAddress;
```

Notice what's happening here: we are exporting a function that accepts a component as a parameter and returns an ES6 class.

To use this with an existing component we do something like this:

```
import React from "react";
import { string } from "prop-types";
import withIPAddress from "./withIPAddress";

const SimpleIPAddress = ({ ip, color = "black" }) =>
```

```
  <p style={{ color }}>
    {ip}
  </p>;

SimpleIPAddress.propTypes = {
  ip: string
};

export default withIPAddress(SimpleIPAddress);
```

We export the result of calling `withIPAddress`, passing in the component in which we want the `ip` prop injected.

4.7.2 Example: language

Here's another example of a HOC that injects the current browser language setting into any component as a prop called `language`. In this case we are using a stateless functional component as the container.

```
// withLanguage.js
import React from 'react';

const withLanguage = Component => props =>
  <Component {...props} language={navigator.language} />;

export default withLanguage;

// MyComponent.js
import React from "react";
import { string } from "prop-types";
import withLanguage from "./withLanguage";

const MyComponent = ({ language }) =>
  <div>
    Browser language: {language}
  </div>;

MyComponent.propTypes = {
  language: string
};

export default withLanguage(MyComponent);
```

4.8 Chaining HOCs

Note that you can also chain HOCs together to create a new component that combines them all. For example:

```
import React from "react";
import { string } from "prop-types";
import withLanguage from "./withLanguage";
import withIPAddress from "./withIPAddress";

const MyComponent = ({ language, ip }) =>
  <div>
    <div>
      Browser language: {language}
```

```
    </div>
    <div>
      IP address: {ip}
    </div>
  </div>;

MyComponent.propTypes = {
  language: string,
  ip: string
};

export default withLanguage(withIPAddress(MyComponent));
```

4.9 Functions as Children

An alternative pattern to HOCs is **functions as children** where you supply a function to call as the child of a container component: this is the equivalent of a **render callback**. Like HOCs you are decoupling your parent and child and it usually follows a similar pattern of a parent that has state you want to hide from the child.

This has some advantages over traditional HOCs:

- It does not pollute the `props` namespace. HOCs have an implicit contract they impose on the inner components which can cause prop name collisions especially when combining them with other HOCs.
- You do not need to use a function to create the container: you use simple composition instead.
- Developers do not need to call an HOC function to create a new wrapped component which can simplify the code: they simply export their child components as normal.

In order for this to work you need to use a function as the special `children` prop and have the outer container component call this function when rendering.

For example, here is a component that exposes the browser language:

```
import React from 'react';
import { func } from 'prop-types';

const Language = ({ children }) =>
  <div>
    {children(navigator.language)}
  </div>;

Language.propTypes = {
  children: func,
};

export default Language;
```

The component simply treats the `children` prop as a function and calls it. It can be used like this:

```
<Language>
  {language =>
    <p>
      Browser language: {language}
    </p>}
</Language>
```

The child node of <Language> is a function which returns the JSX to render.

Now let's imagine a component that calls an API and uses state to store the status, response and error.

```
import React from 'react';
import { string, func } from 'prop-types';

export default class CallAPI extends React.Component {
  static propTypes = {
    api: string,
    children: func,
  };

  state = {
    isFetching: false,
    data: {},
    error: '',
  };

  async componentDidMount() {
    this.setState({ isFetching: true });
    try {
      const response = await fetch(this.props.api);
      const data = await response.json();
      this.setState({ isFetching: false, data });
    } catch ({ message }) {
      this.setState({ isFetching: false, error: message });
    }
  }

  render() {
    return (
      <div>
        {this.props.children({ ...this.state })}
      </div>
    );
  }
}
```

The component makes an API call (specified with a prop) and maintains the state of the call. It renders by calling a function and passing through a copy of the state as an object.

It could be used like this:

```
<CallAPI api="https://api.ipify.org?format=json">
  {({ isFetching, data, error }) => {
    if (isFetching) {
      return <p>Loading...</p>;
    }
    if (error) {
      return <p>Error: {error}</p>;
    }
    return <p>Data: {JSON.stringify(data)}</p>;
  }}
</CallAPI>
```

And of course you can render normal components in the callback, for example:

```
<CallAPI api="https://api.ipify.org?format=json">
{
  props => <MyComponent {...props} />
}
</CallAPI>
```

> There is a caveat to using this pattern: they cannot be optimised by React because **they change on every render** (a new function is declared on every render cycle). This rules out using `shouldComponentUpdate` and `React.PureComponent` which may lead to performance issues. Use this pattern wisely.

4.10 Events

When using JavaScript DOM and `window` events we usually need `this` to point to our component instance.

Spot the bug in this code:

```
import React from "react";

export default class BindBug extends React.Component {
  state = { toggled: false };

  onClick(e) {
    this.setState({ toggled: !this.state.toggled });
  }

  render() {
    const style = {
      fontSize: "36px",
      color: this.state.toggled ? "white" : "black",
      backgroundColor: this.state.toggled ? "red" : "yellow"
    };

    return (
      <div style={style} onClick={this.onClick}>
        Click me
      </div>
    );
  }
}
```

When you click on the `<div>` the `onClick` function handler is called which then tries to call `this.setState`. But the handler has not bound `this` to the component instance and it ends up as `null` which causes the code to crash.

4.10.1 Binding events

To make this work we need to bind the `onClick` function to `this`. There are two ways to do this:

ESNext property initialize syntax (recommended)

The most readable way to do this is via an ESNext property initializer in conjunction with an arrow function. Arrow functions declared in this way are bound to `this` automatically:

```
import React from "react";

export default class BindClassMethod extends React.Component {
  state = { toggled: false };

  onClick = e => {
    this.setState({ toggled: !this.state.toggled });
  };

  render() {
    const style = {
      fontSize: "36px",
      color: this.state.toggled ? "white" : "black",
      backgroundColor: this.state.toggled ? "red" : "yellow"
    };

    return (
      <div style={style} onClick={this.onClick}>
        Click me
      </div>
    );
  }
}
```

Notice the syntax used here:

```
handlerName = (params) => { ... }
```

This is the best option: it is less code and even though this syntax is experimental it is used widely at Facebook.

Here's another example: a component that uses `window.setInterval` to update a counter every second:

```
import React from "react";

export default class Timer extends React.Component {
  state = { counter: 0 };

  componentDidMount() {
    this.timerId = window.setInterval(this.onTimer, 1000);
  }

  componentWillUnmount() {
    window.clearInterval(this.timerId);
  }

  onTimer = () => {
    this.setState(prevState => ({ counter: prevState.counter + 1 }));
  };

  render() {
    return (
      <p>
        Counter: {this.state.counter}
      </p>
```

```
    );
  }
}
```

Note the following:

- The timer ID is stored so it can be cleared when the component unmounts.
- The function version of `setState` is used.

Alternatively you could use an inline arrow function: this will ensure `this` has the correct context:

```
import React from "react";

export default class Timer extends React.Component {
  state = { counter: 0 };

  componentDidMount() {
    this.timerId = window.setInterval(() => {
      this.setState(prevState => ({ counter: prevState.counter + 1 }));
    }, 1000);
  }

  componentWillUnmount() {
    window.clearInterval(this.timerId);
  }

  render() {
    return (
      <p>
        Counter: {this.state.counter}
      </p>
    );
  }
}
```

Constructor binding

Another common way of binding is to add a constructor to your class and use `Function.prototype.bind`:

```
import React from "react";

export default class BindConstructor extends React.Component {
  state = { toggled: false };

  constructor() {
    super();
    this.onClick = this.onClick.bind(this);
  }

  onClick(e) {
    this.setState({ toggled: !this.state.toggled });
  }

  render() {
    const style = {
      fontSize: "36px",
      color: this.state.toggled ? "white" : "black",
```

```
      backgroundColor: this.state.toggled ? "red" : "yellow"
    };

    return (
      <div style={style} onClick={this.onClick}>
        Click me
      </div>
    );
  }
}
```

Although this method is not relying on any experimental syntax it suffers from the following issues:

- It requires you adding a constructor.
- You have to remember call `super` in the constructor before doing anything else.
- It is more code.

4.10.2 Sharing event handlers

Sometimes it is useful to share the share event handlers for your components and there is a simple trick to do this using the DOM `name` attribute (which is exposed as a prop for most React components):

```
import React from "react";

export default class DetailsForm extends React.Component {
  state = {
    name: "",
    email: "",
    phone: ""
  }

  onChange = e => {
    this.setState({
      [e.target.name]: e.target.value
    });
  }

  render() {
    return (
      <div>
        <input name="name" value={this.state.name} onChange={this.onChange} />
        <input name="email" value={this.state.email} onChange={this.onChange} />
        <input name="phone" value={this.state.phone} onChange={this.onChange} />
      </div>
    )
  }
}
```

This takes advantage of the JavaScript **computed property name syntax** to update the state. Notice how the `name` prop for each `<input>` matches the corresponding state property: this allows us to share a single `onChange` handler with all three components.

> Although this looks like magic **it's just JavaScript**.

4.10.3 Handling the ENTER key in a form

If you want to let users press the ENTER key to submit a form then you will need to prevent the default `submit` behaviour of a HTML form. For example:

```
import React from "react";

export default class LoginForm extends React.Component {
  onSubmit = e => {
    // Don't actually submit!
    e.preventDefault();
    // Enter key was pressed
  };

  render() {
    return (
      <form onSubmit={this.onSubmit}>
        <input
          name="username"
          value={this.state.value}
          onChange={this.onChange}
        />
        <input
          name="password"
          type="password"
          value={this.state.password}
          onChange={this.onChange}
        />
        <input type="submit" value="Login" />
      </form>
    );
  }
}
```

This looks pretty much like a standard HTML form: the presence of the `<input type="submit" />` ensures the ENTER key works but by calling `preventDefault` on the submit event you can handle it yourself without the application reloading.

4.11 Conditional Rendering

Sometimes it is useful to render components based on your props or state and there are at least five different mechanisms available to you (remember: it's just JavaScript!)

> Note that when you conditionally remove a component it **will be re-mounted when you put it back** which means `componentDidMount` and other lifecycle methods will be called again. So if you are, for example, fetching data when the component mounts, it will be called each time. To avoid this use some form of `show` prop and either return `null` from your `render` or use CSS to hide the content.

4.11.1 Store the JSX in a variable

You can declare a variable to hold the JSX you wish to render. If your condition is not met and an `undefined` variable is rendered, then React will simply ignore it.

```
let message;
if (someCondition) {
  message = <p>Hello, world!</p>;
}

return (
  <div>
    <p>Conditional rendering</p>
    {message}
  </div>
)
```

4.11.2 Ternaries

You can also use a **ternary**. Using `null` or `undefined` is enough to stop anything being rendered:

```
return (
  <div>
    <p>Conditional rendering</p>
    {someCondition ? <p>Hello, world!</p> : null}
  </div>
)
```

4.11.3 Logical && operator shortcut

This relies on the fact the JavaScript will stop evaluating an && condition if the preceding checks return `false`.

```
return (
  <div>
    <p>Conditional rendering</p>
    {someCondition && <p>Hello, world!</p>}
  </div>
)
```

So if `someCondition` is `true` then your JSX is rendered, but if it's `false` then your JSX will simply not be evaluated.

This is a very common method to conditionally render something in React.

4.11.4 Return null from your render method

Another common pattern seen in some 3rd-party component libraries is to conditionally render a component based on a boolean prop. For example, you may have a prop called `show` that determines if the component should display at all: if not then your `render` method can simply return `null`.

> The advantage of this is that the component will not be mounted multiple times each time the `show` prop changes which is useful if you are fetching data, setting timers, etc. in `componentDidMount`.

```
// MyComponent.js
const MyComponent = ({ show }) => {
  if (show) {
    return <p>Hello, world!</p>;
  }
  return null;
```

```
};

// SomeOtherComponent.js
...
return (
  <div>
    <p>Conditional rendering</p>
    <MyComponent show={someCondition} />
  </div>
)
```

4.11.5 Hide your component using CSS

A final way is to simply use CSS to hide your component. This also has the advantage of keeping your component mounted.

```
// MyComponent.js
const MyComponent = ({ show }) => {
  const style = {
    display: show ? "block" : "none"
  };

  return <p style={style}>Hello, world!</p>;
};

// SomeOtherComponent.js
...
return (
  <div>
    <p>Conditional rendering</p>
    <MyComponent show={someCondition} />
  </div>
)
```

4.12 Arrays

When dealing with arrays of JavaScript objects you can use `Array.prototype.map` to map from array elements to React components. This is a very common pattern.

The following example shows a component that is rendering an array of stores by mapping each array entry to a new `<li>` component.

```
import React from "react";
import { arrayOf, shape, number, string } from "prop-types";

const StoreList = ({ stores }) =>
  <ul>
    {stores.map(store =>
      <li key={store.id}>
        {store.name}
      </li>
    )}
  </ul>;
```

```
StoreList.propTypes = {
  stores: arrayOf(
    shape({
      id: number.isRequired,
      name: string.isRequired
    }).isRequired
  )
};

export default StoreList;
```

4.12.1 Keys

Keys help React identify which items have changed, are added, or are removed. Keys should be given to the elements inside the array to give the elements a stable identity:

```
<li key={store.id}>{store.name}</li>
```

Avoid using array indexes if array items can reorder.

```
stores.map((store, index) => <li key={index}>{store.name}</li>
```

4.13 Writing Components

For this section we will use an example of a simple button component but the technique is the same no matter what sort of component you are developing.

4.13.1 Designing a Button component

At first our button is very simple:

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

const Button = ({ text }) =>
  <button className="btn">
    {text}
  </button>;
```

4.13.2 More requirements

Now we need support to render an icon:

```
<Button text="Click me!" iconName="paper-plane-o" />

const Button = ({ text, iconName }) =>
  <button className="btn">
    <i className={"fa fa-" + iconName} />
    {" " + text}
  </button>;
```

4.13.3 Even more requirements!

Now the button needs text formatting, icon positioning and icon size support. **The code is getting complicated**.

```
<Button text="Click me" textStyle="bold" iconName="paper-plane-o" iconPosition="top"␣
↪iconSize="2x" />

const Button = props => {
  const icon =
    props.iconName &&
    <i
      className={classnames("fa fa-" + props.iconName, {
        ["fa-" + props.iconSize]: props.iconSize
      })}
    />;

  return (
    <button className="btn">
      {icon &&
        (props.iconPosition === "top"
          ? <div>
              {icon}
            </div>
          : <span>
              {icon + " "}
            </span>)}
      {props.textStyle === "bold"
        ? <strong>
            {props.text}
          </strong>
        : <span>
            {props.text}
          </span>}
    </button>
  );
};
```

It is clear we cannot continue designing the component in this way for long before it becomes unmanageable.

4.13.4 Composition to the rescue

Instead of using lots of props and a single complicated `render` method split the component into smaller chunks and use composition to render it instead.

```
<Button>
  <FontAwesome
    name="paper-plane-o"
    size="2x"
    block />
  <strong>Click me</strong>
</Button>

const Button = ({ children }) =>
  <button className="btn">
    {children}
  </button>;
```

```
const FontAwesome = ({ name, size, block }) =>
  <i
    className={classnames("fa", "fa-" + name, {
      ["fa-" + size]: size,
      ["center-block"]: block
    })}
  />;
```

This takes advantage of the special `children` prop which is the cornerstone of composition using React.

4.13.5 Summary

You might need composition when:

- There are too many props
- There are props to target a specific part of the component (iconName, iconPosition, iconSize, etc.)
- There are props which are directly copied into the inner markup
- There are props which take a complex model object or an array

4.14 Unit Testing

Testing should be simple!

- React components are easy to test.
- Presentational components (stateless functional components) should be treated as pure functions.
- Two common testing patterns are:
  - DOM testing
    - Find DOM nodes, simulate events
  - Snapshot testing
    - Compares files of JSON output
    - Show the diff

4.14.1 Snapshot testing

Snapshot testing is a feature of **Jest** that can be used to test *any* JavaScript object. And thanks to a package called `react-test-renderer` you can convert a React component to an object to use with snapshot testing.

For example:

```
import React from "react";
import renderer from "react-test-renderer";
import IPAddress from "../IPAddress";

it("renders", () => {
  const component = renderer.create(<IPAddress ip="127.0.0.1" />);
  expect(component).toMatchSnapshot();
});
```

When the test runs for the first time a special snapshot file is created in a sub-folder containing the JSON output of the render. The next time you run the test it generates new output and compares it with the snapshot: if there are any differences then the test has failed and you are presented with the object diff. At this point you can decide to regenerate the snapshot.

4.14.2 DOM testing

Alternatively you can render your components into a in-memory DOM (`jsdom`).

- Use `react-dom/test-utils` or `enzyme`
- Find DOM nodes and check attributes
- Simulate events
- Mock event handlers

  Note that this does not work with stateless functional components unless you wrap them with a class (you can use a simple HOC for this.)

For example:

```
import React from "react";
import Greeting from "../Greeting";
import ReactTestUtils from "react-dom/test-utils";

it("renders", () => {
  const onClick = jest.fn();
  const instance = ReactTestUtils.renderIntoDocument(
    <Greeting name="The name" onClick={onClick} />
  );
  const h1 = ReactTestUtils.findRenderedDOMComponentWithTag(instance, "h1");
  ReactTestUtils.Simulate.click(h1);
  expect(onClick).toHaveBeenCalled();
});
```

In this example we use `ReactTestUtils` to render the component, look for the `<h1>` tag and simulate a click event. We then check that our `onClick` prop was called.

Wrapping stateless functional components for `ReactTestUtils`

`ReactTestUtils` does not play well with stateless functional components. To fix this simply wrap your component with a class when testing. You can use an HOC for this in your project:

```
const withClass = Component => {
  return class extends React.Component {
    render() {
      return <Component {...this.props} />;
    }
  };
};

const Component = withClass(Greeting);
const instance = ReactTestUtils.renderIntoDocument(
  <Component name="The name" onClick={onClick} />
);
```

4.15 State

4.15.1 State updates may be asynchronous!

React may batch multiple `setState()` calls into a single update for performance.

Because `this.props` and `this.state` may be updated asynchronously, you should not rely on their values for calculating the next state.

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment
});
```

Instead, you can use the **function version of `setState`**.

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

Always use this version of `setState` if you need access to the previous state or props.

State update functions can be extracted and tested

Another benefit of using the function version of `setState` is you can extract them from your class, turn them into **thunks** and add tests for them. If you stick to using immutable data for your state then the update functions should be pure which makes them even easier to test. You can even share state update functions amongst your components.

For example:

```
// Stores.js
export const addStore = (id, name) => prevState => ({
  stores: [...prevState.stores, { id, name }]
});

export default class Stores extends React.Component {
  state = { stores: [] };

  onAddStore = () => {
    this.setState(addStore("ID", "NEW STORE NAME"));
  }
  ...
}

// Stores.test.js
import { addStore } from "./Stores";

it("adds a store to the state", () => {
  const prevState = {
    stores: [
      {
        id: "1",
        name: "Store 1"
      },
      {
```

```
        id: "2",
        name: "Store 2"
      }
    ]
  }
  expect(addStore("3", "Store 3")(prevState)).toMatchSnapshot();
});
```

4.15.2 Immutable data

- React tends to favour functional programming paradigms
- Mutable data can often be a source of bugs and unintended side effects
- Using immutable data can simplify testing
- Redux relies on immutable state to work correctly
- You don't necessarily need ImmutableJS: ES6 will usually suffice
- Immutable data can be used alongside `React.PureComponent` for a very simple performance boost

Immutable arrays

Here is an example of using `Array.prototype.map` to clone an array and modify a single element:

```
this.setState(prevState => ({
  items: prevState.items.map(item => {
    if (item.id === idToFind) {
      return { ...item, toggled: !item.toggled };
    }
    return item;
  })
}));
```

You can make a shallow copy of an array and add a new element at the same time using the **array spread operator**:

```
this.setState(prevState => ({
  items: [...prevState.items, { id: "3", name: "New store" }]
}));
```

You can remove elements from an array using `Array.prototype.slice`:

```
this.setState(prevState => ({
  // Remove the first element
  items: prevState.items.slice(0, 1)
}));
```

4.16 Props

4.16.1 Destructuring

You can increase code readability be destructuring props. For example:

```
render() {
  const { name, email } = this.props;

  return (
    <div>
      <p>{name}</p>
      <p>{email}</p>
    </div>
  )
}
```

4.16.2 Don't pass on unknown props

If you are wrapping components with another do not pass down any props that the wrapped component does not know about. This will generate a console warning in the browser. For example, this is wrong:

```
const Input = props => {
  const type = props.isNumeric ? "number" : "text";
  // <input> does not know about isNumeric
  // This will generate a console warning
  return <input {...props} type={type} />;
};

Input.propTypes = {
  isNumeric: PropTypes.bool
};
```

To fix this you can use the **object spread operator** to extract the props you care about and add the remaining ones to a single variable. For example:

```
const Input = props => {
  const { isNumeric, ...other } = props;
  const type = isNumeric ? "number" : "text";
  return <input {...other} type={type} />;
};

Input.propTypes = {
  isNumeric: PropTypes.bool
};
```

4.17 Pure Components

> Premature optimization is the root of all evil.

Most of the time you are probably not going to worry about performance but there are times when you might to avoid potentially costly `renders` and this is where `React.PureComponent` can help.

When React needs to reconcile the virtual DOM it will call your component `render` method and compare it with an in-memory copy. If anything has changed then the real DOM is updated. Usually this is fast but if your `render` function is slow (perhaps it renders many components) then there could be a delay while reconciliation takes place.

However, there is a React lifecycle method you can override called `shouldComponentUpdate` and if you return `false` from this then your `render` method **will not be called**.

To make this easier to manage you can derive your component class from `React.PureComponent` which overrides `shouldComponentUpdate` and performs a simple (and fast) value comparison of your props and state: if there are no changes then the function returns `false` and no render will occur.

So if your `render` method renders exactly the same result given the same props and state then you can use `React.PureComponent` for a potential performance boost.

> React performs a *value* comparison of your props and state and **not** a deep object comparison. Therefore you should use immutable data for all props and state to ensure this comparison works as expected: otherwise your component may not render when you expect it to.

For example:

```
import React from "react";
import PropTypes from "prop-types";

export default class MyList extends React.PureComponent {
  static propTypes = {
    items: PropTypes.arrayOf(
      PropTypes.shape({
        id: PropTypes.number,
        text: PropTypes.text
      })
    )
  };

  render() {
    // Only called if the props have changed
    return (
      <List>
        {this.props.items.map(item =>
          <ListItem key={item.id} primaryText={item.text} />
        )}
      </List>
    );
  }
}
```

If the `items` prop changes (is replaced with a new copy of the data) then the component will render.

4.18 Project Structure

There are numerous ways to structure your React project. One common layout for components:

- Components are located in `src/components/ComponentName.js`.
- Component-specific CSS is located in `src/components/ComponentName.css`.
- Component tests are located in `src/components/__tests__/ComponentName.test.js`.
- Component stories are located in `src/components/__stories__/ComponentName.stories.js`
- React Styleguidist component examples (if applicable) are located in `src/components/__examples__/ComponentName.md`

If you're using `redux`:

- Code to initialise your store is located in `src/store.js`

- Reducers are located in `src/reducers`
- Action creators are located in `src/actions`
- Selectors are located in `src/selectors`
- Action constants are located in `src/constants/actions.js`

Try and limit the number of files in the root `src` folder but be careful not to overdo your folder structure. There is nothing wrong with lots of files in one folder (Facebook use a monorepo: they have over 30,000 components in a single folder!)

An example layout may look like this:

```
src\
  index.js
  App.js
  setupTests.js
  components\
    __tests__\
      Button.test.js
    __stories__\
      Button.stories.js
    Button.js
    Button.css
  containers\
    __tests__\
      MainPage.test.js
    MainPage.js
  utils\
    __tests__\
      sharedStuff.test.js
    sharedStuff.js
    testUtils.js
```

Another layout involves a separate folder with each component containing the source code, CSS, tests, stories and any other component-specific files. For this to be manageable you need to also add an `index.js` that imports the component and this is not recommended for beginners.

4.19 Summary

- It's just JavaScript.
- Use functional programming patterns and techniques where possible.
- Use containers/presentational components.
- Always declare your prop types.
- Take advantage of ES6 and ESNext.
- Use immutable data.
- Use snapshot testing.
- Use the function form of `setState` if you need access to the previous state or props.
- Favour small components and composition when building your UI.
- Don't ignore console warnings.

CHAPTER 5

Using SB-Components

dev guide for installing and running sb-components

5.1 Install peer dependencies

- react
- react-dom
- bootstrap
- jquery
- react-modal
- font-awesome
- typeface-pt-sans-caption
- typeface-pt-serif
- type-pt-serif-caption
- @sbac/SBAC-Global-UI-Kit

5.2 Install sb-components using npm:

```
npm install --save @osu-cass/sb-components
```

5.3 Typings

Included in lib

5.4 Required Assets

Copies images from sbac global to project public directory Use webpack copy

```
npm install --save-dev copy-webpack-plugin
```

Webpack config:

```
new CopyWebpackPlugin([
    {
        from: path.join(__dirname, 'node_modules', '@sbac/sbac-ui-kit/src/images/'),
        to: path.join(__dirname, 'public', 'Assets/Images')
    }
])
```

5.5 Required Less

In order to use the sbac global style sheet, use a less file to bundle peer dependencies bootstrap, font-awesome, sbac-ui, and this project

- See example in Assets/Styles/bundle.less
- Sbac-ui uses default bootstrap constants and can be overwritten during this step

Create a less file

```
//** bundle.less
//## Peers
@import "~bootstrap/less/bootstrap.less";
@import "~font-awesome/less/font-awesome.less";
@import "~@sbac/sbac-ui-kit/src/less/sbac-ui-kit.less";
@import "~@osu-cass/sb-components/lib/Assets/Styles/sb-components.less";

//## Custom Styles
{your styles here}
```

CHAPTER 6

Accessibility

6.1 Minimum Requirements

- Use a keyboard to interact with components (tab, space, enter, shift-tab)
- Icons should have aria-hidden
- Links or buttons with no text need to have aria-label or labeled-by. More info here
- Roles should be used unless input (input type does this automatically). More info here
- Check contrast of components being used (storybook accessibility tab) or color-contrast-tool
- Storybook accessibility tab should have no issues

6.2 Resources

- A11y here
- React Accessibility here
- MDN button and links here
- MDN Accessibility here
- Font awesome here
- MDN Aria roles here
- Tables here

6.3 TSLint

- Linting is enabled for this project to determine if elements are compliant

  - Note this does not check for everything. Manual checks required

- Enable vscode tslint for the best experience when developing

CHAPTER 7

Using NPM Link

This is the preferred way to try out changes in dependent projects

7.1 Setup

1. Run `npm link` within this project
2. Run `npm link @osu-cass/sb-components`

7.1.1 AP_ItemSampler

Steps for AP_ItemSampler

1. Follow npm link steps
2. Run `npm run watch` within this project
3. Any changes will be watched
4. ItemSampler, has hot-loading and will reload from this project automatically

CHAPTER 8

Publish, Branching and Versioning

8.1 Versioning

Semantic versioning

All versions in npm should also be github releases using tags. Tag names are the version number.

- Official builds use major.minor.patch
  - example: `v1.1.0`
- Pre-Release builds use major.minor.patch-alpha.x
  - example: `v1.2.1-alpha.3`

8.2 Branching

- `master`, (default branch) contains current production code. Official builds and tags
- `hotfix`, immediate bug fixes, pre-release versions and tags. (hotfix => master)
  - example: `hotfix/1.2.1`
- `release`, pre-release versions and tags (release => master)
  - example: `release/1.3.0`
- `dev`, current dev code, (dev => release)
- `feature`, short-lived new feature code, (feature => dev)
  - example: `feature/acc-modal-style`

8.3 Publish Npm Package

builds to lib directory and used in the npm package

`webpack.config.js` has a dev, prod, and a watch command. Running webpack tasks outputs to the `lib` directory. There is a custom `tsconfig.webpack.json` to only include npm needed files and includes `src` directory.

8.3.1 Publishing

1. Update `package.json` with the correct version {major.minor.patch}
2. Run `npm install` and commit `package.json` and `package.lock.json`
3. Run `npm run prepare` to verify the package before pushing.
4. Run `npm publish`, you may need to setup login creds
5. Tag version `git tag -a v{version}`
6. Push commit and push tags `git push` and `git push --tags`

8.3.2 Local Dev

Develop with npm link instead of pushing versions

- Follow NpmLink

8.4 Continuous Integration, TravisCI

Travis will check code quality, conflicts, build, and tests. TODO: Linting support and NPM Publish

- Pull Request, determines if merge will succeed.
- Tags
  - TODO: soon to come, npm publish
- Branch build, runs for each commit
  - `master` will do a publish to github pages

8.5 Pull Requests

All changes to master and dev should be a Pull Request. Master and Dev are protected to prevent accidental pushes. PR's should be code reviewed (1 person required) and pass checks (travis ci, codacy, coveralls...) before accepting a pull request.

CHAPTER 9

Site Styles

This project utilizes bootstrap and the sbac-global style overrides for buttons, typography, form-controls, and custom button group. However, don't use the bootstrap grid system instead use flex when you can.

9.1 Less

All project specific less files are included in Assets/Styles

- `Custom.less` includes any overrides to the imported bootstrap and/or sbac-global. Includes shared custom less for the project that can be shared.
- `Layout.less` includes custom layout classes such as container and section
- `Constants.less` includes all static constants to be shared (sbac constants can be used)

9.2 SBAC Global style

- SBAC styleguide
- `variables.less` github
- `colors.less` github

9.3 Useful Links

- SBAC styleguide
- Flex cheatsheet
- More flex css-tricks

9.4 Layout

All pages must have a container class. Content needs to align with the navbar content. Page app must include a page container.

9.4.1 With Page Title

```
<div className="container {page}-container">
    <h2 className="page-title">Page Title</h2>
    <div className="section section-light" style={style}>
      <p>Test Body...</p>
    </div>
</div>
```

9.4.2 With No Page Title

```
<div className="container {page}-container">
    <div className="section section-light" style={style}>
      <p>Test Body...</p>
    </div>
</div>
```

9.4.3 With Full Body Styles

Helpful for Sample Items Website full page background

```
<div className="page-container {page}-page">
    <div className="container {page}-container">
        <div className="container section-light" style={style}>
        <p>Test Body...</p>
        </div>
    </div>
</div>
```

CHAPTER 10

How to Commit

```
# basic structure
type(optional scope): a message

# an example
feat: allow users to message each other directly

Users no longer have to communicate through public channels; user accounts are now␣
↪treated as their own channels, so you can communicate with individuals directly.

Fixes #31

# also valid
chore(dependencies): add webpack and relevant loaders
```

Types are pre-defined, enforced values, while scopes are optional. You can checkout the list of types with their descriptions for more information. commitlint will tell you the available options when you get them wrong, but it's up to you to know which one is the most relevant.

CHAPTER 11

Roadmap

11.1 1.1.0

- SBAC-ui-kit support
- Toggle between cards and item table
- Style improvements
- 80% code coverage
- TSLint check on travis builds

11.2 1.0

- Scoreguide and Sample Items Website components functional
- Code coverage
- Storybook
- Webpack library
- Typings!

CHAPTER 12

Project Overview

12.1 Structure

- src
  - feature directories
    - tests, snapshots and functional tests
    - ts and tsx files
  - Assets
    - Styles
- mocks
  - feature directories
    - mock data ts
- stories
  - snapshots
  - features directories
    - tests (component ui tests)
- typings
  - custom typings
- lib (build output)
  - feature directories
    - typings
  - index.js (contains sb-component code)
  - index.d.ts (contains list of typings)

- gh-site
  - assets for github.io
  - storybuild (build output)
  - typedocs (build output)

12.1.1 src

Includes all feature code to be shared with projects. Code is separated out in feature directories. Please see styleguide. Jest is used for functional testing including snapshots. All mock data for testing needs to be placed in project root `mocks`.

12.1.2 mocks

Includes all mock data for `src` and `stories`.

12.1.3 stories

Uses storybook component development kit. Run `npm run storybook` and launch `http://localhost:6006`. Each story will create a snapshot testing. All mock data should be placed in `mocks`. Storybook supports hot module reloading (HMR).

12.1.4 typings

Custom typings for the project.

12.1.5 lib

includes build output for ts/tsx code. `lib` only includes js, styles, and typings for external use.

12.1.6 gh-site

Root directory for the github.io pages. `tsdoc` and `storybook` build output. `tsdoc` includes all the project code level documentation auto generated from jsdoc format.

CHAPTER 13

Indices and tables

- genindex
- modindex
- search