Javascript

We use the Airbnb Javascript Style Guide as reference for Javascript style.

React

For all React technologies, we have our own styleguide also heavily influenced by Airbnb React Style Guide.

Table of Contents

Basic Rules

- Only include one React component per file.
- Always use JSX syntax.
- Do not use `React.createElement` unless you're initializing the app from a file that is not JSX.
- Always use export default for Components.
- Use `export default` for reducers, actionCreators and services. As a general rule of thumb, use `export default` for all files that have a unique object to export.

Folder Structure

```
src
|
└───app
|   |
|   └───components
```

```
│     │  └───baseComponents
│     │        └───Input
│     │        └───Text
│     │        └───Button
│     │        └───etc
│     └───screens
│           └───MyScreenComponent
│                 └───assets // Screen specific app assets
│                 | components
│                 | constants.js
│                 | i18n.js
│                 | index.js
│                 | layout.js
│                 | styles.scss
│                 | utils.js
│
└───assets // General app assets
└───config
      | api.js
      | i18n.js
└───constants
└───redux
│     | store.js
│     └───myReducer
│           | actions.js
│           | reducer.js
│           | selectors.js
│
└───propTypes
│     | Model1.js
│     | Model2.js
│
└───scss
└───services
      | MyService.js
│
└───utils
│    index.js
```

Class vs `React.createClass` vs stateless

- If you have internal state and/or refs, prefer `class extends Component` over `React.createClass`. eslint: `react/prefer-es6-class` `react/prefer-stateless-function`

```
// bad
const Listing = React.createClass({
  // ...
  render() {
    return <div>{this.state.hello}</div>;
  }
});

// good
class Listing extends Component {
  // ...
  render() {
    return <div>{this.state.hello}</div>;
  }
}
```

And if you don't have state or refs, only for stateless components, prefer normal functions (not arrow functions) over classes:

```
// bad
class Listing extends Component {
  render() {
    return <div>{this.props.hello}</div>;
  }
}

// bad (relying on function name inference is discouraged)
const Listing = ({ hello }) => (
  <div>{hello}</div>
);

// good
function Listing({ hello }) {
  return <div>{hello}</div>;
}
```

- Avoid using helper render methods when possible. Functions that return JSX elements should probably be layout components.

```
// bad
function TextContainer extends Component {
  renderText = text => <span>text</span>;

  render() {
    return (
      <div>
        {this.renderText('aText')}
      </div>
    )
  }
}

// good
function Text({ text }) {
  return <span>text</span>;
}

function TextContainer({ text }) {
  return (
    <div>
      <Text text={text} />
    </div>
  )
}
```

Mixins

- Do not use mixins.

> Why? Mixins introduce implicit dependencies, cause name clashes, and cause snowballing complexity. Most use cases for mixins can be accomplished in better ways via components, higher-order components, or utility modules.

Naming

- **Extensions**: Use `.js` extension for React components.
- **Filename**: For component filenames and services use PascalCase. E.g., `ReservationCard.js`.
- **Reference Naming**: Use PascalCase for React components and camelCase for their associated elements. eslint: `react/jsx-pascal-case`

```
// bad
import reservationCard from './ReservationCard';

// good
import ReservationCard from './ReservationCard';

// bad
const ReservationItem = <ReservationCard />;

// good
const reservationItem = <ReservationCard />;
```

- **Component Hierarchy**:
  - Component files should be inside folders that match the component's name.
  - Use index.js as the filename of a container component. Use `Container` as the suffix of the component's name.
  - Use layout.js as the filename of a layout component.

```
// MyComponent/index.js
import MyComponent from './layout'
```

```
class MyComponentContainer extends Component {
  // Do smart stuff

  render() {
    return <MyComponent />
  }
}

// MyComponent/layout.js
function MyComponent() {
  return (
    // Some JSX
  )
}
```

- **Higher-order Component Naming**: Use a composite of the higher-order component's name and the passed-in component's name as the `displayName` on the generated component. For example, the higher-order component `withFoo()`, when passed a component `Bar` should produce a component with a `displayName` of `withFoo(Bar)`.

  > Why? A component's `displayName` may be used by developer tools or in error messages, and having a value that clearly expresses this relationship helps people understand what is happening.

```
// bad
function withFoo(WrappedComponent) {
  return function WithFoo(props) {
    return <WrappedComponent {...props} foo />;
  }
}

export default withFoo;

// good
function withFoo(WrappedComponent) {
  function WithFoo(props) {
    return <WrappedComponent {...props} foo />;
  }

  const wrappedComponentName = WrappedComponent.displayName
    || WrappedComponent.name
    || 'Component';

  WithFoo.displayName = `withFoo(${wrappedComponentName})`;
  return WithFoo;
}

export default withFoo;
```

- **Props Naming**: Avoid using DOM component prop names for different purposes.

  > Why? People expect props like `style` and `className` to mean one specific thing. Varying this API for a subset of your app makes the code less readable and less maintainable, and may cause bugs.

  ```
  // bad
  <MyComponent style="fancy" />

  // bad
  <MyComponent className="fancy" />

  // good
  <MyComponent variant="fancy" />
  ```

Import

- Prefer wrapping the variables inside a file in an object and then import that object instead of using `import *`

  ```
  // bad
  /* routes.js */
  const userListRoute = '/users';
  const itemListRoute = '/items';

  /* Another file */
  import * as Routes from './routes';

  // good
  /* routes.js */
  const Routes = {
    userListRoute: '/users',
    itemListRoute: '/items'
  }

  export default Routes;

  /* Another file */
  import Routes from './routes';
  ```

Declaration

- Do not use `displayName` for naming components. Instead, name the component by reference.

  ```
  // bad
  export default React.createClass({
  ```

```
  displayName: 'ReservationCard',
  // stuff goes here
});

// good
class ReservationCard extends Component {
}

export default ReservationCard
```

Alignment

- Follow these alignment styles for JSX syntax. eslint: `react/jsx-closing-bracket-location` `react/jsx-closing-tag-location`

```
// bad
<Foo superLongParam="bar"
     anotherSuperLongParam="baz" />

// good
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
/>

// if props fit in one line then keep it on the same line
<Foo bar="bar" />

// children get indented normally
<Foo
  superLongParam="bar"
  anotherSuperLongParam="baz"
>
  <Quux />
</Foo>
```

Quotes

- Always use double quotes (`"`) for JSX attributes, but single quotes (`'`) for all other JS. eslint: `jsx-quotes`

> Why? Regular HTML attributes also typically use double quotes instead of single, so JSX attributes mirror this convention.

```
// bad
<Foo bar='bar' />

// good
```

```jsx
<Foo bar="bar" />

// bad
<Foo style={{ left: "20px" }} />

// good
<Foo style={{ left: '20px' }} />
```

Spacing

- Always include a single space in your self-closing tag. eslint: `no-multi-spaces`, `react/jsx-tag-spacing`

  ```jsx
  // bad
  <Foo/>

  // very bad
  <Foo                 />

  // bad
  <Foo
   />

  // good
  <Foo />
  ```

- Do not pad JSX curly braces with spaces. eslint: `react/jsx-curly-spacing`

  ```jsx
  // bad
  <Foo bar={ baz } />

  // good
  <Foo bar={baz} />
  ```

Props

- Always use camelCase for prop names.

  ```jsx
  // bad
  <Foo
    UserName="hello"
    phone_number={12345678}
  />

  // good
  <Foo
    userName="hello"
  ```

```
  phoneNumber={12345678}
/>
```

- Always use object destructuring to explicitly get props variables in the render function of class Components:

```
import MyComponent from './layout';

// bad
class MyComponentContainer extends Component {
  render() {
    return <MyComponent foo={this.props.foo} bar={this.props.bar} />
  }
}

// good
class MyComponentContainer extends Component {
  render() {
    const { foo, bar } = this.props;
    return <MyComponent foo={foo} bar={bar} />
  }
}
```

- Always use object destructuring to explicitly get props variables in layout Components:

```
// bad
function MyComponent(props) {
  return <span>{this.props.foo}</span>
}

// good
function MyComponent({ foo }) {
  return <span>{foo}</span>
}
```

- Omit the value of the prop when it is explicitly `true`. eslint: `react/jsx-boolean-value`

```
// bad
<Foo
  hidden={true}
/>

// good
<Foo
  hidden
/>
```

```
// good
<Foo hidden />
```

- Explicitly `true` props should be passed last:

```
// bad
<Foo
  hidden
  disabled
  aProp={aPropValue}
/>

// good
<Foo
  aPropValue={aPropValue}
  hidden
  disabled
/>
```

- Avoid passing arrow functions in props when possible. Instead, create a reference to the function and pass that reference.

  > Why? Passing arrow functions as props in render creates a new function each time the component renders, which is less performant.

```
import MyComponent from './layout';

// bad
class MyComponentContainer extends Component {
  render() {
    return <MyComponent foo={bar => bar + 1} />
  }
}

// good
class MyComponentContainer extends Component {
  foo = bar => bar + 1;

  render() {
    return <MyComponent foo={this.foo} />
  }
}
```

- Always include an `alt` prop on `<img>` tags. If the image is presentational, `alt` can be an empty string or the `<img>` must have `role="presentation"`. eslint: `jsx-a11y/alt-text`

```
// bad
<img src="hello.jpg" />

// good
<img src="hello.jpg" alt="Me waving hello" />

// good
<img src="hello.jpg" alt="" />

// good
<img src="hello.jpg" role="presentation" />
```

- Do not use words like "image", "photo", or "picture" in `<img>` `alt` props. eslint: `jsx-a11y/img-redundant-alt`

  > Why? Screenreaders already announce `img` elements as images, so there is no need to include this information in the alt text.

  ```
  // bad
  <img src="hello.jpg" alt="Picture of me waving hello" />

  // good
  <img src="hello.jpg" alt="Me waving hello" />
  ```

- Use only valid, non-abstract ARIA roles. eslint: `jsx-a11y/aria-role`

  ```
  // bad - not an ARIA role
  <div role="datepicker" />

  // bad - abstract ARIA role
  <div role="range" />

  // good
  <div role="button" />
  ```

- Do not use `accessKey` on elements. eslint: `jsx-a11y/no-access-key`

> Why? Inconsistencies between keyboard shortcuts and keyboard commands used by people using screenreaders and keyboards complicate accessibility.

```
// bad
<div accessKey="h" />

// good
<div />
```

- Avoid using an array index as `key` prop when possible, prefer a unique ID. (why?)

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

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

- Always define explicit defaultProps for all undefined props.

> Why? propTypes are a form of documentation, and providing defaultProps means the reader of your code doesn't have to assume as much. In addition, it can mean that your code can omit certain type checks.

```jsx
// bad
function SFC({ foo, bar, children }) {
  return <div>{foo}{bar}{children}</div>;
}
SFC.propTypes = {
  foo: PropTypes.number.isRequired,
  bar: PropTypes.string,
  children: PropTypes.node,
};

// good
function SFC({ foo, bar, children }) {
  return <div>{foo}{bar}{children}</div>;
}
SFC.propTypes = {
  foo: PropTypes.number.isRequired,
  bar: PropTypes.string,
  children: PropTypes.node,
};
SFC.defaultProps = {
  bar: '',
  children: null,
};
```

- Avoid spreading props.

> Why? Otherwise you're more likely to pass unnecessary props down to components. And for React v15.6.1 and older, you could pass invalid HTML attributes to the DOM.

Exceptions:

- HOCs that proxy down props and hoist propTypes

```
function HOC(WrappedComponent) {
  return class Proxy extends Component {
    Proxy.propTypes = {
      text: PropTypes.string,
      isLoading: PropTypes.bool
    };

    render() {
      return <WrappedComponent {...this.props} />
    }
  }
}
```

- Base smart components like InputContainer, ButtonContainer, etc. if it's clear that they will want to pass all its props to it's layout component:

```
import Button from './layout';
class ButtonContainer extends Component {
  // do something smart

  render() {
    return <Button {...this.props} />
  }
}
```

- Spreading objects with known, explicit props. This can be particularly useful when testing React components with Mocha's beforeEach construct.

```
function Foo {
  const props = {
    text: '',
    isPublished: false
  }

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

export default Foo;
```

Notes for use: Filter out unnecessary props when possible. Also, use prop-types-exact to help prevent bugs.

```
// good
render() {
  const { irrelevantProp, ...relevantProps  } = this.props;
  return <WrappedComponent {...relevantProps} />
}

// bad
render() {
  const { irrelevantProp, ...relevantProps  } = this.props;
  return <WrappedComponent {...this.props} />
}
```

Refs

- Always use ref callbacks. eslint: `react/no-string-refs`

  > Note: Only use refs for class components. Reasons in React docs: https://reactjs.org/docs/refs-and-the-dom.html#refs-and-functional-components

  ```
  // bad
  <Foo
    ref="myRef"
  />

  // good
  setRef = ref => this.myRef = ref;

  <Foo
    ref={this.setRef}
  />
  ```

Parentheses

- Wrap JSX tags in parentheses when they span more than one line. eslint: `react/jsx-wrap-multilines`

  ```
  // bad
  render() {
    return <MyComponent variant="long body" foo="bar">
             <MyChild />
           </MyComponent>;
  }

  // good
  render() {
    return (
      <MyComponent variant="long body" foo="bar">
  ```

```
      <MyChild />
    </MyComponent>
  );
}

// good, when single line
render() {
  const body = <div>hello</div>;
  return <MyComponent>{body}</MyComponent>;
}
```

Tags

- Always self-close tags that have no children. eslint: `react/self-closing-comp`

```
// bad
<Foo variant="stuff"></Foo>

// good
<Foo variant="stuff" />
```

- If your component has multi-line properties, close its tag on a new line. eslint: `react/jsx-closing-bracket-location`

```
// bad
<Foo
  bar="bar"
  baz="baz" />

// good
<Foo
  bar="bar"
  baz="baz"
/>
```

Methods

- Implement all methods in classes as arrow functions, except for lifecycle methods.

> Why? Lifecycle methods should be class functions, not instance functions.

```
// bad
class MyComponent extends Component {
  componentDidMount = () => {
    // do something
  }
  foo() {
    // do something else
  }
}

// good
class MyComponent extends Component {
  componentDidMount() {
    // do something
  }

  foo = () => {
    // do something else
  }
}
```

- Avoid using `bind` in class Components. You can instead use arrow functions. If for some reason you must do so, bind event handlers for the render method in the constructor. eslint: `react/jsx-no-bind`

> Why? A bind call in the render path creates a brand new function on every single render.

```
// bad
class extends Component {
  onClickDiv() {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv.bind(this)} />;
  }
}

// not that bad
class extends Component {
  constructor(props) {
    super(props);

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

  onClickDiv() {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv} />;
  }
}

// good
```

```
class extends Component {
  // No binding needed. onClickDiv has reference to `this` because
it's an arrow function.

  onClickDiv = () => {
    // do stuff
  }

  render() {
    return <div onClick={this.onClickDiv} />;
  }
}

}
```

- Do not use underscore prefix for internal methods of a React component.

> Why? Underscore prefixes are sometimes used as a convention in other languages to denote privacy. But, unlike those languages, there is no native support for privacy in JavaScript, everything is public. Regardless of your intentions, adding underscore prefixes to your properties does not actually make them private, and any property (underscore-prefixed or not) should be treated as being public. See issues #1024, and #490 for a more in-depth discussion.

```
// bad
React.createClass({
  _onClickSubmit() {
    // do stuff
  },

  // other stuff
});

// good
class extends Component {
  onClickSubmit() {
    // do stuff
  }

  // other stuff
}
```

- Be sure to return a value in your `render` methods. eslint: `react/require-render-return`

```
// bad
render() {
  (<div />);
}

// good
```

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

Ordering

- Ordering for `class extends Component`:

1. optional `static` methods
2. `constructor`
3. `getChildContext`
4. `componentWillMount`
5. `componentDidMount`
6. `componentWillReceiveProps`
7. `shouldComponentUpdate`
8. `componentWillUpdate`
9. `componentDidUpdate`
10. `componentWillUnmount`
11. `componentDidCatch`
12. *clickHandlers or eventHandlers* like `onClickSubmit()` or `onChangeDescription()`
13. *getter methods for `render`* like `getSelectReason()` or `getFooterContent()`
14. *optional render methods* like `renderNavigation()` or `renderProfilePicture()`
15. `render`
16. prop types
17. default props
18. mapStateToProps (if using Redux)
19. mapDispatchToProps (if using Redux)
20. export default MyComponent

- How to define `propTypes`, `defaultProps`, `contextTypes`, etc...

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

class Link extends Component {
  static methodsAreOk() {
    return true;
  }

  render() {
    const { url, id, text } = this.props;
    return <a href={url} data-id={id}>{text}</a>;
  }
}

Link.propTypes = {
  id: PropTypes.number.isRequired,
  url: PropTypes.string.isRequired,
  text: PropTypes.string,
};

Link.defaultProps = {
  text: 'Hello World',
};

export default Link;
```

`isMounted`

- Do not use `isMounted`. eslint: `react/no-is-mounted`

> Why? `isMounted` is an anti-pattern, is not available when using ES6 classes, and is on its way to being officially deprecated.

HOCs

If the HOC you are creating requires some sort of configuration or extra parameters, don't pass them with the wrapped component. Make a function that receives the configuration and returns a HOC. You can then pass the wrapped component to that HOC.

```
//good HOC without configuration
function withSomethingExtra(WrappedComponent){ }
//good HOC with configuration
function withSomethingExtra(config1, config2)(WrappedComponent){ }
```

```
//bad HOC
function withSomethingExtra(WrappedComponent, config1, config2){ }
function withSomethingExtra({component: WrappedComponent, config1,
config2}){ }
```

This way, we know exactly how to use each HOC and we can compose them with other libraries like `recompose`'s `compose`

```
const composedHoc = compose(hoc1(config1), hoc2, hoc3(config3));
const WrappedComponent = composedHoc(Component);
```

Translation

This JSX/React style guide is also available in other languages:

- 🇨🇳 **Chinese (Simplified)**: JasonBoy/javascript
- 🇹🇼 **Chinese (Traditional)**: jigsawye/javascript

- 🇪🇸 **Español**: agrcrobles/javascript
- 🇯🇵 **Japanese**: mitsuruog/javascript-style-guide
- 🇰🇷 **Korean**: apple77y/javascript
- 🇵🇱 **Polish**: pietraszekl/javascript
- 🇧🇷 **Portuguese**: ronal2do/javascript
- 🇷🇺 **Russian**: leonidlebedev/javascript-airbnb
- 🇹🇭 **Thai**: lvarayut/javascript-style-guide
- 🇹🇷 **Turkish**: alioguzhan/react-style-guide
- 🇺🇦 **Ukrainian**: ivanzusko/javascript

⬆ back to top