Skip to end of metadata

- Created by Sandeep Khandewale, last modified on Jan 31, 2020
-
-
-
-

Go to start of metadata

Based on Airbnb React/JSX Coding Guidelines

Basic Rules

- Only include one React component per file.
- However, multiple Stateless, or Pure, Components are allowed per file. eslint: `react/no-multi-comp`.
- Always use JSX syntax.
- Do not use `React.createElement` unless you're initializing the app from a file that is not JSX.

Class vs `React.createClass` vs stateless

- If you have internal state and/or refs, prefer `class extends React.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 React.Component {
 // ...
  render() {
 return <div>{this.state.hello}</div>;
  }
}
```

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

```
// bad
class Listing extends React.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>;
}
```

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 `.jsx` extension for React components. eslint: `react/jsx-filename-extension`
- Filename: Use PascalCase for filenames. E.g., `ReservationCard.jsx`.
- Reference Naming: Use PascalCase for React components and camelCase for their instances. eslint: `react/jsx-pascal-case`

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

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

// bad
const ReservationItem = <ReservationCard />;

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

- Component Naming: Use the filename as the component name. For example, `ReservationCard.jsx` should have a reference name of `ReservationCard`. However, for root components of a directory, use `index.jsx` as the filename and use the directory name as the component name:

```
// bad
import Footer from './Footer/Footer';

// bad
import Footer from './Footer/index';

// good
import Footer from './Footer';
```

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

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

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

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

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 export default class ReservationCard extends
React.Component { }
```

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>
• 
•   // bad
•   {showButton &&
•     <Button />
•   }
• 
•   // bad
•   {
•    showButton &&
•       <Button />
•   }
• 
•   // good
•   {showButton && (
•     <Button />
•   )}
• 
•   // good
  {showButton && <Button />}
```

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
<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`
- `// bad`
- `<Foo/>`
-
- `// very bad`
- `<Foo                 />`
-
- `// bad`
- `<Foo`
- ` />`
-
- `// good`

`<Foo />`
- Do not pad JSX curly braces with spaces. eslint: `react/jsx-curly-spacing`
- `// bad`
- `<Foo bar={ baz } />`
-
- `// good`

`<Foo bar={baz} />`

Props

- Always use camelCase for prop names.

`// bad <Foo UserName="hello" phone_number={12345678} /> // good <Foo userName="hello" phoneNumber={12345678} />`
- 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 />`
- 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, prefer a stable ID. eslint: `react/no-array-index-key`

Why? Not using a stable ID [is an anti-pattern]() because it can negatively impact performance and cause issues with component state.
We don’t recommend using indexes for keys if the order of items may change.

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

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

- Always define explicit defaultProps for all non-required 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.

```
// 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,
};
```

- Use spread props sparingly.

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 React.Component {
    Proxy.propTypes = {
 text: PropTypes.string,
 isLoading: PropTypes.bool
    };

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

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

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

  return (<div {...props} />);
}
```

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

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

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

Refs

- Always use ref callbacks. eslint: `react/no-string-refs`
- `// bad`
- `<Foo`
- `  ref="myRef"`
- `/>`
-
- `// good`
- `<Foo`
- `  ref={(ref) => { this.myRef = ref; }}`
`/>`

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

- Use arrow functions to close over local variables.

```
function ItemList(props) { return ( <ul> {props.items.map((item, index) => (
<Item key={item.key} onClick={() => doSomethingWith(item.name, index)} />
))} </ul> ); }
```

- 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 React.Component {
  onClickDiv() {
 // do stuff
  }

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

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

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

  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 React.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 React.Component`:

1. optional `static` methods
2. `constructor`
3. `getChildContext`
4. `componentWillMount`
5. `componentDidMount`
6. `componentWillReceiveProps`
7. `shouldComponentUpdate`
8. `componentWillUpdate`
9. `componentDidUpdate`
10. `componentWillUnmount`
11. *clickHandlers or eventHandlers* like `onClickSubmit()` or `onChangeDescription()`
12. *getter methods for `render`* like `getSelectReason()` or `getFooterContent()`
13. *optional render methods* like `renderNavigation()` or `renderProfilePicture()`
14. `render`

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

```
import React from 'react'; import PropTypes from 'prop-types'; const
propTypes = {  id: PropTypes.number.isRequired,  url:
PropTypes.string.isRequired,  text: PropTypes.string, }; const defaultProps =
{  text: 'Hello World', }; class Link extends React.Component {  static
methodsAreOk() {  return true;  } render() {  return <a href={this.props.url}
data-id={this.props.id}>{this.props.text}</a>;  } } Link.propTypes =
propTypes; Link.defaultProps = defaultProps; export default Link;
```

- Ordering for `React.createClass`: eslint: `react/sort-comp`

1. `displayName`
2. `propTypes`
3. `contextTypes`
4. `childContextTypes`
5. `mixins`
6. `statics`
7. `defaultProps`
8. `getDefaultProps`
9. `getInitialState`
10. `getChildContext`
11. `componentWillMount`
12. `componentDidMount`
13. `componentWillReceiveProps`
14. `shouldComponentUpdate`
15. `componentWillUpdate`
16. `componentDidUpdate`
17. `componentWillUnmount`
18. *clickHandlers or eventHandlers* like `onClickSubmit()` or `onChangeDescription()`
19. *getter methods for `render`* like `getSelectReason()` or `getFooterContent()`

20. *optional render methods* like `renderNavigation()` or `renderProfilePicture()`
21. `render`

`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.

Security / OWASP related coding guidelines

- Use JSX and avoid React.createElement like APIs - since JSX has sanitization and escaping in built
- Avoid dangerouslySetInnerHTML. In case you have to use it - avoid setting values that are input from user / service. Use sanitization and escaping for any values being set for this field
- maxLength for free text fields is necessary, especially for those fields whose text value indulges in http operations
- For any AJAX call, make sure it is using secure communication channels.
- For any AJAX response - developer should be cautious about consuming the fields that are part of response in UI,
- e.g. it is necessary to escape and sanitize it before assigning it as value to UI text field / label.
- e.g. while displaying error message - we should not take entire error message from response body - since it may contain error details that backend missed to omit. UI developer should refer to error code and use error messages available to UI
- Access Control Logic - The part of code that is not executable due to current user not being authorized, should not be delievered in script bundle at all. It should be conditionally loaded
- It is not practically feasible in every small UI component that should be available for certain condition for user. So impact analysis should be made in such case and such treatment should be done only for medium / high impact areas.
- This is not really front end area entirely - but our code should store cookies wisely (http only cookies) so that these cookies are not accessible to code.