ReactJS Style Guide

> *Mosaic's code style guide for ReactJS applications.*

Timo Zimmermann
Mosaic S.A.R.L
mosaic.mc

Table of Contents

Variables

Use `const` whenever possible. If a variable needs to be reassigned a different value, use `let`. In all cases, `var` must be avoided. When declaring multiple variables, group them as shown below and divide `const` and `let` variables by an empty line.

Examples:

```
// Do
const a = {};
const b = [];
const c = '';

let d = '';
let e = '';

// Don't
const a = {};
let e = '';
const b = [];
const c = '';
let d = '';
```

Naming Convention

Always use camel case to name your variables. Function names should contain verbs. Examples for good function names are `convertToHtml`, `download` or `fetchUserById`. Names for variables should be as short but as precise as possible. Good examples are `user` or `projects`. Booleans should be prefixed with `is` or `has`. For example `isOwner` or `hasRootAccess`. The file name of a React component should start with a capital letter.

Quotes

Use single quotes `''` for strings. If necessary, you can also use template strings.

Indentation

Use four spaces for indentation.

Line Length

Use a line length of 140 characters.

Functions

Use anonymous arrow functions whereever possible. If you need access `this`, use named function expressions. Always wrap the function body in curly brackets.

Examples:

```
// Do
const func = () => {
    // ...
};
```

```
function func() {
    // ...
};

// Don't
const func = () =>
    // ...

const func = function() {
    // ...
};
```

Parameters

Awlays put optional parameters at the end.

Examples:

```
// Do
const func = (a, b = {}) => {};

// Don't
const func = (b = {}, a) => {};
```

Imports

Group import statements by the following order separated by an empty line.

- Libraries
- React components
    - If you import plenty of components from the same directory, for example you import ten components from the `UI` directory and five from the `layout` directory, group them together and divide them by an empty line.
- Services
- Configs
- Utils
- Assets and media
- Stylesheets

If a file contains JSX, the first line should contain the React import statement. This might not always be necessary, but it immediately shows whether the file contains a React component.

Exports

Use default exports whenever possible. If you are not exporting a React component, export an object. Assign the object to a variable before exporting it. This allows you to add new methods to a service in the future without having to change any import statements.

Examples:

```
// Do
const utils = {
```

```
    clipboard,
    time
};

export default utils;

// Don't
export default {
    clipboard,
    time
};
```

When importing services, do not deconstruct them to keep the context of the respective method you are executing.

Examples:

```
// Do
import utils from 'utils'

await utils.time.sleep();

// Don't
import { time } from 'utils';

await time.sleep();
```

Component Structure

A React components should be structured as followed:

- Global state
- Local State
- Refs
- Functions
- useCallbacks
- useEffects
- Dynamic variables and useMemos
- JSX

Examples:

```
import React, { useState, useRef, useCallback, useEffect } from 'react';

import useStore from 'store';

const Component = () => {
    // Global state
    const { state } = useStore();

    // Local state
    const [isLoading, setIsLoading] = useState(false);

    // Refs
```

```
    const chartRef = useRef({});

    /**
     * Toggles the add user modal.
     */
    const toggleModal = () => {};

    /**
     * Fetches the current user and saves their
     * data to the local state.
     */
    const fetchUser = useCallback(async () => {});

    /**
     * Registers a window resize event listener
     * when the component mounts and removes it
     * again when it unmounts.
     */
    useEffect(() => {}, []);

    // Determine whether the current user is admin
    const isAdmin = user.permissions?.find(p => p.name === 'admin') || false;

    return (
        <div>...</div>
    );
};

export default Component;
```

CSS Modules

When using CSS Modules, name the file similar to the file that contains the React component the CSS modules belongs to.

Examples:

```
// Do
Dashboard.js
Dashboard.module.scss

// Don't
Dashboard.js
style.module.scss
```

CSS classes should be named with short precise names. Since in most cases the classes are scoped to a single React component you should avoid methologies such as BEM but use generic class names such as `.container` or `.list` as they are not overwriting the styles of other components. If you run out of possible names and have to start using class names such as `.leftList` and `.rightList`, you should probably split up your code into multiple components.

CSS Properties Order

Keeping CSS properties ordered uniformly, helps to quickly find the desired properties. Empty lines between properties must be avoided. Keep the common properties in the following order.

- content
- position
- top
- right
- bottom
- left
- width
- height
- display
- flex-direction
- justify-content
- align-items
- margin
- padding
- border-radius
- border
- background-color
- box-shadow
- transition
- opacity
- animation
- font-size
- font-weight
- color
- outline
- z-index

Conditional Rendering

Keep the return statement as simple as possible. Move complex logic out of the return statement.

Examples:

```
// Do
return isLoading ? <Spinner/> : <List/>;

// Don't
return isLoading ?
    <Spinner/>
    : (
        <ul>
            <li>a</li>
            <li>b</li>
            <li>c</li>
        </ul>
    );
```

If Statements

Complex conditions can become hard to read quickly. Keep them as clear as possible even if the code becomes more verbose. Avoid nested and multi line ternary operators. If conditions become to complex, assign them to separate variables first. Wrapping the body of an if statement is mandatory in all cases.

Examples:

```
// Do
const hasA = !!c.find(x => x.isAdmin);
const hasB = d > e;

const isF = (hasA && hasB) ? 'x' : 'y';

if (hasA && (start < end)) {
    // ...
}

// Don't
const isF = (!!c.find(x => x.isAdmin) && d > e) ? 'x' : 'y';

const isF = !!c.find(x => x.isAdmin) &&
                d > e ?
                'x' :
                'y';

if (hasA && hasB) return 'x';

if (hasA && hasB)
    return 'x';
```

Fallback Values

Make sure to safely access or deconstruct properties and apply the respective fallback values.

```
// Do
const firstName = data?.users?.find(u => u.id === id)?.firstName || '';

const dataPoints = (rawData || []).map(d => d.dataPoints || {});

// Don't
const firstName = data?.users?.find(u => u.id === id)?.firstName || false;

const dataPoints = res.data[0].rawData.map(d => d.dataPoints);
```

Exceptions

If you write a function in which an exception could occur, wrap the function body in a try-catch block. Whenever possible wrap the entire function body in a try block for better readability. The error passed to the catch method must be called `error`. Don't throw an error inside the catch statement if for instance a condition is not met. Handle the scenario gracefully.

Examples:

```
// Do
const fetchUser = async () => {
    try {
        const user = users.find(user => user.id === id);

        if (!user) {
            notifications.alert('Could not find user.');
            return;
        }

        const res = await api.getUserByEmail(user.email);

        setUserData(res || {});

    } catch(error) {
        // handle error
    }
};

// Avoid if possible
const fetchUser = async () => {
    const user = users.find(user => user.id === id);

    if (!user) {
        notifications.alert('Could not find user.');
        return;
    }

    try {
        const res = await api.getUserByEmail(user.email);

        setUserData(res || {});

    } catch(error) {
        // handle error
    }
};

// Don't
try {
    if (a < b) {
        throw new Error('A is not smaller');
    }

} catch(e) {
    // Handle error
}
```