Airbnb JS Style Guide

Full Stack Web Development

Naming Conventions

Comments

Whitespace

Commas

Semicolons

References

Variables

Arrays

Strings

Objects

Properties

Destructuring

Blocks

Control Statements

Functions

Arrow Functions

Modules

Blocks

Control Statements

Functions

Arrow Functions

Modules

Blocks

Blocks

- Use braces with all multiline blocks. eslint: **nonblock-statement-body-position**

```
// bad
if (test)
  return false;

// good
if (test) return false;

// good
if (test) {
  return false;
}

// bad
function foo() { return false; }

// good
function bar() {
  return false;
}
```

Blocks

- If you're using multiline blocks with `if` and `else`, put `else` on the same line as your `if` block's closing brace. eslint: **brace-style**

```
// bad
if (test) {
  thing1();
  thing2();
}
else {
  thing3();
}

// good
if (test) {
  thing1();
  thing2();
} else {
  thing3();
}
```

Blocks

- If an `if` block always executes a `return` statement, the subsequent `else` block is unnecessary. A `return` in an `else if` block following an `if` block that contains a `return` can be separated into multiple `if` blocks. eslint: `no-else-return`

```
// bad
function foo() {
  if (x) {
    return x;
  } else {
    return y;
  }
}

// bad
function cats() {
  if (x) {
    return x;
  } else if (y) {
    return y;
  }
}

// bad
function dogs() {
  if (x) {
    return x;
  } else {
    if (y) {
      return y;
    }
  }
}
```

```
// good
function foo() {
  if (x) {
    return x;
  }

  return y;
}

// good
function cats() {
  if (x) {
    return x;
  }

  if (y) {
    return y;
  }
}

// good
function dogs(x) {
  if (x) {
    if (z) {
      return y;
    }
  } else {
    return z;
  }
}
```

Control Statements

Control Statements

- In case your control statement (`if` , `while` etc.) gets too long or exceeds the maximum line length, each (grouped) condition could be put into a new line. The logical operator should begin the line.

> *"Why? Requiring operators at the beginning of the line keeps the operators aligned and follows a pattern similar to method chaining. This also improves readability by making it easier to visually follow complex logic."*

```
// bad
if ((foo === 123 || bar === "abc") && doesItLookGoodWhenItBecomesThatLong() && isThisReallyHappening()) {
  thing1();
}

// bad
if (foo === 123 &&
  bar === "abc") {
  thing1();
}

// bad
if (foo === 123
  && bar === "abc") {
  thing1();
}

// bad
if (
  foo === 123 &&
  bar === "abc"
) {
  thing1();
}
```

```
// good
if (
  foo === 123
  && bar === "abc"
) {
  thing1();
}

// good
if (
  (foo === 123 || bar === "abc")
  && doesItLookGoodWhenItBecomesThatLong()
  && isThisReallyHappening()
) {
  thing1();
}

// good
if (foo === 123 && bar === "abc") {
  thing1();
}
```

Control Statements

Don't use selection operators in place of control statements.

```
// bad
!isRunning && startRunning();

// good
if (!isRunning) {
  startRunning();
}
```

Functions

Functions

- Spacing in a function signature. eslint: **space-before-function-paren** **space-before-blocks**

> *“Why? Consistency is good, and you shouldn’t have to add or remove a space when adding or removing a name.”*

```
// bad
const f = function(){};
const g = function (){};
const h = function() {};

// good
const x = function () {};
const y = function a() {};
```

Functions

- Functions with multiline signatures, or invocations, should be indented just like every other multiline list in this guide: with each item on a line by itself, with a trailing comma on the last item. eslint: **`function-paren-newline`**

```
// bad
function foo(bar,
             baz,
             quux) {
  // ...
}

// good
function foo(
  bar,
  baz,
  quux,
) {
  // ...
}

// bad
console.log(foo,
  bar,
  baz);

// good
console.log(
  foo,
  bar,
  baz,
);
```

Functions

- Never reassign parameters. eslint: **no-param-reassign**

> *"Why? Reassigning parameters can lead to unexpected behavior, especially when accessing the* **arguments** *object. It can also cause optimization issues, especially in V8."*

```
// bad
function f1(a) {
  a = 1;
  // ...
}

function f2(a) {
  if (!a) { a = 1; }
  // ...
}

// good
function f3(a) {
  const b = a || 1;
  // ...
}

function f4(a = 1) {
  // ...
}
```

Functions

- Never mutate parameters. eslint: **no-param-reassign**

> *"Why? Manipulating objects passed in as parameters can cause unwanted variable side effects in the original caller."*

```
// bad
function f1(obj) {
  obj.key = 1;
}

// good
function f2(obj) {
  const key = Object.prototype.hasOwnProperty.call(obj, "key") ? obj.key : 1;
}
```

Functions

- Use default parameter syntax rather than mutating function arguments.

```
// really bad
function handleThings(opts) {
  // No! We shouldn't mutate function arguments.
  // Double bad: if opts is falsy it'll be set to an object which may
  // be what you want but it can introduce subtle bugs.
  opts = opts || {};
  // ...
}

// still bad
function handleThings(opts) {
  if (opts === void 0) {
    opts = {};
  }
  // ...
}

// good
function handleThings(opts = {}) {
  // ...
}
```

Functions

- Always put default parameters last. eslint: **default-param-last**

```
// bad
function handleThings(opts = {}, name) {
  // ...
}

// good
function handleThings(name, opts = {}) {
  // ...
}
```

Functions

- Wrap immediately invoked function expressions in parentheses. eslint: **wrap-iife**

> *"Why? An immediately invoked function expression is a single unit - wrapping both it, and its invocation parens, in parens, cleanly expresses this. Note that in a world with modules everywhere, you almost never need an IIFE."*

```
// immediately-invoked function expression (IIFE)
(function () {
  console.log("Welcome to the Internet. Please follow me.");
}());
```

Arrow Functions

Arrow Functions

- When you must use an anonymous function (as when passing an inline callback), use arrow function notation. eslint: **`prefer-arrow-callback`** , **`arrow-spacing`**

> *"Why? It creates a version of the function that executes in the context of* **`this`** *, which is usually what you want, and is a more concise syntax.*
>
> *Why not? If you have a fairly complicated function, you might move that logic out into its own named function expression."*

```
// bad
[1, 2, 3].map(function (x) {
  const y = x + 1;
  return x * y;
});

// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});
```

Arrow Functions

- Always include parentheses around arguments for clarity and consistency. eslint: **`arrow-parens`**

> *"Why? Minimizes diff churn when adding or removing arguments."*

```
// bad
[1, 2, 3].map(x => x * x);

// good
[1, 2, 3].map((x) => x * x);

// bad
[1, 2, 3].map(number => (
  `A long string with the ${number}. It’s so long that we don’t want it to take up space on the .map line!`
));

// good
[1, 2, 3].map((number) => (
  `A long string with the ${number}. It’s so long that we don’t want it to take up space on the .map line!`
));

// bad
[1, 2, 3].map(x => {
  const y = x + 1;
  return x * y;
});

// good
[1, 2, 3].map((x) => {
  const y = x + 1;
  return x * y;
});
```

Arrow Functions

- If the function body consists of a single statement returning an expression without side effects, omit the braces and use the implicit return. Otherwise, keep the braces and use a `return` statement. eslint: `arrow-parens` , `arrow-body-style`

> *"Why? Syntactic sugar. It reads well when multiple functions are chained together."*

```
// bad
[1, 2, 3].map((number) => {
  const nextNumber = number + 1;
  `A string containing the ${nextNumber}.`;
});

// good
[1, 2, 3].map((number) => `A string containing the ${number + 1}.`);

// good
[1, 2, 3].map((number) => {
  const nextNumber = number + 1;
  return `A string containing the ${nextNumber}.`;
});

// good
[1, 2, 3].map((number, index) => ({
  [index]: number,
}));
```

Modules

Modules

- Always use modules (**`import`** / **`export`**) over a non-standard module system. You can always transpile to your preferred module system.

> *"Why? Modules are the future, let's start using the future now."*

```
// bad
const AirbnbStyleGuide = require("./AirbnbStyleGuide");
module.exports = AirbnbStyleGuide.es6;

// ok
import AirbnbStyleGuide from "./AirbnbStyleGuide";
export default AirbnbStyleGuide.es6;

// best
import { es6 } from "./AirbnbStyleGuide";
export default es6;
```

Modules

- Do not use wildcard imports.

> *“Why? This makes sure you have a single default export.”*

```
// bad
import * as AirbnbStyleGuide from "./AirbnbStyleGuide";

// good
import AirbnbStyleGuide from "./AirbnbStyleGuide";
```

- And do not export directly from an import.

> *“Why? Although the one-liner is concise, having one clear way to import and one clear way to export makes things consistent.”*

```
// bad
// filename es6.js
export { es6 as default } from "./AirbnbStyleGuide";

// good
// filename es6.js
import { es6 } from "./AirbnbStyleGuide";
export default es6;
```

Modules

- Only import from a path in one place. eslint: **no-duplicate-imports**

> *"Why? Having multiple lines that import from the same path can make code harder to maintain."*

```
// bad
import foo from "foo";
// … some other imports … //
import { named1, named2 } from "foo";

// good
import foo, { named1, named2 } from "foo";

// good
import foo, {
  named1,
  named2,
} from "foo";
```

Modules

- Do not export mutable bindings. eslint: **`import/no-mutable-exports`**

> *“Why? Mutation should be avoided in general, but in particular when exporting mutable bindings. While this technique may be needed for some special cases, in general, only constant references should be exported.”*

```
// bad
let foo = 3;
export { foo };

// good
const foo = 3;
export { foo };
```

Modules

- In modules with a single export, prefer default export over named export. eslint: **`import/prefer-default-export`**

> *"Why? To encourage more files that only ever export one thing, which is better for readability and maintainability."*

```
// bad
export function foo() {}

// good
export default function foo() {}
```

Modules

- Put all `import` s above non-import statements. eslint: **`import/first`**

> *"Why? Since `import` s are hoisted, keeping them all at the top prevents surprising behavior."*

```
// bad
import foo from "foo";
foo.init();

import bar from "bar";

// good
import foo from "foo";
import bar from "bar";

foo.init();
```

Modules

- Multiline imports should be indented just like multiline array and object literals. eslint: **`object-curly-newline`**

> *"Why? The curly braces follow the same indentation rules as every other curly brace block in the style guide, as do the trailing commas."*

```
// bad
import {longNameA, longNameB, longNameC, longNameD, longNameE} from "path";

// good
import {
  longNameA,
  longNameB,
  longNameC,
  longNameD,
  longNameE,
} from "path";
```

Airbnb JS Style Guide

Full Stack Web Development