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

Strings

Objects

Properties

Destructuring

Strings

Strings

- Strings that cause the line to go over 100 characters should not be written across multiple lines using string concatenation.

  > *“Why? Broken strings are painful to work with and make code less searchable.”*

```
// bad
const errorMessage = "This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast."";

// bad
const errorMessage = "This is a super long error that was thrown because " +
  "of Batman. When you stop to think about how Batman had anything to do " +
  "with this, you would get nowhere fast.";

// good
const errorMessage = "This is a super long error that was thrown because of Batman. When you stop to think ab
```

Strings

- When programmatically building up strings, use template strings instead of concatenation. eslint: `prefer-template` `template-curly-spacing`

> *"Why? Template strings give you a readable, concise syntax with proper newlines and string interpolation features."*

```
// bad
function sayHi(name) {
  return "How are you, " + name + "?";
}

// bad
function sayHi(name) {
  return ["How are you, ", name, "?"].join();
}

// bad
function sayHi(name) {
  return `How are you, ${ name }?`;
}

// good
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

Strings

- Never use `eval()` on a string, it opens too many vulnerabilities. eslint: `no-eval`
- Do not unnecessarily escape characters in strings. eslint: `no-useless-escape`

> *"Why? Backslashes harm readability, thus they should only be present when necessary."*

```
// bad
const foo = "\"this\" \i\s \"quoted\"";

// good
const foo = "\"this\" is "quoted"";
const foo = `my name is "${name}"`;
```

Objects

Objects

- Use the literal syntax for object creation. eslint: `no-new-object`

```
// bad
const item = new Object();

// good
const item = {};
```

Objects

- Use computed property names when creating objects with dynamic property names.

> *"Why? They allow you to define all the properties of an object in one place."*

```
function getKey(k) {
  return `a key named ${k}`;
}

// bad
const obj = {
  id: 5,
  name: "San Francisco",
};
obj[getKey("enabled")] = true;

// good
const obj = {
  id: 5,
  name: "San Francisco",
  [getKey("enabled")]: true,
};
```

Objects

- Use object method shorthand. eslint: `object-shorthand`

```
// bad
const atom = {
  value: 1,

  addValue: function (value) {
    return atom.value + value;
  },
};

// good
const atom = {
  value: 1,

  addValue(value) {
    return atom.value + value;
  },
};
```

Objects

- Use property value shorthand. eslint: `object-shorthand`

> *"Why? It is shorter and descriptive."*

```
const lukeSkywalker = "Luke Skywalker";

// bad
const obj = {
  lukeSkywalker: lukeSkywalker,
};

// good
const obj = {
  lukeSkywalker,
};
```

Objects

- Group your shorthand properties at the beginning of your object declaration.

> *"Why? It's easier to tell which properties are using the shorthand."*

```
const anakinSkywalker = "Anakin Skywalker";
const lukeSkywalker = "Luke Skywalker";

// bad
const obj = {
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  lukeSkywalker,
  episodeThree: 3,
  mayTheFourth: 4,
  anakinSkywalker,
};

// good
const obj = {
  lukeSkywalker,
  anakinSkywalker,
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  episodeThree: 3,
  mayTheFourth: 4,
};
```

Objects

- Only quote properties that are invalid identifiers. eslint: `quote-props`

> *"Why? In general we consider it subjectively easier to read. It improves syntax highlighting, and is also more easily optimized by many JS engines."*

```
// bad
const bad = {
  "foo": 3,
  "bar": 4,
  "data-blah": 5,
};

// good
const good = {
  foo: 3,
  bar: 4,
  'data-blah': 5,
};
```

Objects

- Do not call `Object.prototype` methods directly, such as `hasOwnProperty`, `propertyIsEnumerable`, and `isPrototypeOf`. eslint: no-prototype-builtins

> *"Why? These methods may be shadowed by properties on the object in question - consider `{ hasOwnProperty: false }` - or, the object may be a null object `( Object.create(null) )`."*

```
// bad
console.log(object.hasOwnProperty(key));

// good
console.log(Object.prototype.hasOwnProperty.call(object, key));

// best
const has = Object.prototype.hasOwnProperty; // cache the lookup once, in module
console.log(has.call(object, key));
/* or */
import has from 'has'; // https://www.npmjs.com/package/has
console.log(has(object, key));
```

Properties

Properties

- Use dot notation when accessing properties. eslint: `dot-notation`

```
const luke = {
  jedi: true,
  age: 28,
};

// bad
const isJedi = luke['jedi'];

// good
const isJedi = luke.jedi;
```

Properties

- Use bracket notation `[]` when accessing properties with a variable.

```
const luke = {
  jedi: true,
  age: 28,
};

function getProp(prop) {
  return luke[prop];
}

const isJedi = getProp('jedi');
```

- Use exponentiation operator `**` when calculating exponentiations. eslint: `no-restricted-properties` .

```
// bad
const binary = Math.pow(2, 10);

// good
const binary = 2 ** 10;
```

Destructuring

Destructuring

- Use object destructuring when accessing and using multiple properties of an object. eslint: `prefer-destructuring`

> *"Why? Destructuring saves you from creating temporary references for those properties, and from repetitive access of the object. Repeating object access creates more repetitive code, requires more reading, and creates more opportunities for mistakes. Destructuring objects also provides a single site of definition of the object structure that is used in the block, rather than requiring reading the entire block to determine what is used."*

```
// bad
function getFullName(user) {
  const firstName = user.firstName;
  const lastName = user.lastName;

  return `${firstName} ${lastName}`;
}

// good
function getFullName(user) {
  const { firstName, lastName } = user;
  return `${firstName} ${lastName}`;
}

// best
function getFullName({ firstName, lastName }) {
  return `${firstName} ${lastName}`;
}
```

Destructuring

- Use array destructuring. eslint: `prefer-destructuring`

```
const arr = [1, 2, 3, 4];

// bad
const first = arr[0];
const second = arr[1];

// good
const [first, second] = arr;
```

Destructuring

- Use object destructuring for multiple return values, not array destructuring.

> *"Why? You can add new properties over time or change the order of things without breaking call sites."*

```
// bad
function processInput(input) {
  // then a miracle occurs
  return [left, right, top, bottom];
}

// the caller needs to think about the order of return data
const [left, __, top] = processInput(input);

// good
function processInput(input) {
  // then a miracle occurs
  return { left, right, top, bottom };
}

// the caller selects only the data they need
const { left, top } = processInput(input);
```

Airbnb JS Style Guide

Full Stack Web Development