The How and Why of the AirBnB Style Guide

Matthew Radcliffe
@mattkineme
mradcliffe

Kosada Incorporated

- Kosada in Columbus and Athens, Ohio
    - Upgrade sites, untangle messes, support and services provider
- Drupal core and contrib developer since 2007
    - PostgreSQL
    - Maintain a bunch of modules (xero, footermap, freelinking, etc…)
    - Opinionated
- Front end developer
    - React, Redux, Angular, AngularJS, Ampersand, JQuery, Vanilla.
- Board game designer, party parrot apologist  , Q/A, devops, Higher EDU nemesis.

Summary

- Code standards - What are they good for?
- AirBnB Style Guide - Getting Started with ESLint
- Opinionated run-through of specific items

Code Standards

- Benefits
  - Consistent
  - Readable
  - Predictable
  - Efficient
- Risks
  - Opinionated ("Strict")
  - Restrictive?

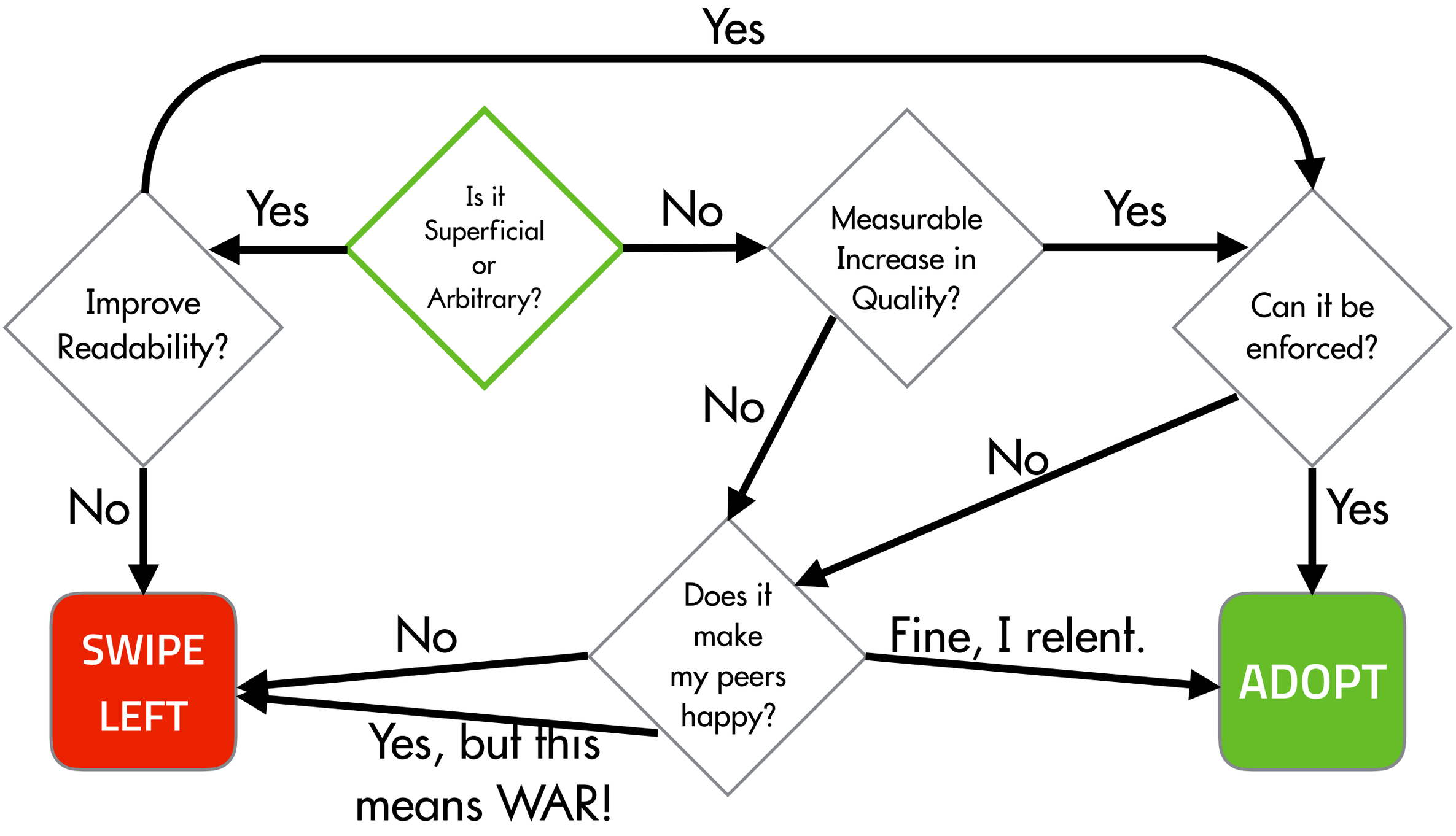
Evaluate Standards

- The easiest way to get started is to accept and enforce an entire standard,
    - But I like to know what I am agreeing to before I do something.
- Develop your own criteria for evaluating standards,
    - But research what others have done first.
    - "What rules do the community tend to override?"

AirBnB Issue Queue. What rules do the community tend to override? https://github.com/airbnb/javascript/issues/1089.

My Criteria

- This is my personal criteria though we probably all have our own criteria:
    - Should improve readability for arbitrary or superficial code standards.
    - Should have a measurable or demonstrable code quality improvement.
    - Should be automatable (predictability).
    - Should accept defeat to reduce stress in various communities.


Yes
Improve Readability?
Yes
Is it Superficial or Arbitrary?
No
Measurable Increase in Quality?
Yes
Can it be enforced?
No
No
No
Yes
SWIPE LEFT
Does it make my peers happy?
No
Yes, but this means WAR!
Fine, I relent.
ADOPT

The ~~Greatest~~ Worst Debate

- 1 space, 2 spaces, 4 spaces, 8 spaces, or tabs.
  - Arbitrary for most languages.
  - Improves readability at 2, 4 spaces or tabs.
- It can be enforced automatically, but there is conflict:
  - Tabs make Kosada (and some clients) happy so I use spaces in my custom code.
  - Spaces make Drupal and JavaScript communities happy, and I'm no longer trying to be contrary on that point so I use spaces in my open source code.

The AirBnB Style Guide

- AirBnB decided to share with the world their JavaScript coding standards
  - https://github.com/airbnb/javascript
  - http://airbnb.io/javascript/
- But it is not definitive.
- Tools allow us to extend and override coding standards for our own projects.
- Drupal Core officially adopted the AirBnB style guide, with modifications, for all ES6 code.
  - But it is not binding for contributed modules (yet).

Implementing the AirBnb Style Guide

- No framework
  - `npm install eslint-config-airbnb-base eslint eslint-plugin-import --save-dev`

- React
  - `npm install eslint-config-airbnb eslint-plugin-jsx-a11y eslint-plugin-react eslint-plugin-import --save-dev`

- Configure .eslintrc.yml (or .json).

- Configure your editor to use it if it's not automatically configured to do so.

N.B. eslint-plugin-jsx-ally and eslint 4 issue: https://github.com/evcohen/eslint-plugin-jsx-ally/issues/283

```
extends: eslint-config-airbnb-base
# extends: eslint-config-airbnb
# extends:
  # - eslint-config-airbnb-base
  # - plugin:react/recommended

env:
  browser: true
  node: true
  es6: true

# parser:
#  ecmaFeatures:
#    jsx: true
```

Importance of Automation

- If there's one thing I learned from btopro it is automation.
  - Your coding standards should bet automated as much as possible.
  - The AirBnB style guide has most of its standards as eslint rules, but not all of them so human code review is always still important (maybe that will save our jobs for a little while).
- Add linting to your review and build process.
  - In JavaScript, this usually means adding to the scripts object in package.json. eslint will return a non-zero exit code and fail your build if it detects linting errors.

N.B. What I learned from btopro may or may not be the lesson he was trying to convey.

```
{
  "name": "dcp-airbnb-sample",
  "version": "1.0.0",
  "description": "AirBnB Sample Code",
  "main": "index.js",
  "scripts": {
    "lint": "eslint '*.js'",
    "test": "mocha --require test '*.spec.js'",
    "build": "npm run-script lint && npm test"
  },
  "author": "",
  "license": "MIT",
  "devDependencies": {
    "babel-core": "^6.25.0",
    "babel-preset-env": "^1.6.0",
    "chai": "^4.1.0",
    "eslint": "~3.19.0",
    "eslint-config-airbnb-base": "^11.2.0",
    "eslint-plugin-import": "^2.7.0",
    "mocha": "^3.4.2"
  },
  "dependencies": {
    "lodash": "^4.17.4"
  }
}
```

"Air"-only
makes me dumber
on a hot and restless summer night

a haiku by Matthew Radcliffe

Constants and Changing Stuff

- prefer-const, no-const-reassign, no-var
    - var is function-scoped and easy to mutate, which could lead to unintended side effects.
    - let and const are block-scoped.
    - Re-assigning variables is "bad" because it decreases readability.
        - Distinguish between these two by using const when a variable is constant and won't be re-assigned to a different value, and let when you do need to do that (for loop).

```
function constNoChange(foo, bar) {
  const foonobar = 'foo';
  const foobar = 'foobar';

  if (foo) {
    if (bar) {
      return foobar;
    }
    return foonobar;
  }
  return false;
}
```

```
function letGood(foo, bar) {
  if (foo) {
    let ret = 'foo';
    if (bar) {
      ret = 'foobar';
    }
    return ret;
  } else if (bar) {
    const ret = 'a bar without foo is no bar at all';
    return ret;
  }

  return false;
}
```

A Quiver Full of Arrows

- Arrow Functions `() => {}` / `() => ()` / `=>`

  - Improved code quality and reduce code smell by inheriting scope and allowing brevity for map/reduce/filter functionality.

  - Arrow functions in functional programming languages were really confusing for me to read, but I have come to enjoy using them with the best practices in the AirBnB style guide.

  - rules: prefer-arrow-callback, arrow-spacing, arrow-parens, arrow-body-style, no-confusing-arrow.

```
const fixture = [
  { uid: 1, name: 'admin', mail: 'admin@example.com', field_lastname: '' },
  { uid: 6, name: 'btopro', mail: 'btopro@example.com', field_lastname: 'Ollendyke' },
  { uid: 30, name: 'mradcliffe', mail: 'mradcliffe@softpixel.com', field_lastname: 'Radcliffe' },
];

const btopro = fixture.find(user => user.name === 'btopro');
const uids = fixture.map(user => user.uid);

const hasLastNameOf = function hasLastNameOf(user, lastname) {
  return user.field_lastname === lastname;
};
```

```
const radcliffesBeforeOllendykes = fixture.sort((aUser, bUser) => {
  if (aUser.uid < bUser.uid) {
    if (hasLastNameOf(aUser, 'Ollendyke') && hasLastNameOf(bUser, 'Radcliffe')) {
      return 1;
    }
    return -1;
  } else if (aUser.uid > bUser.uid) {
    if (hasLastNameOf(bUser, 'Ollendyke') && hasLastNameOf(aUser, 'Radcliffe')) {
      return -1;
    }
    return 1;
  }
  return 0;
});
```

Declarations vs Expressions

- The style guide favors assigning named function expressions rather than anonymous functions or declared functions (func-style).
    - The benefit over naming a function expression is clear - it makes it easier to debug.
- Declared functions are hoisted, which means it is possible to use the function before it is declared. This can be confusing, but it is a preference and not objectively better to do one way or the other. However no-use-before-define should restrict that anyway.
    - Most of the following examples choose to use function declarations because I'm not using them before I define them.

ESLint docs. enforce the consistent use of either function declarations or expressions (func-style). https://eslint.org/docs/rules/func-style.

```
// anonFunction is an anonymous function expression that is harder to debug.
const anonFunction = bar => `foo${bar}`;

// hoistedFunction declaration is hoisted above its call, which is confusing.
hoistedFunction('bar');

function hoistedFunction(bar) {
  return `foo${bar}`;
}
```

```
const nonHoistedFunction = function namedFunction(bar) {
  return `foo${bar}`;
};

const fooBarObject = {
  foobar: function namedFunction(bar) {
    return `foo${bar}`;
  },
};
```

Use of "Static" Methods

- class-methods-use-this restricts the EcmaScript 6 Class concept so that any method that does not use object properties via this should be declared static.

    - This has a slight code quality improvement as well as semantic meaning.

- But static methods aren't inheritable so base classes might not use this, but their children do.

- Neither the airbnb style guide nor ESLint provide any good reason for adopting this standard. Notably some React component render methods do not use this.

    - Make this a warning or add method exceptions for your classes.

```
class Model {
  constructor(values = {}) {
    this.value = {};
    this.setValues(values);
  }
  getDefinition() {
    return [{ name: 'id', required: false }];
  }
  setValues(values) {
    this.getDefinition().forEach((definition) => {
      if (undefined !== values[definition.name]) {
        this.value[definition.name] = values[definition.name];
      } else if (undefined === values[definition.name] && undefined !== definition.default) {
        this.value[definition.name] = definition.default;
      } else if (undefined === values[definition.name] && definition.required) {
        throw new Error(`Missing required property ${definition.name}.`);
      } else {
        this.value[definition.name] = null;
      }
    });
  }
}

class UserModel extends Model {
  getDefinition() {
    return super.getDefinition().concat([
      { name: 'name', required: true },
      { name: 'mail', required: true },
      { name: 'picture', required: false, default: '' },
    ]);
  }
}
```

Template your strings

- prefer-template suggests to use templatized strings instead of string concatenation or any other method for creating a dynamic string.
    - ECMAScript 6 introduced templatized strings so we should use them.
    - Keeping strings on one line is easier to search for improved maintainability.
    - Does string interpolation improve readability?
    - Is there a better way to organize a long paragraph it than in code?

```
function getNonTemplateUrl(baseUrl, endpoint, queryParams = []) {
  let bad = baseUrl + '/' + endpoint + '?';
  bad += queryParams.reduce((result, param) => {
    return result + param.key + '=' + param.value;
  }, '');
  return bad;
}
```

```
function getTemplateUrl(baseUrl, endpoint, queryParams = []) {
  const paramString = queryParams.reduce((result, param) => {
    return `${result}${param.key}=${param.value}`;
  }, '');
  return `${baseUrl}/${endpoint}?${paramString}`;
}

export { getNonTemplateUrl, getTemplateUrl };
```

Destructuring: Get What You Need

- Destructuring is a new concept that allows us to extract properties from objects or arrays without creating temporary values in memory using assignment.

- The style guide recommends using object destructuring for returning complex data rather than arrays for maintainability.

- Both array and object destructuring should be used to pull out data respectively.

```
export default class FakeComponent {
  constructor(props) {
    this.props = Object.assign({}, props);
  }
  render() {
    return `<div>${this.formatPicture()}${this.formatUsername()}</div>`;
  }
  formatPicture() {
    // Bad.
    const pictureUrl = this.props.picture;
    const name = this.props.name;
    return `<figure><img src="${pictureUrl}" /><figcaption>${name}'s
avatar</figcaption></figure>`;
  }
  formatUsername() {
    // Good.
    const { name, mail } = this.props;
    return `<a href="mailto:${mail}">${name}</a>`;
  }
  getProperties() {
    const { name, mail } = this.props;
    return { name, mail };
  }
}
```

Spreading Out

- prefer-spread suggests to use the spread operator (…) to expand an array into its parts.

- Using this rule simplifies code that needs to send in a variable number of arguments into a function.

- The alternative is to use a prototype builtin.

```
// Old style.
function dateArgument(params) {
  const dateParams = [null].concat(params);
  return new (Function.prototype.bind.apply(Date, dateParams));
}

// params = ['2017', '09', '21'];
function dateSpread(params) {
  return new Date(...params);
}

// Copy an array argument so that we do not slip up and mutate it.
function copyWithFoo(arr = []) {
  const copy = [...arr];
  copy.push({ foo: 'bar' });
  return copy;
}

export {
  dateSpread,
  dateArgument,
  copyWithFoo,
};
```

Resting In

- prefer-rest-params suggests to use the rest operator (...) to collect variable function arguments instead of using the special arguments variable.

- The rest operator is the semantic opposite of the spread operator.

- This makes it easier to create functions similar to the Date builtin.

- Mutating the special arguments is quirky.

```
export default function foo(bar, ...optionalArgs) {
  const argMap = ['blah', 'halb'];
  const value = { foo: bar };

  optionalArgs.forEach((arg, index) => {
    const property = argMap[index];
    value[property] = arg;
  });

  return value;
}
```

Who Iterates the Iterators?

- no-iterator, guard-for-in are standards in the guide that restrict the use of for...in and for loops.
    - This is "good" functional programming practice as builtin iterators mutate values.
- Higher-order functions are available in ES5 so we should be using them now, but what about objects?
    - Need to pull in libraries like lodash or async when dealing with objects. Drupal core already has underscore, jquery, etc… available.
    - Also remember that HTML NodeLists are NOT arrays in all browsers, and thus you may still need an iterator.
    - Conclusion: acceptable to override as a warning.

```
import * as _ from 'lodash';

const objFixture = {
  1: { uid: 1, name: 'admin', mail: 'admin@example.com' },
  6: { uid: 6, name: 'btopro', mail: 'btopro@example.com' },
  30: { uid: 30, name: 'mradcliffe', mail: 'mradcliffe@softpixel.com' },
};
const forInValue = [];

for (let n in objFixture) {
  forInValue.push(objFixture[n]);
}

const lodashValue = _.flatMap(objFixture, item => item, []);

export {
  forInValue,
  lodashValue,
};
```

What About Promises?

- AirBnB doesn't offer any style guides for promises, but here are some tips I've learned over the years.
    - Chain, don't fan.
    - Good naming (past tense for promise variables, reject should be called error).
    - Avoid returning data not in a promise.
    - Synchronous code above asynchronous code.
    - Avoid inappropriate chaining (e.g. multiple synchronous gets and apply then to all of them)
        - Use Promise.all instead.
    - Avoid inspecting promise state (chai-as-promised exception for unit tests).

Neal Lindsay. Good Patterns Using Promises. ColumbusJS User Group Meetup. 2016/01/20.
Dave Atchley. ES6 Promise Anti-Patterns. http://www.datchley.name/promise-patterns-anti-patterns/. 2015/07/15.

```
function getUserContentFanned(name) {
  return fetch('/user.json')
    .then((users) => {
      const user = _.find(users, account => account.name === name);
      if (undefined === user) {
        throw new Error('No user by that name.');
      }
      return fetch('/node.json')
        .then((content) => {
          const nodes = _.find(content, node => node.uid === user.uid);
          if (undefined === nodes) {
            throw new Error('No nodes for that user name.');
          }
          return nodes;
        });
    });
}
```

```
function getUserContentChained(name) {
  return fetch('/user.json')
    .then(users => (_.find(users, user => user.name === name)))
    .then((user) => {
      if (undefined === user) {
        throw new Error('No user by that name.');
      }
      return Promise.all([user.uid, fetch('/node.json')]);
    })
    .then((results) => {
      const [uid, content] = results;
      const nodes = _.find(content, node => (node.uid === uid));
      if (undefined === nodes) {
        throw new Error('No nodes for that user name.');
      }
      return nodes;
    });
}
```

Other Notable Standards

- Hoisting: it is important to read this section thoroughly as you adopt ES6 and ES7.
- jQuery:
  - Name returned jQuery objects with $ for clarity.
  - Reduce the number of DOM lookups.

ES6+ Features and Future Standards

- Trailing commas in function parameter lists and calls (ES7)
- Async/Await
- Generators
- ES6 Generators: AirBnB teams chose not to use these because it is harder to transpile to ES5. This probably can be ignored now. Generator spacing?

Overriding The "Dumb" Stuff

- So if you think some of these rules are "dumb", then you can override them!

- Use the rules property in your eslint configuration file (.eslintrc.yml or .eslintrc.json).

  - Use an integer 0, I or 2 to indicate ignore, warn or error respectively.

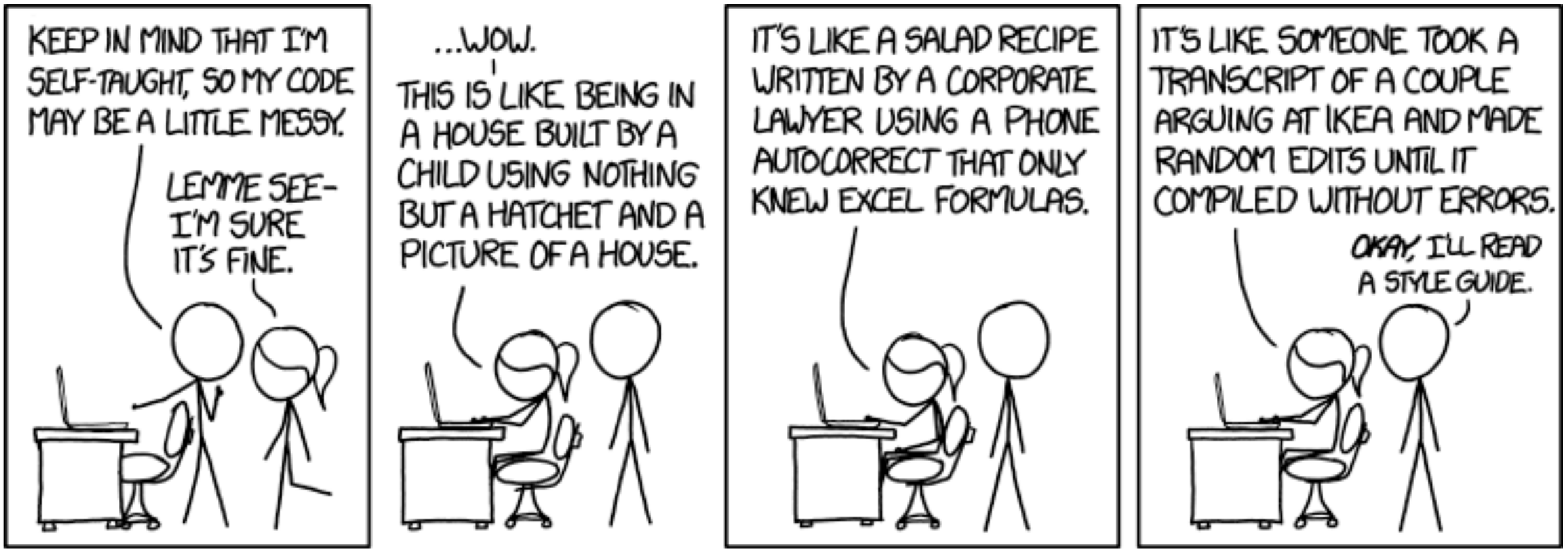
  - Some options have additional configuration in which case the value is an array.

```
global:
  Drupal: true
  drupalSettings: true
  drupalTranslations: true
  domready: true
  jQuery: true
  _: true
  matchMedia: true
  Backbone: true
  Modernizr: true
  CKEDITOR: true

rules:
  consistent-return: 0
  no-underscrore-dangle: 0
  max-nested-callbacks:
    - 1
    - 3
no-mutable-exports: 1
no-plusplus:
  - 1
  -
    allowForLoopAfterthoughts: true
no-param-reassign: 0
no-prototype-builtins: 0
valid-jsdoc:
  - 1
  -
    prefer:
      returns: return
      property: prop
brace-style:
  - error
  - stroustrup
no-unused-vars: 1
```

Conclusions

- The AirBnB style guide offers a fairly good "guide" to adopt in your project for ES6 code.
- All members of your team should take a day to read through the guide and understand it.
- Then (politely) decide what you want to override.
- Add eslint (or jscs) to your build process.



Questions? Opinions?

"Air"-only
makes me dumber
on a hot and restless summer night

a haiku by Matthew Radcliffe

エアーのみ
頭悪くて
熱帯夜

ラッドクリフ