GoodJsCode™ Guidebook

Write clean and elegant code, by following the good practices 🚀

Hi! 👋 I'm Pierre-Henry Soria. An enthusiastic and passionate software engineer. Originally from Brussels, (Belgium), I'm currently living in the wonderful land called "Australia" (Adelaide).

I've been coding for over 10 years and I decided to share my knowledge in terms of writing good code.

On a daily basis, I review hundreds of lines of code. Brand new micro-services, new features, new refactoring, hotfix, and so on. I've seen so many different coding styles as well as good and bad coding habits from the developers I've been working with.

With this book, you will have **the essential to know**, straight to the solution of coding better and cleaner. It's a practical book. You won't have superfluous information. Just the important things.

Time is so valuable and important that I only want to share what you need to know, without unnecessary details, to be there only for making the book fatter.

📖 Table of Contents

- The "One Thing" principle
- Don't comment what it does. Write what it does.
- Boat anchor - Unused code
- Minimalist code
- Reuse your code across your different projects by packing them into small NPM libraries
- Tests come first. Never Last
- Import only what you need
- Conditions into clear function names
- Readable Name: Variables
- Readable Name: Functions
- Readable Name: Classes
- Guard Clauses approach
- .gitignore and .gitattributes to every project
- Demeter Law

- Debugging efficiently
- Fewer arguments is more efficient
- Stub/mock only what you need
- Remove the redundant things
- Ego is your enemy
- Don’t use abbreviations
- American English spelling. The default choice when coding
- Destruct array elements in a readable way
- Readable numbers
- Avoid “else-statement”
- Prioritize `async` / `await` over Promises
- No magic numbers
- Always use `assert.strictEqual`
- Updating an object - The right way
- Stop using `Date()` when doing benchmarks
- Lock down your object
- Consider aliases when destructing an object
- Always use the strict type comparison
- Always write pure functions

The “One Thing” principle 1️⃣

When writing a function, remind yourself that it should ideally do only one thing. Think about what you learned already concerning the comments. The code should say everything. No comments should be needed. Splitting the code into small readable functions and reusable portions of code will drastically make your code readable and eliminate the need of copy/paste the same piece of code just because they haven’t been properly moved into reusable functions/classes.

❌ Non-readable function

```
function retrieveName(user) {
  if (user.name && user.name !=== 'admin' && user.name.length >= 5) {
    return user.name;
  }
}
```

✅ One Thing. Neat & Clean

```
const isRegularUser = (name) => {
  return name !=== config.ADMIN_USERNAME;
}

const isValidNameLength = (name, minimumLength = 5) => {
  return name.length >= minimumLength;
}

const isEligibleName(name) {
  return isRegularUser(name) && isValidNameLength(name);
}

// …

function retrieveName(user) {
  const name = user?.name;

  if (isEligibleName(name)) {
    return user.name;
  }
}
```

⬆️ Back to top

Don’t comment what it does ❌ Write what it does ✅

It’s crucial to name your functions and variable names exactly what they do.

If the code requires too many comments to be understood, the code needs to be refactored. The code has to be understood by reading it. Not by reading the comments. And the same applies when you write tests.

Your code has to be your comments. At the end of the day, as developers, we tend to be lazy and we don’t read the comment (carefully). However, the code, we always do.

❌ Bad practice

```
let validName = false;
// We check if name is valid by checking its length
if (name.length >= 3 && name.length <= 20) {
  validName = true;
  // Now we know the name is valid
  // …
}
```

✅ Good example

```
const isValidName = (name) => {
  return (
    name.length >= config.name.minimum && name.length <= config.name.maximum
  );
};
// …

if (isValidName('Peter')) {
  // Valid ✅
}
```

⬆️ Back to top

Boat anchor - Unused code

Never keep some unused code or commented code, *just in case* for history reason. Sadly, it's still very common to find commented code in PRs.

Nowadays, everybody uses a version control system like git, so there is always a history where you can go backwards.

❌ Downside of keeping unused code

1. We think we will come back removing it when it's time. Very likely, we will et busy on something else and we will forget removing it.
2. The unused code will add more complexity for a later refactoring.
3. Even if unused, it will still show up when searching in our codebase (which adds more complexity too).
4. For new developers joining the project, they don't know if they can remove it or not.

✅ Action to take

Add a BitBucket/GitHub pipeline or a git hook on your project level for rejecting any unused/commented code.

⬆️ Back to top

Minimalist code

Coding in a minimalist way is the best pattern you can follow.

Each time you need to create a new feature or add something to a project, see how you can reduce the amount of code. There are so many ways to achieve the solution. And there is always a shorten and clearer version which should always be the chosen one. Think twice before starting writing your code what would be the simplest solution you can write. Brain storm about it. Later, you will save much more time while writing your code.

⬆️ Back to top

Reuse your code across your different projects by packing them into small NPM libraries

❌ Wrong approach

My project is small (well, it always starts small). I don't want to spend time splitting functionalities into separated packages. Later on, somehow, my project grow bigger and bigger indeed. However, since I haven't spent time splitting my code into packages at the beginning. Now, I think it will take even longer to refactor my code into reusable packages. In short, I'm in a trap. My architecture is just not scalable.

✅ Right approach

Always think about reusibility. No matter how small or big is your project.

Splitting your code into small reusable external packages will always speed you up in the long run. For instance, there will be times where you will need a very similar functionality to be used in another project for another client's application.

Whether you are building a new project from scratch or implementing new features into it, always think about splitting your code early into small reusable internal NPM packages, so other potential products will be able to benefit from them and won't have to reinvent the wheel. Moreover, your code will gain in consistency thanks to reusing the same packages.

Finally, if there is an improvement or bug fix needed, you will have to change only one single place (the package) and not every impacted project.

🍰 Icing on the cake, you can make public some of your packages by open source them on GitHub and other online code repositories to get some free marketing coverage and potentially some good users of your library and contributors as well 💪

⬆️ Back to top

🏁 Tests come first. Never last

Never wait until the last moment to add some unit tests of the recent changes you have added.

Too many developers underestimate their tests during the development stage.

Don't create a pull request without unit tests. The other developers reviewing your pull request are not only reviewing your changes but also your tests.

Moreover, without unit tests, you have no idea if you introduce a new bug. Your new changes may not work as expected. Lastly, there will be chances you won't get time or rush up writing your tests if you push them for later.

❌ Stop doing

Stop neglecting the importance of unit tests. Tests are there to help you developing faster in the long run.

✅ Start doing

Create your tests even before changing your code. Create or update the current tests to expect the new behavior you will introduce in the codebase. Your test will then fail. Then, update the `src` of the project with the desired changes. Finally, run your unit tests. If the changes are correct and your tests are correctly written, your test should now pass 👍 Well done! You are now following the TDD development approach 💪

Remember, unit tests are there to save your day 🎉

⬆️ Back to top

Import only what you need

Have the good practice of importing only the functions/variables you need. This will prevent against function/variable conflicts, but also optimizing your code/improving readability by only expose the needed functions.

❌ Importing everything

```
import _ from 'lodash';

// …

if (_.isEmpty(something)) {
  _.upperFirst(something);
}
```

✅ Import only the needed ones

```
import { isEmpty as _isEmpty, upperFirst as _upperFirst } from 'lodash';

// …

if (_isEmpty(something)) {
  _upperFirst(something);
}
```

⬆️ Back to top

Conditions into clear function names

❌ Non-readable condition

```
const { active, feature, setting } = qrCodeData;

if (active && feature === 'visitor' && setting.name.length > 0) {
  // …
}
```

The condition is hard to read, long, not reusable, and would very likely need to be documented as well.

✅ Clear boolean conditional function

```
const canQrCode = ({ active, feature, setting }, featureName) => {
  return active && feature === 'visitor' && setting.name.length > 0;
};

// …

if (canQrCode(qrCodeData, 'visitor')) {
  // …
}
```

Here, the code doesn't need to be commented. The boolean function says what it does, producing a readable and clean code. Icing on the cake, the code is scalable. Indeed, the function `canQrCode` can be placed in a service or helper, increasing the reusability and decreasing the chance of duplicating code.

⬆️ Back to top

Readable Name: Variables

Mentioning clear good names and explicit names for your variables is critical to prevent confusions. Sometimes, we spend more time understanding what a variable is supposed to contain rather than achieving the given task.

❌ Bad Variable Name

```
// `nbPages` naming doesn't mean much ❌
const nbPages = postService.getAllItems();

let res = '';
for (let i = 1; i <= nbPages; i++) {
  res += '<a href="?page=' + i + '">' + i + '</a>';
  }
}
```

Giving `i` for the name of the increment variable is a terrible idea. Although, this is the standard for showing examples with the `for` loop, you should never do the same with your production application (sadly, plenty of developers just repeat what they've learned and we can't blame them, but it's now time to change!).

Every time you declare a new variable, look at the best and short words you can use to describe what it stores.

✅ Good example, with a clear variable name

```
// `totalItems` is a much better name ✅
const totalItems = postService.getAllItems();
// …

let htmlPaginationLink = '';
for (let currentPage = 1; currentPage <= totalItems; currentPage++) {
  htmlPaginationLink +=
    '<a href="?page=' + currentPage + '">' + currentPage + '</a>';
}
```

⬆️ Back to top

Readable Name: Functions

❌ Complicated (vague/unclear) function name

```
function cleaning(url) {
  // …
}
```

✅ Explicit descriptive name

```
function removeSpecialCharactersUrl(url) {
  // …
}
```

⬆️ Back to top

Readable Name: Classes

❌ Generic/Vague name

```
class Helper {
  // 'Helper' doesn't mean anything

  stripUrl(url) {
    // ...
    return url.replace('&amp;', '');
  }

  // ...
}
```

The class name is vague and doesn't clearly communicate what it does. By reading the name, we don't have a clear idea of what `Helper` contains and how to use it.

✅ Clear/Explicit name

```
class Sanitizer {
  // <- Name is already more explicit

  constructor(value) {
    this.value = value;
  }

  url() {
    this.value.replace('&amp;', '');
    // ...
  }
}
```

In this case, the class name clearly says what it does. It's the opposite of a black box. By saying what it does, it should also follow the single responsibility principle of achieving only one single purpose. Class names should be a (singular) noun that starts with a capital letter. The class can also contain more than one noun, if so, each word has to be capitalized (this is called **UpperCamel** case).

⬆️ Back to top

Guard Clauses approach

The guard clauses pattern is the way of leaving a function earlier by removing the redundant `else {}` blocks after a `return` statement.

Let's see a snippet that doesn't follow the guard clause pattern and a clean and readable example that does.

The two samples represents the body of a function. Inside of the function we have the following 👇

❌ The "not-readable" way

```
if (payment.bonus) {
  // … some logics
  if (payment.bonus.voucher) {
    return true; // eligible to a discount
  } else if (payment.confirmed) {
    if (payment.bonus.referral === 'friend') {
      return true;
    } else {
      return false;
    }
  }
  return false;
}
```

✅ Clean readable way

```
if (!payment.bonus) {
  return false;
}

if (payment.bonus.voucher) {
  return true;
}

if (payment.bonus.referral === 'friend') {
  return true;
}

return false;
```

On this example, we can notice how we could remove the complicated nested conditionals thanks to exiting the function as early as possible with the `return` statement.

⬆️ Back to top

.gitignore and .gitattributes to every project

When you are about to distribute your library, you should always create a `.gitignore` and `.gitattributes` in your project to prevent undesirable files to be presented in there.

✅ `.gitignore`

As soon as you commit files, your project needs a `.gitignore` file. It guarantees to exclude specific files from being committed in your codebase.

✅ `.gitattributes`

When you publish your package to be used by a dependency manager, it's crucial to exclude the developing files (such as the `tests` folders, `.github` configuration folder, `CONTRIBUTING.md`, `SECURITY.me`, `.gitignore`, `.gitattributes` itself, and so on...). Indeed, when you install a new package through your favorite package manager (npm, yarn, ...), **you only want to download the required source files, that's all** without including the test files, pipeline configuration files, etc, not needed for production.

⬆️ Back to top

Demeter Law

The **demeter law**, AKA the **principle of least knowledge** states that each unit of code should only have very limited knowledge about other units of code. They should only talk to their close friends, not to their strangers 🙃 It shouldn't have any knowledge on the inner details of the objects it manipulates.

❌ Chaining methods

```
object.doA()?.doB()?.doC(); // violated deleter's law
```

Here, `doB` and `doC` might receive side-effects from their sub-chaining functions.

✅ Non-chaining version

```
object.doA();
object.doB();
object.doC();
```

Each method doesn't rely on each other. They are independent and safe from eventual refactoring.

⬆️ Back to top

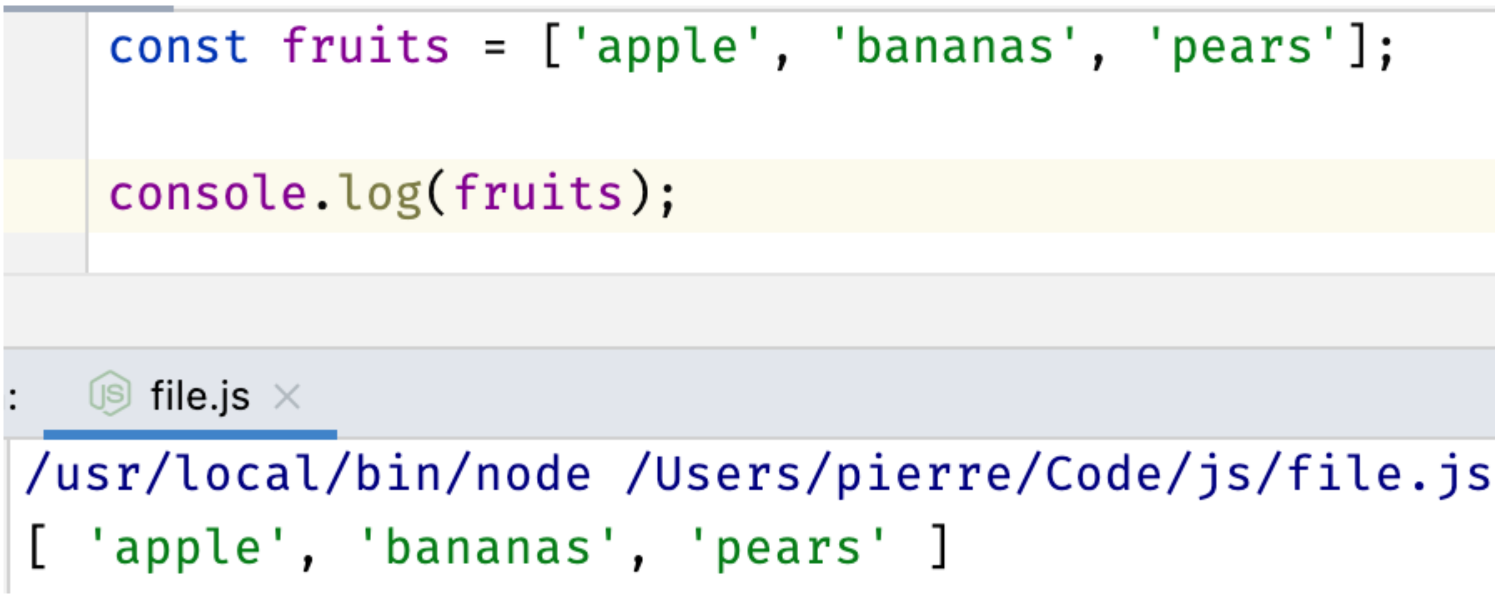
Debugging efficiently

When debugging with arrays or objects, it's always a good practice to use `console.table()`

❌ `console.log(array)`

`console.log` makes the result harder to read. You should always aim for the most efficient option.

```
const favoriteFruits = ['apple', 'bananas', 'pears'];
console.log(favoriteFruits);
```



```
const fruits = ['apple', 'bananas', 'pears'];

console.log(fruits);
```

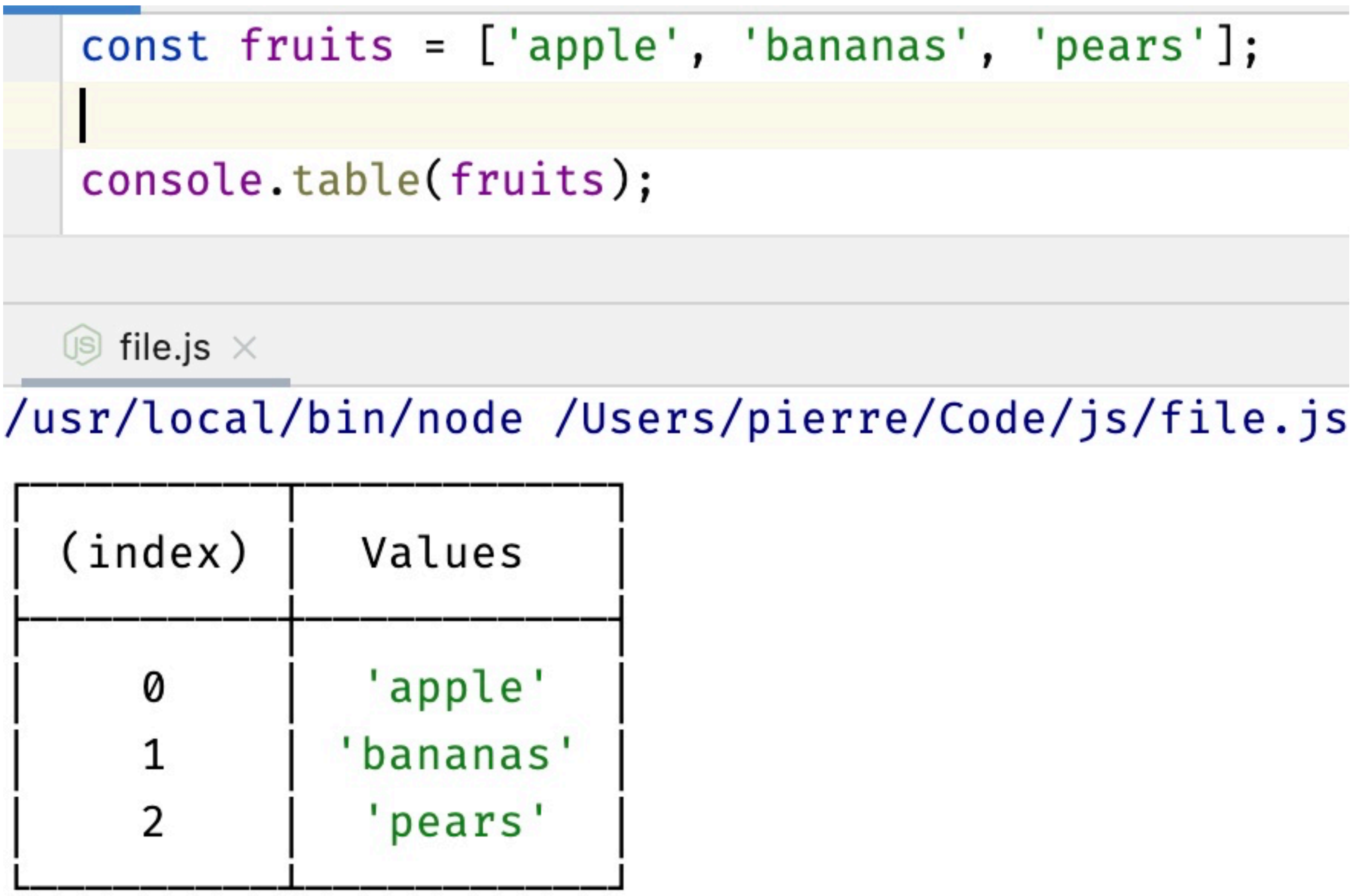
```
/usr/local/bin/node /Users/pierre/Code/js/file.js
[ 'apple', 'bananas', 'pears' ]
```

✅ `console.table(array)`

Using `console.table` saves you time as the result is shown in a clear table, improving the readability of the log when debugging an array or object.

Mozilla gives us a clear example to see how `console.table` can help you.

```
const favoriteFruits = ['apple', 'bananas', 'pears'];
console.table(favoriteFruits);
```



```
const fruits = ['apple', 'bananas', 'pears'];

console.table(fruits);
```

file.js

```
/usr/local/bin/node /Users/pierre/Code/js/file.js
┌─────────┬───────────┐
│ (index) │  Values   │
├─────────┼───────────┤
│    0    │  'apple'  │
│    1    │ 'bananas' │
│    2    │  'pears'  │
└─────────┴───────────┘
```

⬆️ Back to top

Fewer arguments is more efficient

If your functions have more than 3 arguments, it means you could have written a much better and cleaner code. In short, the purpose of your function does too much and violates the single responsibility principle, leading to debugging and reusability hells.

In short, the fewer arguments your function has, the more efficient it will become as you will prevent complexity in your code.

❌ Unclean code

```
function readItem(
  id: number,
  userModel: UserModel,
  siteInfoModel: SiteModel,
  security: SecurityCheck
) {}
```

✅ Clean code

```
user = new User(id);
// …
function readItem(user: User) {}
```

With this version, because it has only relevant arguments, reusing the function elsewhere will be possible as the function doesn’t rely or depend on unnecessary arguments/objects.

⬆️ Back to top

Stub/mock only what you need

When you stub an object (with Sinon for instance), it’s a good and clean practice to decide only what functions you need to stub out, instead of stubbing out the entier object. Doing so, you also prevent overriding internal implementations of functions which cause all sorts of inconsistencies in your business logic.

❌ Everything is stub

```
sinon.stub(classToBeStubbed);
```

✅ Stub only what you need

```
sinon.stub(classToBeStubbed, 'myNeededFunction');
```

Here, we stub only the individual method we need.

⬆️ Back to top

Remove the redundant things

When we code, we often tend to write "unnecessary" things, which don't increase the readability of the code either.

For instance, in a switch-statement, having a `default` clause that isn’t used.

❌ Redundant version

```js
const PRINT_ACTION = 'print';
const RETURN_ACTION = 'return';
// …
switch (action) {
  case PRINT_ACTION:
    console.log(message);
    break;

  case RETURN_ACTION:
    return message;
    break; // ❌ This is redundant as we already exit the `switch` with `return`

  default:
    break; // ❌ This is redundant
}
```

✅ Neat version

```js
const PRINT_ACTION = 'print';
const RETURN_ACTION = 'return';
// …
switch (action) {
  case PRINT_ACTION:
    console.log(message);
    break;
  case RETURN_ACTION:
    return message;

  default:
    throw new Error(`Invalid ${action}`);
}
```

⬆️ Back to top

Ego is your enemy ✋

Too often I see developers taking the comments on their pull requests personally because they see what they have done as their own creation. When you receive a change request, don't feel judged! This is actually an improvement reward for yourself 🏆

If you want to be a good developer, leave your ego in your closet. Never bring it to work. This will just slow your progression down and could even pollute your brain space and the company culture.

❌ Taking what others say as personally

✅ See every feedback as a learning experience

> When you write code, it's not your code, it's everybody's else code. Don't take what you write personally. It's just a little part of the whole vessel.

⬆️ Back to top

Don't use abbreviations 😐

Abbreviations are hard to understand. For new joiners who aren't familiar with the company's terms. And even with common English abbreviations, they can be quite difficult to be understood by non-native English speakers who might be hired in the future or when outsourcing developers from overseas.

Having as a golden rule to never use abbreviations in your codebase (so at least, you'll never have to take any further decisions on this topic) is critical for preventing confusion later on.

❌ Difficult to read. Hard to remember over time

```
if (type === Type.PPL_CTRL) {
  // We are on People Controller
  // Logic comes here
}

if (type === Type.PPL_ACT) {
  // We are on People Action
  // …
}

if (type === Type.MSG_DMN_EVT) {
  // We are on Messaging Domain Event
  // …
}

// ...
```

```
const IndexFn = () => {
  // index function
};
```

✅ Clear names (without comments needed 👌)

```
if (type === Type.PEOPLE_CONTROLLER) {
  // Logic comes here
}

if (type === Type.PEOPLE_ACTION) {
  // Logic here
}

if (type === Type.MESSAGING_DOMAIN) {
  // Logic here
}

// ...

const index = () => {
  // No need to have `Fn` or `Function` as suffix in the name
  // Having `function` or `method` for an actual function is refundant is general
  // ...
};
```

⬆️ Back to top

🇺🇸 American English spelling. The default choice when coding

I always recommend to only use **US English** in your code. If you mix both British and American spellings, it will introduce some sort of confusions for later and might lead to interrogations for new developers joining the development of your software.

Most of the 3rd-party libraries are written in American English. As we use them in our codebase, we should prioritize US English as well in our code. I've seen codebase with words such as "*licence*" and "*license*", "*colour*" and "*color*", or "*organisation*" and "*organization*". When you need to search for terms / refactor some code, and you find both spellings, it requires more time and consumes further brain space, which could have been avoided at the first place by following a consistent spelling.

⬆️ Back to top

Destructing array elements - Make it readable

When you need to destruct an array with JavaScript (ES6 and newer), and you want to pickup only the second or third array, there is a much cleaner way than using the `,` to skip the previous array keys.

❌ Bad Way

```
const meals = ['Breakfast', 'Lunch', 'Apéro', 'Dinner'];

const [, , , favoriteMeal] = meals;
console.log(favoriteMeal); // Dinner
```

✅ Recommended readable way

```
const meals = [
  'Breakfast', // index 0
  'Lunch', // index 1
  'Apéro', // index 2
  'Dinner', // index 3
];

const { 3: favoriteMeal } = meals; // Get the 4th value, index '3'
console.log(favoriteMeal); // Dinner
```

Here, we destruct the array as an object with its index number and give an alias name `favoriteMeal` to it.

⬆️ Back to top

Readable numbers

With JavaScript, you can use **numeric separators** and **exponential notations** to make numbers easier to read. The execution of the code remains exactly the same, but it's definitely easier to read.

❌ Without numeric separators

```
const largeNumbers = 1000000000;
```

✅ Clear/readable numbers

```
const largeNumbers = 1_000_000_000;
```

Note: numeric separators works also with other languages than JavaScript such as Python, Kotlin, ...

⬆️ Back to top

Avoid “else-statement”

Similar to what I mentioned concerning the importance of writing portions of code that only do “**One Thing**”, you should also avoid using the *if-else statement*.

Too many times, we see code such as below with unnecessary and pointless `else` blocks.

❌ Conditions with unnecessary `else {}`

```
const getUrl = () => {
  if (options.includes('url')) {
    return options.url;
  } else {
    return DEFAULT_URL;
  }
};
```

This code could easily be replaced by a clearer version.

✅ Option 1. Use default values

```
const getUrl = () => {
  let url = DEFAULT_URL;

  if (options.includes('url')) {
    url = options.url;
  }
  return url;
```

By having a default value, we remove the need of a `else {}` block.

✅ Option 2. Use Guard Clauses approach

```
const getUrl = () => {
  if (options.includes('url')) {
    return options.url; // if true, we return `options.url` and leave the functio
  }
  return DEFAULT_URL;
};
```

With this approach, we leave the function early, preventing complicated and unreadable nested conditions in the future.

⬆️ Back to top

Prioritize `async/await` over Promises

❌ With promises

```
const isProfileNameAllowed = (id) => {
  return profileModel.get(id).then((profile) => {
    return Ban.name(profile.name);
  }).then((ban) => ({
    return ban.value
  }).catch((err) => {
    logger.error({ message: err.message });
  });
}
```

Promises make your code harder to read.

✅ With `async` / `await`

```
const isProfileNameAllowed = async (id) => {
  try {
    const profile await profileModel.get(id);
    const {value: result } = await Ban.name(profile.name);
    return result;
    } catch (err) {
      logger.error({ message: err.message });
    }
  }
}
```

By using `async` / `await`, you avoid callback hell, which happens with promises chaining when data are passed through a sert of functions and leads to unmanageable code due to its nested fallbacks.

⬆️ Back to top

No magic numbers 🔢

Avoid as much as you can to hardcode changeable values which could change over time, such as the amount of total posts to display, timeout delay, and other similar information.

❌ Code containing magic numbers

```
setTimeout(() => {
  window, (location.href = '/');
}, 3000);
```

```
getLatestBlogPost(0, 20);
```

✅ No hardcoded numbers

```
setTimeout(() => {
  window, (location.href = '/');
}, config.REFRESH_DELAY_IN_MS);
```

```
const POSTS_PER_PAGE = 20;

getLatestBlogPost(0, POSTS_PER_PAGE);
```

⬆️ Back to top

Always use `assert.strictEqual`

With your test assertion library (e.g. chai), always consider using the strict equal assertion method.

❌ Don't just use `assert.equal`

```
assert.equal('+63464632781', phoneNumber);
assert.equal(validNumber, true);
```

✅ Use appropriate strict functions from your assertion library

```
assert.strictEqual('+63464632781', phoneNumber);
assert.isTrue(validNumber);
```

⬆️ Back to top

Updating an object - The right way

❌ Avoid `Object.assign` as it's verbose and longer to read

```
const user = Object.assign(data, {
  name: 'foo',
  email: 'foo@bar.co',
  company: 'foo bar inc',
});
```

✅ Use destructing assignment, spread syntax

```
const user = {
  ...data,
  name: 'foo',
  email: 'foo@bar.co',
  company: 'foo bar inc',
};
```

⬆️ Back to top

Stop using `Date()` when doing benchmarks

JavaScript is offering a much nicer alternative when you need to measure the performance of a page during a benchmark.

❌ Stop doing

```
const start = new Date();
// your code ...
const end = new Date();

const executionTime = start.getTime() - end.getTime();
```

✅ With ` performance.now()

```
const start = performance.now();
// your code ...
const end = performade.now();

const executionTime = start - end;
```

⬆️ Back to top

Lock down your object 🔐

It's always a good idea to const lock the properties when creating an object. That way, the values of your properties object will only be read-only.

❌ Without locking an object

```
const canBeChanged = { name: 'Pierre' }:

canBeChanged.name = 'Henry'; // `name` is now "Henry"
```

✅ With `as const` to lock an object

```
const cannotBeChanged = { name: 'Pierre' } as const;
```

```
cannotBeChanged = 'Henry'; // Won't be possible. JS will throw an error as `name`
```

⬆️ Back to top

Consider aliases when destructing an object

❌ Without aliases

```
const { data } = getUser(profileId);
const profileName = data.name;
// ...
```

✅ With clear alias name

```
const { data: profile } = getUser(profileId);
// then, use `profile` as the new var name 🙂
const profileName = profile.name;
// ...
```

⬆️ Back to top

Always use the strict type comparison

When doing some kind of comparison, always use the `===` strict comparison.

❌ Don’t use loose comparisons

```
if ('abc' == true) {
} // this gives true ❌

if (props.address != details.address) {
} // result might not be what you expect
```

✅ Use strict comparisons

```
if ('abc' === true) {
} // This is false ✅

if (props.address !== details.address) {
} // Correct expectation
```

⬆️ Back to top

Always write pure functions

A function is only pure if the **given input always returns the same output**.

A pure function never produces side-effects, meaning that it cannot change any external states. The pure function only depends on its own arguments, and from the function's scope.

```
function multiply(x, y) {
  return x * y;
}
```

A more tricky scenario can occur when you are passing an object. Imagine you are passing a “user” object to another function. If you modify the object “user” in the function, it will modify the actual user object because the object passed in as parameter is actually a reference of the object, which is the opposite of a distinct new cloned object.

To prevent this downside, you will have to deep clone the object first (you can use the loads *cloneDeep*function) and then Object.freeze(copyUser) when returning it. This will guarantee the “copyUser” to be immutable.

For instance:

```
import { cloneDeep as _cloneDeep } from 'lodash';

function changeUser(user) {
  const copyUser = _cloneDeep(user); // copyUser = { …user };
  copyUser.name = 'Edward Ford';

  return Object.freeze(copyUser);
}
```

✅ By writing pure functions, you will make the code easier to be read and understood. You only need to focus your attention on the function itself without having to look at the surrounding environments, states and properties outside the function's scope, and preventing you to spend hours in debugging.

⬆️ Back to top

About the Author

Pierre-Henry Soria. A super passionate and enthusiastic software engineer, and a true cheese & chocolate lover 💫

☕️ Did you enjoy this guideline book (that will most likely take your developer career to the next level)? If so, would you consider **offering me a coffee**? 😊

TWITTER GITHUB