JavaScript Clean Coding Best Practices

Writing clean code is what you must know and do in order to call yourself a professional developer. There is no reasonable excuse for doing anything less than your best.

In this blog post, we will cover general clean coding principles for naming and using variables & functions, as well as some JavaScript specific clean coding best practices.

“Even bad code can function. But if the code isn’t clean, it can bring a development organization to its knees.” — Robert C. Martin (Uncle Bob)

First of all, what does clean coding mean?

Clean coding means that in the first place you write code for your later self and for your co-workers and not for the machine.

Your code must be easily understandable for humans.

"Write code for your later self and for your co-workers in the first place - not for the machine." via @RisingStack

You know you are working on a clean code when each routine you read turns out to be pretty much what you expected.

JavaSctipr Clean Coding: The only valid measurement of code quality is WTFs/minute

JavaScript Clean Coding Best Practices

Now that we know what every developer should aim for, let’s go through the best practices!

How should I name my variables?

Use intention-revealing names and don't worry if you have long variable names instead of saving a few keyboard strokes.

If you follow this practice, your names become searchable, which helps a lot when you do refactors or you are just looking for something.

// DON'T

```
let d
let elapsed
const ages = arr.map((i) => i.age)
```

// DO

```
let daysSinceModification
const agesOfUsers = users.map((user) => user.age)
```

Also, make meaningful distinctions and don't add extra, unnecessary nouns to the variable names, like its type (hungarian notation).

// DON'T

```
let nameString
let theUsers
```

// DO

```
let name
let users
```

Make your variable names easy to pronounce, because for the human mind it takes less effort to process.

When you are doing code reviews with your fellow developers, these names are easier to reference.

// DON'T

```
let fName, lName
let cntr

let full = false
if (cart.size > 100) {
  full = true
}
```

// DO

```
let firstName, lastName
let counter

const MAX_CART_SIZE = 100
// ...
const isFull = cart.size > MAX_CART_SIZE
```

In short, don't cause extra mental mapping with your names.

How should I write my functions?

Your functions should do one thing only on one level of abstraction.

Functions should do one thing. They should do it well. They should do it only. — Robert C. Martin (Uncle Bob)

// DON'T

```
function getUserRouteHandler (req, res) {
  const { userId } = req.params
  // inline SQL query
  knex('user')
    .where({ id: userId })
    .first()
    .then((user) => res.json(user))
}
```

// DO

```
// User model (eg. models/user.js)
const tableName = 'user'
const User = {
  getOne (userId) {
    return knex(tableName)
      .where({ id: userId })
      .first()
  }
}

// route handler (eg. server/routes/user/get.js)
function getUserRouteHandler (req, res) {
  const { userId } = req.params
  User.getOne(userId)
    .then((user) => res.json(user))
}
```

After you wrote your functions properly, you can test how well you did with CPU profiling - which helps you to find bottlenecks.

Use long, descriptive names

A function name should be a verb or a verb phrase, and it needs to communicate its intent, as well as the order and intent of the arguments.

A long descriptive name is way better than a short, enigmatic name or a long descriptive comment.

// DON'T

```
/**
 * Invite a new user with its email address
 * @param {String} user email address
 */
function inv (user) { /* implementation */ }
```

// DO

```
function inviteUser (emailAddress) { /* implementation */ }
```

Avoid long argument list

Use a single object parameter and destructuring assignment instead. It also makes handling optional parameters much easier.

// DON'T function getRegisteredUsers (fields, include, fromDate, toDate) { /* implementation */ }
getRegisteredUsers(['firstName', 'lastName', 'email'], ['invitedUsers'], '2016-09-26', '2016-12-13')

// DO function getRegisteredUsers ({ fields, include, fromDate, toDate }) { /* implementation */ }

getRegisteredUsers({
fields: ['firstName', 'lastName', 'email'], include: ['invitedUsers'], fromDate: '2016-09-26', toDate: '2016-12-13' })

Reduce side effects

Use pure functions without side effects, whenever you can. They are really easy to use and test.

// DON'T

```
function addItemToCart (cart, item, quantity = 1) {
  const alreadyInCart = cart.get(item.id) || 0
  cart.set(item.id, alreadyInCart + quantity)
  return cart
}
```

// DO

```
// not modifying the original cart
function addItemToCart (cart, item, quantity = 1) {
  const cartCopy = new Map(cart)
  const alreadyInCart = cartCopy.get(item.id) || 0
  cartCopy.set(item.id, alreadyInCart + quantity)
  return cartCopy
}

// or by invert the method location
// you can expect that the original object will be mutated
// addItemToCart(cart, item, quantity) -> cart.addItem(item, quantity)
const cart = new Map()
Object.assign(cart, {
  addItem (item, quantity = 1) {
    const alreadyInCart = this.get(item.id) || 0
    this.set(item.id, alreadyInCart + quantity)
    return this
  }
})
```

Organize your functions in a file according to the stepdown rule

Higher level functions should be on top and lower levels below. It makes it natural to read the source code.

// DON'T

```
// "I need the full name for something..."
function getFullName (user) {
  return `${user.firstName} ${user.lastName}`
}

function renderEmailTemplate (user) {
  // "oh, here"
  const fullName = getFullName(user)
  return `Dear ${fullName}, ...`
}
```

// DO

```
function renderEmailTemplate (user) {
  // "I need the full name of the user"
  const fullName = getFullName(user)
  return `Dear ${fullName}, ...`
}

// "I use this for the email template rendering"
function getFullName (user) {
  return `${user.firstName} ${user.lastName}`
}
```

Query or modification

Functions should either do something (modify) or answer something (query), but not both.

Everyone likes to write JavaScript differently, what to do?

As JavaScript is dynamic and loosely typed, it is especially prone to programmer errors.

Use project or company wise linter rules and formatting style.

The stricter the rules, the less effort will go into pointing out bad formatting in code reviews. It should cover things like consistent naming, indentation size, whitespace placement and even semicolons.

"The stricter the linter rules, the less effort needed to point out bad formatting in code reviews." by @RisingStack

The standard JS style is quite nice to start with, but in my opinion, it isn't strict enough. I can agree most of the rules in the Airbnb style.

How to write nice async code?

Use Promises whenever you can.

Promises are natively available from Node 4. Instead of writing nested callbacks, you can have chainable Promise calls.

// AVOID asyncFunc1((err, result1) => {
asyncFunc2(result1, (err, result2) => { asyncFunc3(result2, (err, result3) => { console.lor(result3) }) }) })

// PREFER asyncFuncPromise1()
.then(asyncFuncPromise2) .then(asyncFuncPromise3) .then((result) => console.log(result)) .catch((err) => console.error(err)) Most of the libraries out there have both callback and promise interfaces, prefer the latter. You can even convert callback APIs to promise based one by wrapping them using packages like es6-promisify.

// AVOID const fs = require('fs')

function readJSON (filePath, callback) {
fs.readFile(filePath, (err, data) => { if (err) { return callback(err) }

```
try {
  callback(null, JSON.parse(data))
} catch (ex) {
  callback(ex)
}
```

}) }

readJSON('./package.json', (err, pkg) => { console.log(err, pkg) })

// PREFER const fs = require('fs')
const promisify = require('es6-promisify')

const readFile = promisify(fs.readFile)
function readJSON (filePath) {
return readFile(filePath) .then((data) => JSON.parse(data)) }

readJSON('./package.json')
.then((pkg) => console.log(pkg)) .catch((err) => console.error(err)) The next step would be to use async/await (≥ Node 7) or generators with co (≥ Node 4) to achieve synchronous like control flows for your asynchronous code.

const request = require('request-promise-native')

function getExtractFromWikipedia (title) {
return request({ uri: 'https://en.wikipedia.org/w/api.php', qs: { titles: title, action: 'query', format: 'json', prop: 'extracts', exintro: true, explaintext: true }, method: 'GET', json: true }) .then((body) => Object.keys(body.query.pages).map((key) => body.query.pages[key].extract)) .then((extracts) => extracts[0]) .catch((err) => { console.error('getExtractFromWikipedia() error:', err) throw err }) }

// PREFER async function getExtractFromWikipedia (title) {
let body try { body = await request({ /* same parameters as above */ }) } catch (err) { console.error('getExtractFromWikipedia() error:', err) throw err }

const extracts = Object.keys(body.query.pages).map((key) => body.query.pages[key].extract) return extracts[0] }

// or const co = require('co')

const getExtractFromWikipedia = co.wrap(function * (title) {
let body try { body = yield request({ /* same parameters as above */ }) } catch (err) { console.error('getExtractFromWikipedia() error:', err) throw err }

const extracts = Object.keys(body.query.pages).map((key) => body.query.pages[key].extract) return extracts[0] })

getExtractFromWikipedia('Robert Cecil Martin')
.then((robert) => console.log(robert))

How should I write performant code?

In the first place, you should write clean code, then use profiling to find performance bottlenecks.

Never try to write performant and smart code first, instead, optimize the code when you need to and refer to true impact instead of micro-benchmarks.

"Write clean code first and optimize it when you need to. Refer to true impact instead of micro-benchmarks!"