# code-review-tips

## Table of Contents

1. Introduction
2. Why Review Code?
3. Basics
4. Readability
5. Side Effects
6. Limits
7. Security
8. Performance
9. Testing
10. Miscellaneous

## Introduction

Code reviews can inspire dread in both reviewer and reviewee. Having your code analyzed can feel invasive and uncomfortable. Even worse, reviewing other people's code can feel like a painful and ambiguous exercise, searching for problems and not even knowing where to begin.

This project aims to provide some solid tips for how to review the code that you and your team write. All examples are written in JavaScript, but the advice should be applicable to any project of any language. This is by no means an exhaustive list, but hopefully this will help you catch as many bugs as possible long before users ever see your feature.

## Why Review Code?

Code reviews are a necessary part of the software engineering process because you alone can't catch every problem in a piece of code you write. That's ok though! Even the best basketball players in the world miss shots.

Having others review our work ensures that we deliver the best product to users and with the least amount of errors. Make sure your team implements a code review process for new code that is introduced into your codebase. Find a process that works for you and your team. There's no one size fits all. The important point is to do code reviews as regularly as possible.

## Basics

### Code reviews should be as automated as possible

Avoid discussing details that can be handled by a static analysis tool. Don't argue about nuances such as code formatting and whether to use `let` or `var`. Having a formatter and linter can save your team a lot of time from reviews that your computer can do for you.

### Code reviews should avoid API discussion

These discussions should happen before the code is even written. Don't try to argue about the floor plan once you've poured the concrete foundation.

### Code reviews should be kind

It's scary to have your code reviewed and it can bring about feelings of insecurity in even the most experienced developer. Be positive in your language and keep your teammates comfortable and secure in their work!

## Readability

## Typos should be corrected

Avoid nitpicking as much as you can and save it for your linter, compiler, and formatter. When you can't, such as in the case of typos, leave a kind comment suggesting a fix. It's the little things that make a big difference sometimes!

## Variable and function names should be clear

Naming is one of the hardest problems in computer science. We've all given names to variables, functions, and files that are confusing. Help your teammate out by suggesting a clearer name, if the one you're reading doesn't make sense.

```
// This function could be better named as namesToUpperCase
function u(names) {
  // ...
}
```

## Functions should be short

Functions should do one thing! Long functions usually mean that they are doing too much. Tell your teammate to split out the function into multiple different ones.

```
// This is both emailing clients and deciding which are active. Should be
// 2 different functions.
function emailClients(clients) {
  clients.forEach(client => {
    const clientRecord = database.lookup(client);
    if (clientRecord.isActive()) {
      email(client);
    }
  });
}
```

## Files should be cohesive, and ideally short

Just like functions, a file should be about one thing. A file represents a module and a module should do one thing for your codebase.

For example, if your module is called `fake-name-generator` it should just be responsible for creating fake names like "Keyser Söze". If the `fake-name-generator` also includes a bunch of utility functions for querying a database of names, that should be in a separate module.

There's no rule for how long a file should be, but if it's long like below **and** includes functions that don't relate to one another, then it should probably be split apart.

```
// Line 1
import _ from 'lodash';
function generateFakeNames() {
  // ..
}

// Line 1128
function queryRemoteDatabase() {
  // ...
}
```

### Exported functions should be documented

If your function is intended to be used by other libraries, it helps to add documentation so users of it know what it does.

```
// This needs documentation. What is this function for? How is it used?
export function networkMonitor(graph, duration, failureCallback) {
  // ...
}
```

### Complex code should be commented

If you have named things well and the logic is still confusing, then it's time for a comment.

```
function leftPad(str, len, ch) {
  str = str + '';
  len = len - str.length;

  while (true) {
    // This needs a comment, why a bitwise and here?
    if (len & 1) pad += ch;
    // This needs a comment, why a bit shift here?
    len >>= 1;
    if (len) ch += ch;
    else break;
  }

  return pad + str;
}
```

## Side Effects

### Functions should be as pure as possible

```
// Global variable is referenced by the following function.
// If we had another function that used this name, now it'd be an array and it
// could break it. Instead it's better to pass in a name parameter
let name = 'Ryan McDermott';

function splitIntoFirstAndLastName() {
  name = name.split(' ');
}

splitIntoFirstAndLastName();
```

### I/O functions should have failure cases handled

Any function that does I/O should handle when something goes wrong

```
function getIngredientsFromFile() {
  const onFulfilled = buffer => {
    let lines = buffer.split('\n');
    return lines.map(line => <Ingredient ingredient={line} />);
  };

  // What about when this is rejected because of an error? What do we return?
  return readFile('./ingredients.txt').then(onFulfilled);
}
```

## Limits

### Null cases should be handled

If you have a list component for example, all is well and good if you display a nice beautiful table that shows all its data. Your users love it and you get a promotion! But what happens when no data comes back? What do you show in the null case? Your code should be resilient to every case that can occur. If there's something bad that can happen in your code, eventually it will happen.

```
class InventoryList {
  constructor(data) {
    this.data = data;
  }

  render() {
    return (
      <table>
        <caption>Inventory</caption>
        <thead>
           <tr>
            <th scope="col">ID</th>
```

```
            <th scope="col">Product</th>
           </tr>
         </thead>
         <tbody>
           // We should show something for the null case here if there's // nothing
           in the data inventory
           {Object.keys(this.data.inventory).map(itemId => (
             <tr key={i}>
               <td>{itemId}</td>

               <td>{this.state.inventory[itemId].product}</td>
             </tr>
           ))}
         </tbody>
       </table>
     );
   }
}
```

## Large cases should be handled

In the list above, what would happen if 10,000 items came back from the inventory? In that case, you need some form of pagination or infinite scroll. Be sure to always assess the potential edge cases in terms of volume, especially when it comes to UI programming.

## Singular cases should be handled

```
class MoneyDislay {
  constructor(amount) {
    this.amount = amount;
  }

  render() {
    // What happens if the user has 1 dollar? You can't say plural "dollars"
    return (
      <div className="fancy-class">
        You have {this.amount} dollars in your account
      </div>
    );
  }
}
```

## User input should be limited

Users can potentially input an unlimited amount of data to send to you. It's important to set limits if a function takes any kind of user data in.

```
router.route('/message').post((req, res) => {
  const message = req.body.content;
```

```
  // What happens if the message is many megabytes of data? Do we want to store
  // that in the database? We should set limits on the size.
  db.save(message);
});
```

### Functions should handle unexpected user input

Users will always surprise you with the data they give you. Don't expect that you will always get the right type of data or even any data in a request from a user. And don't rely on client-side validation alone

```
router.route('/transfer-money').post((req, res) => {
  const amount = req.body.amount;
  const from = user.id;
  const to = req.body.to;

  // What happens if we got a string instead of a number as our amount? This
  // function would fail
  transferMoney(from, to, amount);
});
```

## Security

Data security is the most important aspect of your application. If users can't trust you with their data, then you won't have a business. There are numerous different types of security exploits that can plague an app, depending on the particular language and runtime environment. Below is a very small and incomplete list of common security problems. Don't rely on this alone! Automate as much security review as you can on every commit, and perform routine security audits.

### XSS should not be possible

Cross-site scripting (XSS) is one of the largest vectors for security attacks on a web application. It occurs when you take user data and include it in your page without first properly sanitizing it. This can cause your site to execute source code from remote pages.

```
function getBadges() {
  let badge = document.getElementsByClassName('badge');
  let nameQueryParam = getQueryParams('name');

  /**
   * What if nameQueryParam was `<script>sendCookie(document.cookie)</script>`?
   * If that was the query param, a malicious user could lure a user to click a
   * link with that as the `name` query param, and have the user unknowingly
   * send their data to a bad actor.
```

```
   */
  badge.children[0].innerHTML = nameQueryParam;
}
```

### Personally Identifiable Information (PII) should not leak

You bear an enormous weight of responsibility every time you take in user data. If you leak data in URLs, in analytics tracking to third parties, or even expose data to employees that shouldn't have access, you greatly hurt your users and your business. Be careful with other people's lives!

```
router.route('/bank-user-info').get((req, res) => {
  const name = user.name;
  const id = user.id;
  const socialSecurityNumber = user.ssn;

  // There's no reason to send a socialSecurityNumber back in a query parameter
  // This would be exposed in the URL and potentially to any middleman on the
  // network watching internet traffic
  res.addToQueryParams({
    name,
    id,
    socialSecurityNumber
  });
});
```

## Performance

### Functions should use efficient algorithms and data structures

This is different for every particular case, but use your best judgment to see if there are any ways to improve the efficiency of a piece of code. Your users will thank you for the faster speeds!

```
// If mentions was a hash data structure, you wouldn't need to iterate through
// all mentions to find a user. You could simply return the presence of the
// user key in the mentions hash
function isUserMentionedInComments(mentions, user) {
  let mentioned = false;
  mentions.forEach(mention => {
    if (mention.user === user) {
      mentioned = true;
    }
  });

  return mentioned;
}
```

### Important actions should be logged

Logging helps give metrics about performance and insight into user behavior. Not every action needs to be logged, but decide with your team what makes sense to keep track of for data analytics. And be sure that no personally identifiable information is exposed!

```
router.route('/request-ride').post((req, res) => {
  const currentLocation = req.body.currentLocation;
  const destination = req.body.destination;

  requestRide(user, currentLocation, destination).then(result => {
    // We should log before and after this block to get a metric for how long
    // this task took, and potentially even what locations were involved in ride
    // ...
  });
});
```

## Testing

### New code should be tested

All new code should include a test, whether it fixes a bug, or is a new feature. If it's a bug fix, it should have a test proving that the bug is fixed. And if it's a new feature, then every component should be unit tested and there should be an integration test ensuring that the feature works with the rest of the system.

### Tests should actually test all of what the function does

```
function payEmployeeSalary(employeeId, amount, callback) {
  db.get('EMPLOYEES', employeeId)
    .then(user => {
      return sendMoney(user, amount);
    })
    .then(res => {
      if (callback) {
        callback(res);
      }

      return res;
    });
}

const callback = res => console.log('called', res);
const employee = createFakeEmployee('john jacob jingleheimer schmidt');
const result = payEmployeeSalary(employee.id, 1000, callback);
assert(result.status === enums.SUCCESS);
// What about the callback? That should be tested
```

### Tests should stress edge cases and limits of a function

```
function dateAddDays(dateTime, day) {
  // ...
}

let dateTime = '1/1/2017';
let date1 = dateAddDays(dateTime, 5);

assert(date1 === '1/6/2017');

// What happens if we add negative days?
// What happens if we add fractional days: 1.2, 8.7, etc.
// What happens if we add 1 billion days?
```

## Miscellaneous

> "*Everything can be filed under miscellaneous*"
>
> George Bernard Shaw

### TODO comments should be tracked

TODO comments are great for letting you and your fellow engineers know that something needs to be fixed later. Sometimes you gotta ship code and wait to fix it later. But eventually you'll have to clean it up! That's why you should track it and give a corresponding ID from your issue tracking system so you can schedule it and keep track of where the problem is in your codebase.

### Commit messages should be clear and accurately describe new code

We've all written commit messages like "Changed some crap", "damn it", "ugg one more to fix this stupid bug". These are funny and satisfying, but not helpful when you're up on a Saturday morning because you pushed code on a Friday night and can't figure out what the bad code was doing when you `git blame` the commit. Write commit messages that describe the code accurately, and include a ticket number from your issue tracking system if you have one. That will make searching through your commit log much easier.

### The code should do what it's supposed to do

This seems obvious, but most reviewers don't have the time or take the time to manually test every user-facing change. It's important to make sure the business logic of every change is as per design. It's easy to forget that when you're just looking for problems in the code!

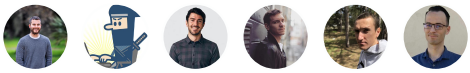
## Releases

No releases published

## Packages

No packages published

## Contributors 6



## Languages

- JavaScript 100.0%