CODING STANDARDS

Wildlife Drones and Flight Plans

Stakeholders

Matthew Evans
Reinhardt Eiselen
Brayan Jansen van Vuuren
Andreas Louw
Deane Roos

Team DR BAM
drbam301@gmail.com

Contents

Naming conventions

Things should be named by what they are* in a descriptive manner. Names should be as self-explanatory as possible.

*They should also be spelt correctly.

Variables

Variables should use camel casing with a lowercase first letter. They should never contain underscores. They should be plural when referring to an array, and singular otherwise.

Good: username, averageAge
Bad: user_name, AverageAge, curr, x, temp

Classes

Classes should be camel cased with an uppercase first letter.

They should never be plural

Good: User, UserManager
Bad: Users, user

Functions

Function names should be verbs describing what the function does.

Good: calculateDistance(), getAge()
Bad: doSomething(), my_fn()

Files

Files should use lowercase with hyphens. They should also adhere to the same naming convention as the rest of the files in the same folder.

Good: user.service.ts, user-manager.service.ts
Bad: User_service.ts

Object properties should be camelcase by default

Good:

```
const items = {
        something: 1,
        somethingElse: 2,
};
```

Bad:

```
const items = {
        Something: 1,
        SomethingElse: 2,
};
```

If properties need to be uppercase or contain spaces/special characters, then use quotation marks:

```
const items = {
        'Something': 1,
        'Authorisation-Header': '...',
        'A name': 'foo',
};
```

Refrain from using generic variable names

Good:
users.forEach(user => {});
Bad:
arr.forEach(el => {});

Types

Use types and declare interfaces whenever possible.

Refrain from using : any as the return type.

Dependencies

Unused dependencies should be removed

Imports

Do not use require. Instead use import * as.

Good:
import * as fs from 'fs';
Bad:
const fs = require('fs');

An exception is if the module will not work without using require (sometimes happens with older npm modules).

Use arrow functions where possible

Good:
users.forEach(user => {});
users.forEach((user, userIndex) => {});

Bad:
users.forEach(function(user) {});

Use map, filter, reduce or other array functions instead of for loops when possible

Use function chaining when available

Don't use for loops when dealing with arrays unless absolutely needed (e.g. if you're skipping multiple items at a time).

Good:

```
userNamesOver20 = users
 .filter(user => user.age > 20)
 .map(user => user.name);
```

More efficient (O(n) vs O(2n)), but may be less readable (the tradeoff is whether the extra performance is worth it):

```
userNamesOver20 = users
 .reduce((usersOver20, user) => {
   if (user.age > 20) {
     usersOver20.push(user.name);
   }
   return usersOver20;
 }, []);
```

Bad:

```
const userNamesOver20 = [];
for (let i = 0; i < users.length; i++) {
 const user = users[i];

 if (user.age > 20) {
   userNamesOver20.push(user);
 }
}
```

Refrain from using .concat inside of .reduce

Though this is arguably more readable, it will result in an efficiency of O(n^2) instead of O(n).

Good:

```
userNamesOver20 = users
 .reduce((usersOver20, user) => {
   if (user.age > 20) {
     usersOver20.push(user.name);
   }
   return usersOver20;
 }, []);
```

Bad:
```
userNamesOver20 = users
 .reduce((usersOver20, user) => {
   return usersOver20.concat(user.name);
 }, []);
```

Indentation

Keep code aligned.

If you have four or more indents in a function, think about creating a second function and calling it instead - it is very unlikely that you will ever have code this deep. Your code might also be very inefficient.

Use async await over .then

.then makes code much less readable. Use async await when possible.

Good:
```
async function findUsersOver20() {
 const users = await usersService.findAllUsers();
 return users.filter(user => user.age > 20);
}
```

Bad:
```
function findUsersOver20() {
 return new Promise((resolve, reject) => {
   usersService.findAllUsers()
     .then(users => {
       resolve(users.filter(user => user.age > 20))
     })
     .catch(reject);
 });
}
```

Even if a function only returns a promise, indicating that the function is async makes the code more readable. (It indicates very clearly that the code returns a promise).

Good:
```
async function getUsers() {
 return await usersService.findAllUsers();
}
```

Bad:
```
function getUsers() {
 return usersService.findAllUsers();
}
```

Use promises not callbacks

Whenever possible, convert callbacks to promises so it works with async await

Good:

```
async function getUsers() {
 const users = await new Promise(resolve =>
    request(`${URL}/users`, resolve)
 );
 ...
}
```

Bad:

```
function getUsers() {
 request(`${URL}/users`, users => {
   // now all code is stuck inside a nested function
   ...
 });
}
```

Use promise.all for multiple concurrent operations that need to finish before a task

Good:

```
async function ngOnInit() {
 await Promise.all([
   getUsers(),
   loadTasks(),
 ]);

 await associateUsersToTasks();
}
```

Bad:

```
async function ngOnInit() {
 await getUsers();
 await loadTasks();

 await associateUsersToTasks();
}
```

If a class isn't going to be re used, create it in the same file and don't export it

Say you have a service that uses a class. Don't create a separate file for that class if it isn't going to be used elsewhere. Just create it beneath the service and don't export it.

This depends on how long the file is, so it's up to your judgement.

Use semi colons

End lines with semicolons. JavaScript does allow endlines without semi-colons, but it enhances readability by showing where statements end - especially when you have multiple ending brackets));));

Use single quotes

‘string’ not “string”

Use template strings over concatenation

Good:
`${user.name} is online`
Bad:
user.name + ‘ is online’

Use let not var

Var works outside of block scope, and can lead to difficulty debugging. Use let instead when possible.

Use const

When a variable’s value does not change, use const.

This will often be the case when a variable points to an object (i.e. it’s a pointer). Very seldom does a pointer ever change value, hence const should be used to reduce confusion.

```
const user = {
 age: 20,
 height: 180,
};

user.age = 21;
```

Use commas at the end of every element in a multi-line array

Because of the way git handles line commits, if you add a single element to a multi-line array in code, then you are modifying two lines.

```
const users = [
 'Matt',
 'Andreas',
 'Deane'
];

const users = [
 'Matt',
 'Andreas',
 'Deane',
 'Reinhardt'
];
```

Whereas if we use commas at the end of every line

```
const users = [
 'Matt',
 'Andreas',
 'Deane',
];
```

Then only one line (the one added) is modified:

```
const users = [
 'Matt',
 'Andreas',
 'Deane',
 'Reinhardt'
];
```

Use spaces after commas

Good: const users = [‘Matt’, ‘Deane’, 2, 3, 4];
Bad: const users = [‘Matt’,‘Deane’,2,3,4];

Use spaces between operators and operands

Good: const age = 2 + 3 * 4 + (8 - getAge());
Bad: const age=2+3*4+(8-getAge());

Write tests while testing code

While testing code in the terminal (throwing random values at a function and expecting a result), write down those values and the results. Make those into a unit test.

Tip: you can run a single unit test from the CLI, making testing the function you are writing easier.

Don't leave trailing whitespace

Don't leave whitespace (tabs or spaces) at the end of lines.

If a line is blank and whitespace is being used for indentation, rather make the line blank.

Leave a blank line at the end of file

Leave a blank line at the end of every file. This will ensure that git doesn't count the newline \r\n as an addition.

Use spaces between braces and brackets in objects and destructuring

Good: import { UserController, UserManager } from '../user'
Bad: import {UserController, UserManager} from '../user'

Good: const [name, age] = getUser();
Bad: const [name, age] = getUser();

Use Object.keys to iterate over objects rather than for in

```
const countries = {
 'South Africa': {
   population: 54000000,
   gdp: 350000000000
 },
 'Zimbabwe': {
   population: 16500000,
   gdp: 17500000000
 },
};

// bad:
for (const countryName in countries) {
 const country = countries[countryName];
 console.log(countryName, country.population);
}

// good:
Object.keys(countries).forEach(countryName => {
 const country = countries[countryName];
 console.log(country);
})
```

This also provides the advantage of getting the key index without having to assign a variable and increment it at each iteration.

Don’t assign variables when returning immediately

Good:

```
function myFn() {
        ...
        return await something();
}
```

Bad:

```
function myFn() {
        ...
        const res = await something();
return res;
}
```

Code block braces should start on the same line

Good:

```
function myFn() {
        ...
}
```

Bad:

```
function myFN()
{
        ...
}
```