Caka, Amarildo
Nunes, Guilherme
Santos, Jefrey

CapScript: Strongly Typed ECMAScript subset for Capitual

2018

www.capitual.com

Caka, Amarildo <amarildo.caka@capitual.io>
Nunes, Guilherme <guilherme.nunes@capitual.io>
Santos, Jefrey <jefrey.santos@capitual.io>

CapScript: Strongly Typed ECMAScript subset for Capitual

2018

www.capitual.com

Abstract

JavaScript ecosystem is growing fast. The programming language that, in the past, had only been used for animated page effects and user input validation, nowadays powers entire websites, from personal blogs to entire banks. One of the language characteristics that can be considered responsible for its sudden acceptance is the ease of use. However, non-experienced programmers who adventure themselves on developing systems that they aren't actually able to develop are putting companies, clients and their own career on a high risk.

In this document, we have pointed some of the concerns that it's needed to stay aware on when working with JavaScript for business-grade applications. We also present companies that have switched their code bases to JavaScript and do not regret it. Finally, we will study Capitual's case, where the entire core-banking is developed using CapScript, a programming language developed internally that is based on JavaScript.

capitual bank

Table of Contents

Introduction

Although JavaScript's syntax may resemble C and C++ (such as with `if/else` structure), a huge difference felt by programmers who are used to develop using the former languages is that JavaScript has dynamic typing. (Flanagan, 2011)

JavaScript variables aren't required to be declared with a type, such as *integer* or *char*. They acquire a data type once they receive a value, but the data type can be modified easily simply by changing the variable value. Furthermore, JavaScript data typing is weak enough to convert types automatically on comparisons and operations.

JavaScript Specification's section 11.9.3, which states about the abstract equality operator (`==`), a comparison between a string and a number (e.g.: `"1"==1`) implies that the string gets converted to number as well.

It is possible to apply type-checking on a JavaScript system, despite its nature of ignoring type mismatching. And it is highly recommended. However, since it's not required to run a JavaScript system, it's easy not to follow these practice standards, and the reasons aren't few: short deadlines, team changes are mentionable reasons, however, since the code may actually *work* without type checking, even code reviewers may let this go unnoticed.

JavaScript dynamic checking is part of what makes JavaScript nice for prototyping and gives developers the speed and productivity they seek. Creating variables without having to worry about its type (at least in most of the times) allows developers to code faster, but if the required attention is not given, this feature can put the programmer on a trap coded by himself.

Finances need accuracy

In financial environment, accuracy is required. This is why, for example, float type is avoided: a floating-point problem may produce a difference in one or more calculations and this difference might result in losses to the application. That's why it is recommended to stay away from floating points, using either integer values only (in this case, storing cents instead of dollars, for instance) or, even better, BigDecimal, which is a library that had been ported to different platforms that is able to do calculations on decimal numbers represented as strings, on the same way a human would do with a paper and pen in hands.

All this accuracy makes us believe that a system that is programmed not to have errors in calculations, even in small margins like $10^{-8,}$ cannot chain procedures after receiving a `null` or a `false`, accepting it as ok since the expected value was `0`.

The set of problems that ignoring such situations is lengthy and directly proportional to the application scale. Bugs get harder to be found out, and an old known developers law says that the more a bug is delayed to be found, the harder it will be to be fixed.

However, such accuracy cannot be obtained quickly. And huge companies and bank tend not to fix what is not broken. If a solution still works, a banking company finds it not

practical to re-implement their entire set of services with new technologies, training their employees and clients to use the newly produced solutions. This is the reason why banks tend to work with the same software solutions that they produced years ago, working on mainframes, even during the cloud century.

Finances need reliability

One advantage that existing core-banking solutions employed by commercial banks, undeniably, is that some of them are running for decades, serving entire countries, with stability and too rare downtimes. Capital One is working with NodeJS for a good portion of its banking services, and recognized, through their software engineer Azat Mardan, that nowadays it's hard to find a business unit that is not using NodeJS. However, they made it clear that aren't going to replace legacy applications, where Java has been used: We invested a lot in Java in the last 10 to 20 years; we're not going to redo everything in Node.js. But with new projects, we are using Node.js. (NodeJS Foundation, 2011)

While Capital One has showed an interesting move towards adopting JavaScript on their internal business applications, their case also has revealed a strong reason why banking companies tend not to switch their banking solutions for new technologies. Throwing away their current solution means years of investment being lost. It's obviously expected that things change in a near future, in technologies sector, making it more interesting for huge companies to switch to cloud-based computing and new technologies.

The largest financial application running on NodeJS is, nowadays, PayPal.

Both Capital One and PayPal agreed that working with NodeJS added ease to the development process. The mentioned reason was the fact that the same programming language had been used both on the application backend and the frontend. Thus, instead of different teams writing documentations for each other, they could basically call a relatively small team of developers on the same room and they all *spoke* the same language. (Harrell, 2013)

NodeJS have good examples of relevant commercial applications running on its environment. On the group of finances, aside from PayPal, we can also mention Walmart, Groupon, Alipay and EBay. A filterless naming also brings Netflix, LinkedIn, Uber, Nasa, Mozilla and Microsoft, which are also NodeJS users who also help in developing the framework. (Collinsworth, 2017)

One could argument that, since it has been chosen by such companies, using JavaScript and NodeJS can be considered safe and less prone to errors. JavaScript's relatively short learning curve, when compared to languages that are largely used on enterprise-grade systems such as Cobol, Java and C/C++, and proved gain of productivity (Harrell, 2013) seems to endorse this affirmation.

Flanagan (2006), in the fifth version of his book JavaScript: The Definitive Guide, has made a comment that, although removed on the last version of his book, did not lose validity. JavaScript is not simple, he starts. Programmers who attempt to use JavaScript for non-trivial tasks often find the process frustrating if they do not have a solid understanding of the language.

JavaScript is, somewhat, forgiving to unsophisticated developers (Flanagan, 2006), and part of this forgiveness comes from its dynamic typing. It's easy to put online a fully working system with JavaScript. Actually, with just 3 lines of code, you have a full featured HTTP server to serve static files. It makes JavaScript attractive to developers who see in the language a simple (yet powerful) entry point. This, of course, does not mean that it is not possible to develop high quality, production-ready, enterprise-grade applications with JavaScript.

But developing reliable systems is not as simple to learn. It takes several years of experience in projects with exemplary code bases, in well trained teams, to deeply understand the development process of softwares that are going to be accepted by businesses.

Type checking as a way to avoid software bugs

Having said that JavaScript's dynamic typing is an attractive to unsophisticated developers, one may understand that dynamic typing is a feature for unsophisticated languages. Therefore, it's worthy to make it clear that this is not the case. Dynamic typing is powerful, but it has to be used consciously.

The present is also not trying to state, also, that type checking is able to make code totally safe.

About 15% of software bugs could be discovered before runtime if type checkers were used (BARR; BIRD; GAO, 2017). Also, typing sums up verbosity to the code, which makes it clearer for a possible reviewer or maintainer how the code works. Complex errors handling are also avoided, since when using dynamic typing, hours of debugging are spent just to verify if a variable that should receive an integer value is, in fact, receiving an integer value. (HAYAT et al, 2017)

Type checkers often run on compile time, but JavaScript is not a compiled language. A JavaScript type checker had to run, therefore, by transpilating the source code to native JavaScript.

Native JavaScript does not need to be converted to bytecode as well. Thus, in order to add static type checking to JavaScript, some kind of transpilation would be needed, since during the transpilation, the type checking occurs and native wrappers for the type checking are written to the output built file.

Static type-checking is also not enough to detect any kind of type mismatch issues at build time. Softwares tend to get fed up through different external methods. If you are going to need an user input, a CSV file or a HTTP request result, since type checkers do not run the script, these issues will not be detected by them. In this case, runtime type checking is made needed.

JavaScript data types

JavaScript recognizes six data types (W3Schools):

Numbers (`0`, `1`, `1e2`, `1.5`, `1e-9`)

JavaScript does not make difference between integers and decimals. They are all stored as floating-point values, following the IEEE 754 standard. It allows JavaScript to represent numbers from $\pm 5 \times 10^{-324}$ to $\pm 1.7976931348623157 \times 10^{308}$.

JavaScript numbers can be presented as:

- Integer values (`0`, `1`)
- Decimal values (`.5`, `1.2`)
- Scientific notation (`10e9`, `10e-9`)
- Built-in constants (`Math.PI`, `Math.E`)

- **Strings (`"Hello, world"`)**
  JavaScript strings are wrapped by single (`'`) or double quotes (`"`). The language stores strings as a sequence of 16-bit Unicode values.
  An interesting difference between JavaScript and other languages is that in JavaScript, strings are immutable. While in C/C++ developers are able to modify, for instance, the second string character simply by editing `variable_name[1]` (since it starts at 0), in JavaScript this is not possible.
  Therefore, if one finds it needed to modify internal characters of a string, ends up redefining the string value. Of course it's possible to use the `.substr()` method to capture parts of the string that will remain.

- **Object (`{foo: "bar"}`, `[1, 2, "foo"]`)**
  Perhaps the more important data type on an object-oriented programming language, objects are largely used to store different types of data. Their structure is similar to associative arrays. Whether using built-in structures (such as with the `Date()` implementation or regular expressions), or structures defined by the developer (through JSON), nearly any software developed using JavaScript will make use of objects.

  Objects' properties are mutables and always passed as references. This means that when one function changes an object property, the change is valid globally.

  Arrays are special objects that store ordered collections of values. These values can have any JavaScript supported type (including other arrays). JavaScript's standard array implementation does not support associative arrays (these are plain objects).

  A controversial fact about JavaScript is that `null` is also an object.

  A note about JSON

  JSON (JavaScript Object Notation) as we know derived from JavaScript objects. It is, basically, how to write objects in JavaScript. These objects contain properties which types can be any of JavaScript's supported data types (including another object).

  Although nowadays it's a public format, being easy for machines and humans to read and write made it largely used by several programming languages (JSON, 2018). Nowadays, it's easy to see JSON files used for defining or storing user settings, and the JSON format is largely used in web services and APIs. Every time one runs a social network or banking application on a smartphone, there is a huge chance that the smartphone is talking to the servers through JSON.

- **Boolean (`true`, `false`)**

  Boolean values store the simplest programming definitions.

  In comparisons using abstract equality operator (`==`) with numbers, boolean values get converted to numbers (ECMA, 2011). As expected, `Number(false) == 0` and `Number(true) == 1`.

Functions (`function fn() {}`, `function() {`, `() => {}`)

In JavaScript, functions can be created through **definition** or **declaration**.

When functions are created through definition, the interpreter reads them before executing the code's main entry point, so the functions are available on their scope as soon as the interpreter starts executing from the entry point.

On the other hand, functions created through declaration are commonly used as anonymous (unnamed) functions, but they can be named as well. In many cases, these functions are used directly into callbacks, but they can be associated to variables, object properties or arrays items.

Undefined (`undefined`)

Any undeclared variable or function is `undefined`. Deleted object properties also become `undefined`.

Checking if the type of a variable is undefined allows the programmer to verify if a variable or function was already declared or is present on a remote API response.

An overview on JavaScript type checking methods

Elliot (2016) mentions that although JavaScript lacks static type checking, it does not mean that no type checking should be done. The interpreter does dynamic type checking at runtime, but it is very forgiving and only throw errors on extreme situations, such as invoking `undefined` as a function.
The most basic type checking method consists in using `typeof` to check if variables comes on the expected data type and interrupt the workflow if this condition was not met.

JavaScript

```
if (typeof variable !== 'string')
   throw new Error('variable should be string')
```

However, one can argue that this is too verbose. In fact, doing so for every used variable looks like a nightmare, and it's possible to end up with more codes for variable checking than for producing the function effect itself.

For this reason, some developers have created libraries or JavaScript subsets that gifts developers an easier way for type checking their variables.

Facebook and Microsoft are showing movement towards JavaScript type checking. The social network team keeps a JavaScript type-checking tool named Flow. It is developed using OCalm and runs static type checking on build time.

Meanwhile the Operating Systems giant claims to have developed their own JavaScript superset, presenting their creation as a new programming language. They are proud on saying that the TypeScript compiler is written in TypeScript.

While TypeScript must be transpilled to JavaScript before deploying to production, Flow's checker is an external binary that must be run in order to check the variable types. (Kyle, 2017)

CapScript

While the solutions mentioned on the previous chapter are interesting moves from these companies and the open source community, at Capitual it was found to be needed to develop our own subset of JavaScript, to be used both on the core-banking and in Capitual interfaces (initially these are WebSockets, REST API and IMTP Protocol, although interfaces does not depend on the core-banking and can be added, removed or modified with no side-effects).

Keeping our own JavaScript subset, named CapScript, allowed us to stay always one step further, since we are able not only to create new language features, but also to adopt new implementation proposals sent to JavaScript's TC39 (Ecma International, Technical Committee 39). At the end, even if these proposals don't get accepted by ECMA to integrate on the native JavaScript, we are still able to define whether CapScript will remain supporting the feature or also drop it.

Other mentionable benefits that we've been able to register by keeping a JavaScript subset were:

- Easier adoption of coding standards
- Code optimization at transpile time
- Language modification for new needs
- Future-proof code base
- Improved security due to transpile-time checks

It's all about JavaScript - a more robust one

CapScript gets transpilled to native JavaScript on runtime. This is done using Babel. Using Babel, we can create plugins with features that CapScript is going to support, in order to distribute it to developers who are working on Capitual's code base. Once they learn what they can do with CapScript, the code base gets more consistent.

Furthermore, no native JavaScript features had been removed. It allows us to keep our code base up to the date not only by new CapScript features, but also with JavaScript updates.

Developing using Babel allows us to create the CapScript features with JavaScript, through developing Babel plugins. Capitual developers make use of `babel-node`, a Babel interpreter that runs on the fly, but it is not suitable for production.

When a new Capitual version is ready to go live, before deploying to our production servers, the entire codebase gets transpilled by Babel to JavaScript. All the CapScript features are ported to JavaScript, through native functions and snippets that are added over the code body, when needed.

With Babel we can also add transpile-time code optimizations, reducing the servers latency, giving users quicker response times.

CapScript is a mix of the best of different worlds: while it maintains up-to-date JavaScript features (as we try to keep Capitual's code as updated and future-proof as possible by looking at newly released NodeJS and ECMAScript stable versions and look at our entire code base, looking for portions that can be rewritten at a more concise way with the new language features), it also keeps looking at interesting ECMAScript proposals and implements these before JavaScript itself (even though some of these proposals aren't going to be accepted by ECMA; in this case, we take their reason into consideration in order to decide whether CapScript will remain supporting that feature). But what makes CapScript totally suitable for Capitual development is the fact that the ideas that get implemented officially in CapScript comes from the daily needs at Capitual labs.

One may ask whether these features should be sent to Babel or even to ECMA, as proposals. For this, Capitual team decided that CapScript features shall not be part of ECMAScript.

JavaScript is a neutral language, which allows developers to create and maintain different kinds of systems. CapScript is specifically business-related. At Capitual, as it was revealed on this document, developers believe that specifying variable types makes the code more robust and less prone to production errors. We believe using CapScript is ideal for business-related projects, but it's needed to keep in mind that JavaScript shall also work for simple tasks. Also, since even the data types aren't standard, but specially fit for our businesses needs, and data typing requires a wider knowledge from the developer (in other words, CapScript is not as forgiving as JavaScript is), we don't think CapScript features could fit standard JavaScript in any way.

Which features do CapScript support?

Capitual's core-banking code base is entirely written using CapScript. A developer, at first sight, is able to notice that CapScript is, in fact, JavaScript, but with important modifications on its syntax.

In this chapter, we list some of the syntax modifications that CapScript had. We will not focus on code optimization techniques, since these are run at transpile-time and produce almost unreadable and minified code.

Strong typing

CapScript's typing is based on TypeScript and Flow, on its syntax. It supports custom types, which allows developers to have `name` and `wallets` types that resolve to `string` and `object`respectively, instead of using `string` and `object` directly. It improves code readability and reduces exposition to risk of production errors of type mismatching. Custom types are declared as:

JavaScript

```
typeset name = string
typeset wallets = object
```

These can be exported as ES6 modules, so it's possible to maintain different custom types on an external file that gets imported by every file that is going to make use of these types. For example:

```JavaScript
export typeset name = string
export typeset wallets = object
```

An usage example could be:

```JavaScript
import {name, wallets} from './types'

let userName : name = "John Doe"
let userWallets : wallets = {}
```

However, CapScript native types are more strict than standard JavaScript types. By importing types from a forked version of `check-types` package, it can verify not only if a variable is a string, but also if the string is not empty. We can make it more concise by writting:

```JavaScript
export typeset name = nonEmptyString
export typeset wallets = nonEmptyObject
```

Obviously we can get back to native dynamic type checking at all. In CapScript allows it to be done in three different ways:

```JavaScript
// by defining type as "any"
let userBio : any

// by defining type as "variant"
let userBio : variant

// or by no defining any type at all
let userBio
```

If needed, we can also use *maybe-types*. Defining a variable as *maybe-type* implies that it can be `undefined` or `null`, or have value according to the specified type. For instance:

```JavaScript
let optionalOption : maybe.string = null // ok
optionalOption = "abc" // ok
optionalOption = 123 // TypeError
```

Fully ES6, ES7 and ES8 compliant

Although browsers and NodeJS does not fully support ES7 as of the time of the writting, CapScript stays always one step further.

When Capitual's core banking was developed, much of the JavaScript features used weren't supported natively by NodeJS interpreter. Interestingly, some of these features were too often used.

Below, we have listed a few features that CapScript had already support to, at the time of the core-banking development. We have preffered to keep the examples as native JavaScript for an easier understanding.

Classes

It's possible to create and extend classes, and declare a `constructor` method that gets called automatically once the class is instanced with `new`.

JavaScript
```
class HelloUser {
    constructor(userName) {
        this.userName = userName
    }
    sayHello() {
        return `Hello, ${this.userName}!`
    }
}

let greeting = new HelloUser("John")
greeting.sayHello() // Hello, John!
```

Constants and scoped variables (`let`)

It's possible to declare variables that aren't allowed to be modified.

JavaScript
```
const APItoken = 'XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX'
APItoken = 'foo' // TypeError: Assignment to constant variable.
```

It's also possible to declare variables that are only valid on their scope.

JavaScript
```
function test() {
    let foo = 'bar'
}

console.log(foo) // ReferenceError: foo is not defined
```

Arrow functions (`() => {}`)

Arrow functions are a shorter way to declare functions. An interesting fact is that it does not have it own scope.

JavaScript
```
fetch('https://someapi.com/read.json').then((res) => {
    console.log(res)
})
```

Default function argument values

By default, ES5 did not support default argument values on the functions. It had to be done so:

JavaScript
```
function showGreeting(user) {
  user = user || "User"
```

```
  return "Hello, "+user
}
```

With ES6 and CapScript, it can now be done as:

JavaScript
```
function showGreeting(user = "User") {
  return "Hello, "+user
}
```

Variable destructuring

Destructuring allows obtaining, on an easier way, only the needed data from objects. For example:

JavaScript
```
let userData = {
  name: 'John',
  bio: 'Lorem ipsum dolor sit amet'
}

let { nome } = user

console.log(nome) // John
```

It can also be done on the function arguments. Capital Development Standard states that any function with 3 or more arguments should use argument destructuring.

JavaScript
```
function greetUser({user = "John", isInBirthday}) {
    return "Hello, "+user+"!" + (isInBirthday ? " Happy birthday!" : "")
}
```

Mainly for big functions, it's a more pleasant task to call destructured arguments-functions, since one is not lost with the argument order:

JavaScript
```
greetUser({
    user: "Jane",
    isInBirthday: true
}) // Hello, Jane! Happy birthday!
```

Rest Operator

Note that this is not about the REST API interface.

Rest operators offer an easy way to copy array contents from one to another. Example:

JavaScript
```
let mostRecentUsersIDs = [1, 2, 3, 4, 5];
```

```
let [first, second, ...remaining] = mostRecentUsersIDs;

console.log(first); // 1
console.log(second); // 2
console.log(remaining); // [3, 4, 5]
```

Spread

Spread operator is similar to the rest operator, but it copies key & value pairs from one object to another. Example:

JavaScript
```
let copySettingsFromThisGroup = { group: 1, name: "Foo", privacy: "public" }

let newGroup = { ...copySettingsFromThisGroup, group: 2 }

console.log(group) // {group: 2, name: "Foo", privacy: "public"}
```

Template literals

It's also possible to include JavaScript variables inside strings without concatenation.

This has already been showed on a previous example:

JavaScript
```
sayHello() {
    return `Hello, ${this.userName}!`
}
```

Through string concatenation, it should be written so:

JavaScript
```
sayHello() {
    return "Hello, "+${this.userName}+"!"
}
```

Short property notation

If one wants to insert a property on an object which key is the same name as the variable that stores its value, probably it would be done so:

JavaScript
```
var foo = 'bar'
var obj = { foo: foo } // { foo: "bar" }
```

With short property notation, the redundancy is avoided:

JavaScript
```
var foo = 'bar'
```

```
var obj = { foo } // { foo: "bar" }
```

Import/export

Importing and exporting variables, classes and functions is made easy, and populating `module.export` is not needed (although it is done behind the walls):

JavaScript
```
export name = "John"
export surname = "Doe"
export getFullName(name, surname) {
    return name + " " + surname
}
```

JavaScript
```
import {name, surname} as userdata from './userdata'
import getFullName from './userdata'

console.log(getFullName(userdata.name, userdata.surname)) // "John Doe"
```

Promises

A commonly complained JavaScript problem is the named callback hell.

Since JavaScript is an asynchronous language, it does not wait for tasks to be done. If a task takes long to finish, the code continues its flow. It can lead problems, since a function that did not finish running yet cannot return its value.

JavaScript
```
let response = asyncHTTPRequest('https://site.com')
console.log(response) // null
```

Because of this, it's common for JavaScript functions to receive, as one of the arguments, a callback function (usually anonymous, but not mandatory) that gets called, receiving the function execution result as argument.

JavaScript
```
asyncHTTPRequest('https://site.com', function(result) {
    console.log(result) // site.com content
})
```

However, once your flow needs to have multiple callbacks, your code gets pushed to right thanks to the indentation. Here comes the callback hell.

JavaScript
```
asyncHTTPRequest('https://site.com/api/gettoken', function(token) {
    asyncHTTPRequest('https://site.com/api/getprofiledata?token='token,
function(result) {

asyncHTTPRequest('https://site.com/api/getuserwallets?token='+token+'&id="res
ult.id', function(wallets) {
            wallets.map(function(wallet) {
```

```
asyncHTTPRequest('https://site.com/api/getwalletbalance?token='+token+'&walle
t='+wallet, function(balance) {
                    // :(
                })
            })
        }
    }
})
```

Promises allows defining callbacks in a linear way, avoiding the push right.

JavaScript

```
new Promise((resolve, reject) => {
    asyncHTTPRequest('https://site.com/api/gettoken', resolve)
}).then(token) => {
    return new Promise((resolve, reject) => {
        asyncHTTPRequest('https://site.com/api/getprofiledata?token='token,
resolve)
    })
}).then((result) => {
    return new Promise((resolve, reject) => {

asyncHTTPRequest('https://site.com/api/getuserwallets?token='+token+'&id="res
ult.id', resolve)
    })
}).then((wallets) => {
    wallets.map(function(wallet) {

asyncHTTPRequest('https://site.com/api/getwalletbalance?token='+token+'&walle
t='+wallet, balance => {
            // :)
        })
    })
}).catch((error) => {
    console.log("Error", error)
})
```

Async/await

Async/await are new ways to represent promises that makes the code even more readable.

By writting await before invoking an asynchronous function, hypothetically the JavaScript interpreter will wait for the function result before continuing the code flow.

Under the hoods, what happens is that asynchronous functions actually are returning promises (the interpreter understands so). When the function returns a value, the interpreter understands it as a promise resolve. If the function throws some value, the interpreter understands it as a promise rejection. Therefore, instead of `.then()`/`.catch()`, the developer will use `try`/`catch`.

The code from the previous example gets a lot of readability improvements when using this feature. Of course the pseudo-function `asyncHTTPRequest` must be modified not to accept a callback, but use a return instead. It's easy to hack the function to accomplish it:

JavaScript
```
function HTTPRequest(url) {
    return new Promise((resolve, reject) => {
        asyncHTTPRequest(url, resolve)
    })
}
```

And here is the usage:

JavaScript
```
async function main() {
    try {
        let token = await HTTPRequest('https://site.com/api/gettoken')
        let result = await
HTTPRequest('https://site.com/api/getprofiledata?token='token)
        let wallets = await
HTTPRequest('https://site.com/api/getuserwallets?token='+token+'&id="result.i
d')
        wallets.map(wallet => {
            let balance = await
HTTPRequest('https://site.com/api/getwalletbalance?token='+token+'&wallet='+w
allet)
            // :)
        })
    } catch(error) {
        console.log("Error", error)
    }
}

main()
```

Adopted TC39 Proposals

These are suggestions for integration with JavaScript that were sent to ECMA and are being tracked on an official repository, but that have been adopted by CapScript, which already supports them.

BigInt and BigDecimal

Computers processors have two interesting problems about calculating with numbers: for not using a decimal base like humans learn since preschool, they use a binary base that is made exclusively by two numbers: 0 and 1.

However, a computer must be able to understand and show information using decimal base, even if, under the hoods, binary base is used. Although computers are able to do it pretty well, it is still very limited: huge numbers cannot be understood because nowadays' processors are unable to convert them to their binary base. This is called integer overflow.

At the same way, decimals aren't properly calculated due to the nature of numbers representation and this can lead to a lot of problems. This is called floating point problem.

Because of this, and mainly for working with finances, CapScript calculates amounts using strings. When math operations are needed, under the hoods, instead of asking the processor directly to calculate the complete numbers, the operation is separated and calculated by parts, from the lowest to the highest classes (starting with unities, then with the tens, then with the hundreds and so on).

Divisions and calculations with decimal numbers are also done on the same way that humans do manually. This leaves the processor with simple calculations like 1+2, instead of complicated calculations that may lead to bugs, like 0.1+0.2 (which the processor wrongly answers as 0.30000000000000004).

String.prototype.trimStart / String.prototype.trimEnd

As mentioned before, JavaScript strings aren't mutable.

These functions allow removing characters from the beginning or ending of the strings.

```JavaScript
String("ABCDE").trimStart(1).trimEnd(2) // BC
```

Throw expressions

This feature allows throwing from more contexts.

A function or method throws a value usually when something goes wrong. However, it's only possible to throw at a standalone statement.

```JavaScript
function getUserName(user_id) {
    if (typeof user_id == 'undefined')
        throw new Error('Missing user_id')

    // function code
}
```

This feature allows snippets like the above to be reduced to:

```JavaScript
function getUserName(user_id = throw new Error('Missing user_id') {
    // function code
}
```

Note that if `user_id` is not provided during the function call, the default value will be used. And the default value throws an error.

Math extensions

A set of improvements is done over the native Math object.

Math.clamp

This method grants that a value is inside a range, converting it to the nearest range member if it is not on the range.

For example, if we have to grant that the variable `x` is inside 1 and 5, we do:

```
JavaScript
Math.clamp(x, 1, 5)
```

In this case, if `x=10` it will become `5`. On the same way, if `x=-8`, it will become `1`.

Math.scale

It converts a value from a scale to another through the rule of three.

For instance, given that Celsius scale goes from 0 to 100, and Fahrenheit goes from 32 to 212 on the same values, we can convert 36º C to Fahrenheit by doing simply:

```
JavaScript
scale(36, 0, 100, 32, 212) // 96.8
```

Math.radians and Math.degrees

Converts between radians and degrees.

It also implies the existence of two constants: `Math.DEG_PER_RAD` = π180π180 and `Math.RAD_PER_DEG` = 180π180π.

String.replaceAll

If one tries to replace a string inside a substring with JavaScript, could do:

```
JavaScript
String("Hello, user!").replace("user", "John") // Hello, John!
```

However the replace only happens once. If there are multiple occurrences of the substring on the original string, only the first occurrence will be replaced:

```
JavaScript
String("Hello, user! How are you, user?").replace("user", "Jane") // Hello,
Jane! How are you, user?
```

In order to fix it, the developer must use regular expressions, with the global (`/g`) modifier:

```
JavaScript
String("Hello, user! How are you, user?").replace(/user/g, "Jane") // Hello,
Jane! How are you, Jane?
```

Or, with CapScript, just use `.replaceAll`:

```
JavaScript
```

```
String("Hello, user! How are you, user?").replaceAll("user", "Jane") //
Hello, Jane! How are you, Jane?
```

Slice notations

Considering that JavaScript arrays start counting at 0 and not at 1, if one wants to get a range of items from inside an array or string, must do:

JavaScript

```
let array = ['A', 'B', 'C', 'D', 'E']
array.slice(1, 3) // ['B', 'C']
```

This modification allows Golang-like slice notations:

JavaScript

```
array.slice[1:3] // ['B', 'C']
```

Conclusion

Developing and keeping a programming language is not an easy task. It takes time, lots of case studies with other languages and, mainly, experience, which is essential to know the real need of a business.

JavaScript ecosystem is growing fast. Its ease for learning and accessibility allows developers to start building amazing applications at light speed. Universities like Stanford are already announcing their migrations to JavaScript for their classes, instead of the very practiced Java, C/C++ or Pascal (Claburn, 2017).

However, it's clear that JavaScript specifications are very forgiving. While this is great for a huge set of applications (perhaps for most of the applications developed using JavaScript), for business and, mainly, banking companies, this forgiveness may be the source of huge financial losses due to unseen software bugs.

With CapScript, we have achieved the freedom of making our adopted programming language to work on the way we need. And choosing keeping it as a subset of JavaScript instead of writing a new language from scratch allowed us to adopt new features and improvements that the original JavaScript receives. Since this language is getting a great attention now, it's obviously needed to keep track of every news on the JavaScript environment.

What is more important for a banking company, we have improved our system's security by adopting CapScript, and we were also able to optimize even more our code base.

CapScript will follow receiving updates constantly and our code base will get improved over time. Its mission is to stay always one step further, maintaining a future-proof code base, allowing our developers to deploy high quality services. CapScript is responsible for a substantial portion of what makes Capitual unique, fast, responsive and loved by whoever uses it.

References

"Capital One Case Study." NodeJS Foundation, Sept. 2017, foundation.nodejs.org/wp-content/uploads/sites/50/2017/09/Node.jsCapitalOneFINAL_casestudy.pdf.

"ECMAScript Language Specification." ECMA International, June 2011, www.ecma-international.org/ecma-262/5.1/.

"Introducing JSON." JSON, 19 Mar. 2018, json.org/.

"JavaScript Data Types." W3Schools, 14 May 2018, www.w3schools.com/js/js_datatypes.asp.

Barr, Earl T.; Bird, Christian; Gao, Zheng. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript. 2017.

Claburn, Thomas. "Stanford Uni's Intro to CompSci Course Adopts JavaScript, Bins Java." The Register, Software, 24 Apr. 2017, www.theregister.co.uk/2017/04/24/stanfordtestsJavaScriptinplaceofjava/.

Collinsworth, Troy. "Node.js: Do More with Less." LinkedIn, 15 Nov. 2017, www.linkedin.com/pulse/why-majority-embrace-nodejs-troy-collinsworth/.

Elliott, Eric. "You Might Not Need TypeScript (or Static Types)." Medium, JavaScript Scene, 5 Dec. 2016, medium.com/JavaScript-scene/you-might-not-need-typescript-or-static-types-aa7cb670a77b.

Flanagan, David. JavaScript: the Definitive Guide. 5th ed., O'Reilly, 2007.

Flanagan, David. JavaScript: the Definitive Guide. 6th ed., O'Reilly, 2011.

Harrell, Jeff. "Node.js at PayPal." PayPal Engineering Blog, PayPal, 22 Nov. 2013, www.paypal-engineering.com/2013/11/22/node-js-at-paypal/.

Hayat, Sharmeen, et al. "Strict Types: Typescript, Flow, JavaScript - to Be or Not to Be?" Medium, Codeburst, 20 Dec. 2017, codeburst.io/strict-types-typescript-flow-JavaScript-to-be-or-not-to-be-959d2d20c007.

Kyle, Jamie, et al. "Usage." Flow, Facebook, Inc., 5 Apr. 2017, flow.org/en/docs/usage/.