**CODING STANDARDS FOR ALL DEVELOPMENT ACTIVITIES USING NODE.JS WITHIN CELIGO PRODUCT ENGINEERING**

**Version Log**

| **Date** | **Version** | **Owner** | **Change** | **Status** |
| --- | --- | --- | --- | --- |
| 01/09/2023 | 1.0 | Umakanth G | Initial Version | In Progress |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

**Review Log**

| **Date** | **Reviewers** | **Comments** |
| --- | --- | --- |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**TABLE OF CONTENTS**

[**OBJECTIVE 2**](#_nz000huzai3p)

[**AUDIENCE 3**](#_hxg5stcbh540)

[**EDITOR AND OTHER TOOLINGS 3**](#_qgsvc5ewpm9p)

[**CODING STYLE AND STANDARDS 3**](#_10twnhia8kj8)

[General Celigo guidelines for JS coding 3](#_m99hlu866t19)

[Capture error format, common patterns styles etc 3](#_zhwsfqem2irg)

[Finding existing code and reusing it 4](#_dwemnmrpahw5)

[**CODE DESIGN 4**](#_80lfzls1q0sc)

[**UNIT TESTING 4**](#_ga3pf5b7mnww)

[**CODE REVIEW 5**](#_nm1vbuyo5x46)

[**DOCUMENTATION 6**](#_hw6v4xz0b5x6)

# **OBJECTIVE**

This document is intended to capture the standards to be followed while developing or reviewing our product using node.js technology within our Product Engineering organization. This standards document has been created after understanding various best practices across the industry and adapting to our team’s needs. It is essential to have a consistent, standardized way across the entire team to organize and write code using node.js to improve readability and thereby its maintainability.

# **AUDIENCE**

Intended for all Product Engineering team members that will be using node.js in their development activity.

# **EDITOR AND OTHER TOOLINGS**

#TODO : Add recommended editors and how should they be set-up

# **CODING STYLE AND STANDARDS**

### General Celigo guidelines for JS coding

* Install StandardJS
* Install SonarLint
* Follow strict coding guidelines. [Recommended guide](https://github.com/celigo/javascript) This based on NodeJS Coding best practices followed in Industry.

### Capture error format, common patterns styles etc

* Each error should be in a canonical format. Eg: [Github](https://github.com/celigo/integrator-adaptor/blob/master/errorStore.js#L38)

[](https://github.com/celigo/integrator-adaptor/blob/master/errorStore.js#L38)

{  
 code: { type:string },  
 message: { type: string },  
 [source](https://github.com/celigo/integrator-adaptor/blob/master/util/errorUtil.js#L88): { type: string, enum: [] }  
}

[](https://github.com/celigo/integrator-adaptor/blob/master/errorStore.js#L38)

For Adaptor Errors, 2 new properties are added.

{

occurredAt: { type: number },

resolved: { type: Boolean }

}



* Do not return errors directly if possible.
* Add them to the error store and return the error from the [error store](https://github.com/celigo/integrator-adaptor/blob/master/errorStore.js).
* Do not return technical errors to a user.
  + Eg: websocket connection lost.
  + Return generic error messages. Eg: ‘internal server error’, ‘something went wrong’ with a reference-id.

### Finding existing code and reusing it

* If you’re trying to implement anything new. Do try to look for patterns used at other places. Eg: uploading to S3, object manipulations ..etc.
* Check unit test cases, utils on what you’re trying to do.

# **CODE DESIGN**

* Follow SOLID principles while writing code. <https://www.xenonstack.com/blog/solid-principles-javascript>
* There shouldn’t be duplicate code. Use functions/move them to utils. If there are already utils for these, make sure they’re used properly. Follow [DRY](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself) principle.
* Make sure that your code is testable by injecting dependencies rather than creating dependencies in the code. <https://tsh.io/blog/dependency-injection-in-node-js/>
* Make sure that there are proper null checks.
* Do not overcomplicate code. Keep it simple and readable.
* Make sure we are not logging unnecessary info to splunk, especially loops that run on customer data.
* Any logs to splunk should be precise. **Should not have User data**. Should always have a max depth for objects. Eg: *require(‘util’).inspect(<obj>, null, { depth: 4 })*
* No sensitive information should be committed to github and no sensitive information should be logged to splunk.
  + Eg: passwords, tokens, PII Data.. etc
* If there is a change in contract, make sure that this is well thought through.
* Code should always be backward compatible. If it is not, we’d need ways to write migration scripts/make sure that the use case is not used by our customers.
* Make sure that your code works during deployments. i.e, some of the instances might run on the old code and some may run on the new code. Your code should work in both of these combinations. ( For Canary and Rolling Upgrades)
* If you’re adding a new module, make sure that the module is well supported, maintained, well downloaded and doesn’t have any major bugs. It should have a valid license. Do not use carat version, use a specific version (eg: use 1.0.1 instead of ^1.0.0)
* Have proper unit test cases for each of the code paths(success/failure paths). Eg: If else loop, switch blocks, entering and without entering loops etc.
* Make sure that package-lock.json is updated correctly. i.e, use npm update integrator-adaptor instead.
* If you’re resolving merge conflicts, make sure that the conflicts are resolved correctly. Do make sure no test cases fail because of this change.
* Remove commented code from the files.
* Commit with proper commit messages(Avoid small changes..etc). If possible, squash all the commits into a single commit. But don’t make it too big.
  + The following is recommended but not mandatory
    - 5 dev Files max.
    - No limit on the number of test files.
    - 200 dev lines change max.
    - 500 test lines changes max.
* Always check Travis for failed unit tests. Make sure there are no failures due to your code changes.
* PR Should should have the PR Checklist <https://celigo.atlassian.net/wiki/spaces/PE/pages/edit-v2/570359883>
* Setup a meeting for the code review kickoff for big reviews.
  + Walk through code and changes.
* Reviewers should Read EPIC, FRD and Design and clarify questions in the initial meeting.

# **UNIT TESTING**

* Try to add unit test cases for every code path that has been added.
* Make sure that any records created via unit test cases in any adaptors(DH, NS, etc) are deleted after the test case is done.
* Do not link one block's outcome to another. Each block should be able to run independently.
* Do not write test cases that are long running. Try to mock the responses/use VCR whenever possible.
* Assert the exact response. Do not skip assertions within tests.
* If you’re writing a test case that is retrying/waiting for a timeout. Use env variables to reduce these waiting timeouts to a lower limit.
* Do not update global variables like test.user/ licenses related to this.
* Try to make sure that test cases can run parallely. i.e, multiple jobs of the same test case file should run parallely.
* If the Unit test case uses any credentials make sure that they are externalized and only test accounts are used.
* Tests should be written for minimal units like function or code block. Don’t write integration tests as unit tests.
* No functionality should be exposed just for the unit testing purposes(i.e. Exposing functions/methods as public just so that tests can be written on them) - use [rewire](https://www.npmjs.com/package/rewire) like modules instead.

# **DOCUMENTATION**

* API Documentation should be part of the code and should be able to generate shareable documents from the code.
* Shouldn’t unnecessary add comments in the code. The code should itself be readable.

# **CODE REVIEW**

## Understand the Scope and the Functionality:

The owner of the tracker, primary reviewer and the secondary reviewer should have a clear understanding of the change expected as part of the story / epic. After getting the clarity, try to think of the impact that it could have on the existing functionality and check if the changes are backward compatible. (All this should actually be covered in the story pointing meeting, but sometimes the tracker could be assigned to some dev who's not present in the meeting). If the scope isn’t clear, try to pull in whoever is required to get the clarity on the scope or create a spike tracker and work on the spike if none have the clarity on scope.

Once the scope is clear: try to list down all the different unit tests that need to pass once the changes are done and discuss them with the primary reviewer.

## How to Review:

* Firstly, the reviewer should try to get a high level understanding of the changes being done in the PR.
  + **For Ex:-** If the PR being reviewed is about addition of a new route, then the reviewer should try to understand different request formats and the different response formats that are valid for success and error scenarios of the route, check if all the cases are being handled properly in the PR. Check if there are unit tests for all the scenarios.
* Understand all the new methods written or existing methods updated, try to get a high-level picture of each method’s functionality and how the methods are interacting.

#### What to Look for in a code review:

1. **Design**: Is the code well-designed? Are the SOLID principles properly followed? I would recommend getting aware of SOLID principles If someone isn’t aware of them yet.
2. **Functionality**: Does the code behave as it is intended? Could there be race conditions etc.?
3. **Complexity**: Can it be made simpler? Will it be easy to understand for other developers?
4. **Unit tests:** Ensure that unit tests are written for all the scenarios, trackers are created for missing unit tests.
5. **Others:**
   1. Verify if the errors have the source defined correctly and if the errors are correctly defined in errorStore.js.
   2. Verify for null checks.
   3. Ensure that none of the [common mistakes](#_z20pg4zuqtg) are part of the PR.
   4. Verify if the error scenarios are being handled properly by throwing the exceptions.
   5. Check if the interactions of various methods make sense or can it be made simpler?
   6. Check the input formats / parameters and the return types of the method. The return type must be consistent from a method. We shouldn’t be returning arrays in few scenarios and objects in few scenarios.
   7. Check for naming conventions and see if they can be improved.
   8. Check if there are any global variables declared and see if that will cause any issues.

## Common mistakes being done that can easily be avoided:

#### Try/catch around asynchronous code:

Wrapping the asynchronous code in try/catch will not work as expected. This is because the callback’s context is completely different from the caller context. Try to read about node event loop to understand why this doesn’t work as expected.

#### Writing logs in the incorrect format if any are required:

We follow a pattern of ‘logName=xyz, message=xyz, status=xyz’. Please try to follow the pattern wherever required.

#### Declaring errors in the errorStore:

ensuring that the source of the error is properly set.

* All the errors should be defined in the errorStore.js and each of them should contain <source, code, message>.

#### Writing the synchronous code in the asynchronous pattern:

For Ex:

function method1 (params…, callback) {

// business logic with some async functionality

method2({...}, function () {

// some business logic

})

}

function method2 (inputParam1, callback) {

// synchronous code

return callback(err, {...})

}



The method2 in the above code is completely synchronous. We don't need to pass the callback to it and invoke the callback with data. We can just invoke method2 in a normal way from method1 and return the required data from it.

#### Unnecessary null checks:

function method1 (...) {

let x = 'xyz'

if (!x) return error

}



We don't need to have redundant null checks to be safe. This could sometimes cause the code to behave in an unexpected way.

#### Unnecessary encapsulation:

Sometimes, if two lines are repeating at a few places, we tend to write a util function for these two lines and replace the two lines with a util method. That's unnecessary. Having too many such util methods will reduce the readability of the code and cause a lot of interdependence in the code.

#### Wrap the method around try/catch:

Always wrap the method around try/catch if there is a chance that it throws an exception.

#### Consistent return type from the method:

The return type must be consistent from a method. We shouldn’t be returning arrays in few scenarios and objects in few scenarios.

#### Handle the errors at the source:

For example:-

function method1 (.....) {

// some business logic

method2 (..., function (err, resp) {

if (err && err.code === 'xyg') return callback(...)

if (err && err.code === 'abc') return callback(...)

// some business logic

})

}



If the code in method2’s callback is looked at, we are kind of handling it to send different errors in case of different error codes. If the same method2 is being invoked from different methods all this logic needs to be handled there too. So, it's always better to pass the proper err from method2 itself than duplicating the err handling across code base.

#### Others:

* Declare the method as the class method if the object/instance context is required.
  + Sometimes, the method is declared as the stand alone function and the object (this) is passed to the method for accessing the properties of the object. The method needs to be declared as a class method if it needs access to the object properties.
* Having console.log in the PR.
  + Ensure that you don’t add such logs. Please do a self review once all the changes are done to avoid them.
* Ignoring the unittest failures.
  + Please ensure all the unit tests are passing before the PR is assigned for review.
* PR is raised to the wrong branch.
  + Ensure that PR is pointing to a correct branch.
* All the review comments aren’t addressed in the first iteration.
  + Ensure that all the PR comments are addressed before assigning the PR back for review.

## Process:

All the developers should have clarity on who's going to be the primary reviewer and who's going to be the secondary reviewer for their trackers. The primary reviewer and the developer should always be in agreement with the code changes. (Primary reviewer should have the context of the code changes even before the PR gets to his/her queue)

* Self review after raising the PR and ensure that it's clean.
  + Ensure that none of the [common mistakes](#_z20pg4zuqtg) are part of your code.
  + Ensure that all the unit tests are passing or make sure that none of the unit tests are breaking because of the changes made in your code.
  + Check if the AI code reviewer is figuring out something that needs to be fixed.
* Primary review.
  + Ensure that none of the [common mistakes](#_z20pg4zuqtg) are part of the code.
  + Ensure that the unit tests are written for all the scenarios.
  + Ensure that all the unit tests are passing or make sure that none of the unit tests are breaking because of the changes made in your code.
  + Check if the AI code reviewer is figuring out something that needs to be fixed.
* Secondary review.

## **CODE STRUCTURE**

* Each repo should be named in such a way that it represents what it does.
  + Each tuple is separated by a hyphen(-). Eg: http-adaptor, netsuite-distributed-adaptor, integrator-workers
* All source code should be placed under /src folder
* All testcases should be placed under /\_\_tests\_\_ folder
* All new repos will be using jest and it’s config is present under jest.config.json

| {  "verbose": true,  "testMatch": [  "\*\*/\_\_tests\_\_/unit/\*\*/\*.test.js"  ],  "collectCoverageFrom": [  "src/\*\*/\*.{js,jsx,mjs}",  "!src/static/\*\*"  ],  "setupFiles": [  "<rootDir>/\_\_tests\_\_/unit/setup/testSetup.js"  ],  "testTimeout": 240000 } |
| --- |

* All static files like metadata files should be placed under /src/static folder.
* We use make to run most of the commands. Create a new file called Makefile with the commands

| JEST=./node\_modules/.bin/jest  build:  rm -rf ./node\_modules  npm ci  build-production:  rm -rf ./node\_modules  npm ci --production  test:  NODE\_ENV=unittest \  $(JEST) --coverage server:  node ./src/index.js |
| --- |

* A service definition file has to be added which defines what the service does

| {  "name": "http-adaptor",  "version": "0.0.1",  "type": "REST",  "owner": "owner@celigo.com",  "ResourceDependency": [],  "ServiceDependency": [],  "node\_version": "18.12.1",  "npm\_version": "8.19.2" } |
| --- |