ES6 async features in node

> Using ES6 features in NodeJS ver 7.6 and above to create ***bulletproof*** asynchronous processes

Code samples from this document are available here es6 async tips examples

Fully working code example is available: Accounts List demo described here: Install and run

1. Brief history of asynchronous processes in nodeJS

1.1 The birth of callbacks

To handle **io** processes which require the main javascript thread to wait, NodeJS was built around **libuv**. When any request for resource is made, it is placed in a queue of callbacks. Many native node functions eg **fs** use callbacks and as a result many packages also use callback functions to handle asynchronous processes.

As typical asynchronous code got more complex problems arose. Callbacks lead to many levels of idented steps which is difficult to read/understand/debug and if the err argument isn't handled properly it can easy lead to the asynchronous process landing in 'limbo' which only becomes apparant once an unususual error off the 'sweet path' occurs. These issues of indeting and limbo errors apply even if packages such as **async** are used to create waterfall patterns etc

1.2 Advent of Promises

Since Sept 2015 native Promises were added to nodeJS. Since then there has been a shift to prefering promises to functions with callbacks. Directionally all nodejs functions will migrate to returning promises as well or instead of providing callbacks

1.2.1 Convert function-with-callback to a Promise

This basic pattern converts a function with callback to a promise:

```
new Promise((resolve, reject) => {
   fs.readFile(
      './test.txt',  function(err, result) {

     if (err) {
       reject(err)
     } else {
       resolve(result)
     }
   })
 }).then(result=>{
     console.log(result)
 }){
 }.catch(err=>{
     console.log(err)
 })
```

Similarly generators, success-fail functions, iterables etc can be converted to Promises.

Note: In above example there is a risk of not all errors being trapped see this section:Have each step return a Promise

1.2.2 Promise static methods

1. **Promise.resolve***(49).then(console.log)* // 49
2. **Promise.reject***({message:'Problem'}).then(console.log)* // {message:'Problem'}
3. **Promise.all***([Promise.resolve(49), Promise.resolve(49)]).then(console.log)* // [49,50]
4. **Promise.race***([Promise.resolve(49),Promise.resolve(49)]).then(console.log)* // 49 (first to resolve)

1.2.3 Issues with Promises

Even though Promises help to create more consistent and more reliable asynchronous patterns, the issue of indented steps or stages remains and

The repeated use of layered *new Promise((resolve,reject)=>{ .. })* patterns can lead to cumbersome constructs.

A Promise also only passes a single argument to the **.then(fn)** function, so if multiple resolved values from a chain need to be retained they have to be handled outside the chain.

1.3 ES6 Classes and async/await get added to the mix

In Feb 2017 (version 7.6 of Node) **'async' methods** were added to to the supported ES6 class specification. This guide describes how to use these and other ES6 features to create more elegant and mainatinable asynchronous patterns

1. Using class structiures data can be passed or cumulated through Promise chains in the **this** of the newed class
2. Promises, functions, generators and static method can be mixed in the same structure
3. **await** keyword allows linear sequences of async processes without need for indenting
4. Code has enhanced readability

2. Creating ES6 'Bulletproof' asynchronous processes

2.1 Overview of pattern

1. Place each step of process in a module that returns a promise if its an asynchronous step (facilitates unit testing and sharing)
2. Wrap legacy callback handlers in Promise bodies in **try { .. } catch(e){ resolve(e) }**
3. Add each step as an **async** method of a 'steps' class
4. Add a sequence-logic method which has the sequence logic laid out with **await** statements
5. Return a Promise in **.then()** or **catch()** steps of a chain to ensure errors are passed down the chain
6. Always complete chain with a **catch(err=>{ .. })** as nodeJS throws an exeption if rejected values are not handled.

In many cases you may just log the error as in 99.9..% of cases no error will be thrown but just in case you will know when it does

2.2 ES6 Classes

2.2.1 Simplest Case of ES6 class with async method

```
// simplest ES6 class with async method
let testClass = class {
  async test() {
    return await 27
  }
}
new testClass()
  .test()
  .then(console.log) // 27
  .catch(err => {
    // handle any errors
  })
```

2.2.2 '*', 'async', 'static' and 'constructor' keywords

This example illustrates the use of *, async, static and constructor

```
let test = class {
  constructor(arg) {
    this.input = arg
  }
  *generatorFn(y) {
    yield 4
    return yield this.normal(y)
  }

  async promiseFn(x) {
    return x + 1
  }
  normal(x) {
    return x + 1
  }
  static init() {
    return 6
  }
}

let tester = new test(3),
  gen = tester.generatorFn(4)

console.log(tester.input) // 3
console.log(gen.next().value) // 4
console.log(gen.next().value) // 5
console.log(test.init()) // 6

tester
  .promiseFn(6)
  .then(console.log) // 7
  .catch(err => {})
```

3. Model code for each item of pattern

3.1 Working example

In our working example there are two asynchronous processes. The first create a SOAP server has 2 steps createServer and createSOAPServer. The second has 3 asynchronous steps and one synchronous step

i) createSOAPCLient,

ii) makeSoapRequest,

iii) parseXMLToJSON and

iv) trimJSON

3.2 Place each step of process in a module that returns a promise if its an asynchronous step

Facilitates unit testing and sharing

3.2.1. create SOAP Client

A function with callback soap.createClient is wrapped in a promise and the promise is returned

Promises trap and handle many (but not all) errors Errors thrown in Promise bodies are generally trapped by the Promise and will appear in the catch method at the end of the promise chain. This isn't the case for some callback funtions occuring in the Promise body (see next item)

```
let soap = require('strong-soap').soap
module.exports = function(wsdl, options) {
  let self = this
  return new Promise((resolve, reject) => {
    soap.createClient(wsdl, options, function(err, client) {
      if (client)
        client.on('request', function(requestXML) {
          self.requestBody = requestXML
        })
      try {
        if (err) {
          reject(err)
        } else if (client) {
          // enable multiple requests with same client
          self.soapClient = client

          resolve(client)
        }
      } catch (e) {
        console.log(17, e)
        reject(e)
      }
    })
  })
}
```

3.3 Wrap legacy callback handlers in Promise bodies in a try-catch

3.3.1 Make Soap Request

A function with callback **self.soapClient.GetAccountsList()** is wrapped in a promise and the promise is returned

Errors thrown in callback functions occuring within Promise bodies may not be trapped by the Promise itself. This is because the error gets thrown in a defferent context (that of the function wrapping the callback)

It is advisable to wrap the body of the callback in a

try{ .. }catch(e){reject(e)}

wrapper to ensure that the error is trapped and passed to the catch at the end of the Promise chain

```
module.exports = function(method, requestParams) {
  let self = this
  if (!this.soapClient) {
    throw 'create soapClient before making request'
  }

  return new Promise((resolve, reject) => {
    self.soapClient[method](requestParams, {}, function(err) {
      try {
        if (err) {
          reject(err)
        } else {
          resolve(arguments[2])
        }
      } catch (e) {
        reject(e)
      }
    })
  })
}
```

3.3.2 Parse XML to JSON

Promise body wrapped in try-catch

```
let toJSON = require('xmljson').to_json
module.exports = function(xml) {
  return new Promise(function(resolve, reject) {
    try {
      toJSON(xml, function(err, result) {
        if (err) {
          resolve(err)
        } else if (result) {
          resolve(result)
        } else {
          reject('unable to parse xml')
        }
      })
    } catch (e) {
      reject(e)
    }
  })
}
```

3.4 Add each step as an async method of a 'steps' class

soapRequestSteps class holds the 'steps' of our 4-stage asynchronous process

```
// insert modules into class methods
let makeSoapRequest = require('./helpers/makeSoapRequest')
let convertXMLToJSON = require('./helpers/convertXMLToJSON')
let createSOAPClient = require('./helpers/createSOAPClient')
let trimAccountsList = require('./requestHelpers/AccountsList/trimAccountsList')

let soapRequestSteps = class {
    constructor(wsdl, options) {
      this.clientReady = this.createSOAPClient(wsdl, options)
    }

    async createSOAPClient(wsdl, options) {
      return createSOAPClient.apply(this, arguments)
    }

    async makeSoapRequest(method, params) {

      // make sure the client has been created
      if (!await this.clientReady) {
        throw 'SOAP client not available'
      }
      return makeSoapRequest.apply(this, arguments)
   }
 }
```

Errors thrown in async methods of classes are generally trapped by the generated Promsie and will appear in the catch method at the end of the Promise chain

3.5 Add a sequence-logic method which has the sequence logic laid out with await statements

Using **async** keyword allows a sequence of asynchronous steps to be followed without indenting

```
{
...

   async getAccountsList(customerId) {

     // async step
     let xml = await this.makeSoapRequest('GetAccountsList', {
       AccountsRequest: { customerId: customerId }
     })
     this.responseBody = xml

     // async step
     let json = await convertXMLToJSON(xml)

     // sync step
     return trimAccountsList(customerId, json)
   }

...

}
```

3.6 Return a Promise in .then() or catch() steps of a chain to ensure errors are passed down the chain

```
let customerId = process.argv[2] || '23456789'

let soapClient = new soapRequestSteps(
   'http://127.0.0.1:5089/accountsList?WSDL',
   {}
)

soapClient
  .getAccountsList(customerId)
  .then(result => {
    console.log()
    console.log(result)
    console.log()
  })
  .then(() => {
    soapClient.logExchange()
  })
```

3.7 6. Always complete chain with a catch(err=>{ .. })

nodeJS throws an exception if rejected values are not handled by a catch at end of chain

```
let soapClient = new soapRequestSteps(
   'http://127.0.0.1:5089/accountsList?WSDL',
   { /* SOAP options */ }
)

soapClient
  .getAccountsList(customerId)
  .then(result => {
    console.log()
    console.log(result)
    console.log()
  })
  .then(() => {
    soapClient.logExchange()
  })
  .catch(err => {
    console.log(62, err)
  })
```

Want to avoid this:

```
(node:62158) [DEP0018] DeprecationWarning: Unhandled promise rejections are deprecate
d.
In the future, promise rejections that are not handled will terminate the Node.js
process with a non-zero exit code.
```