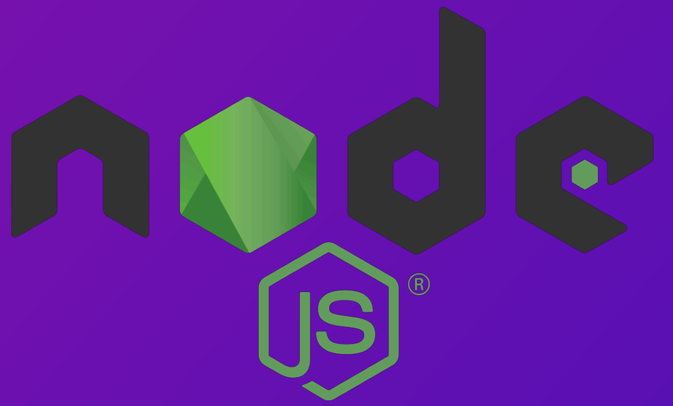
Visualize Future

Common Mistake In Nodejs

nguyentvk@uiza.io

- Introduction
- Forget close connection
- Blocking the event loop
- Executing a callback multiple times
- Callback Hell
- Best practice
- Reference



Introduction

- Node.js has seen an important growth in the past years, with big companies such as Netflix, Linkedin, PayPal adopting it

- More and more people are picking up Node and publishing modules to NPM at such a pace that exceeds other languages

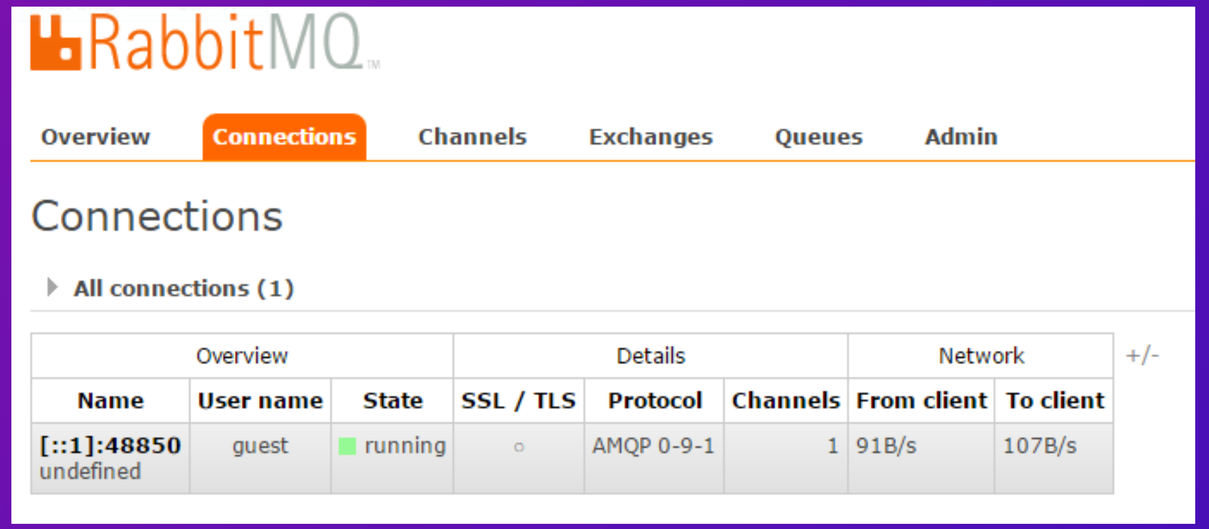
- In this article we will talk about the most common mistakes Node developers make and how to avoid them

Forget close connection

RabbitMQ

- In the past, Uiza had a problem with rabbitMQ about connection/channel. This problem happen when migration data for TOPICA. RabbitMQ was not accept any connection, so message couldn’t delivery to special queue. Detail error was “ECONNREFUSED”.

- After two days later, reason was found “not close channel after create”. The limit connection of rabbitMQ is about 1024 by default . When you forgot close channel, number connection grow up very fast. When RabbitMQ reaches this limit, it can't accept more connections.



Forget close connection

RabbitMQ

- Solution

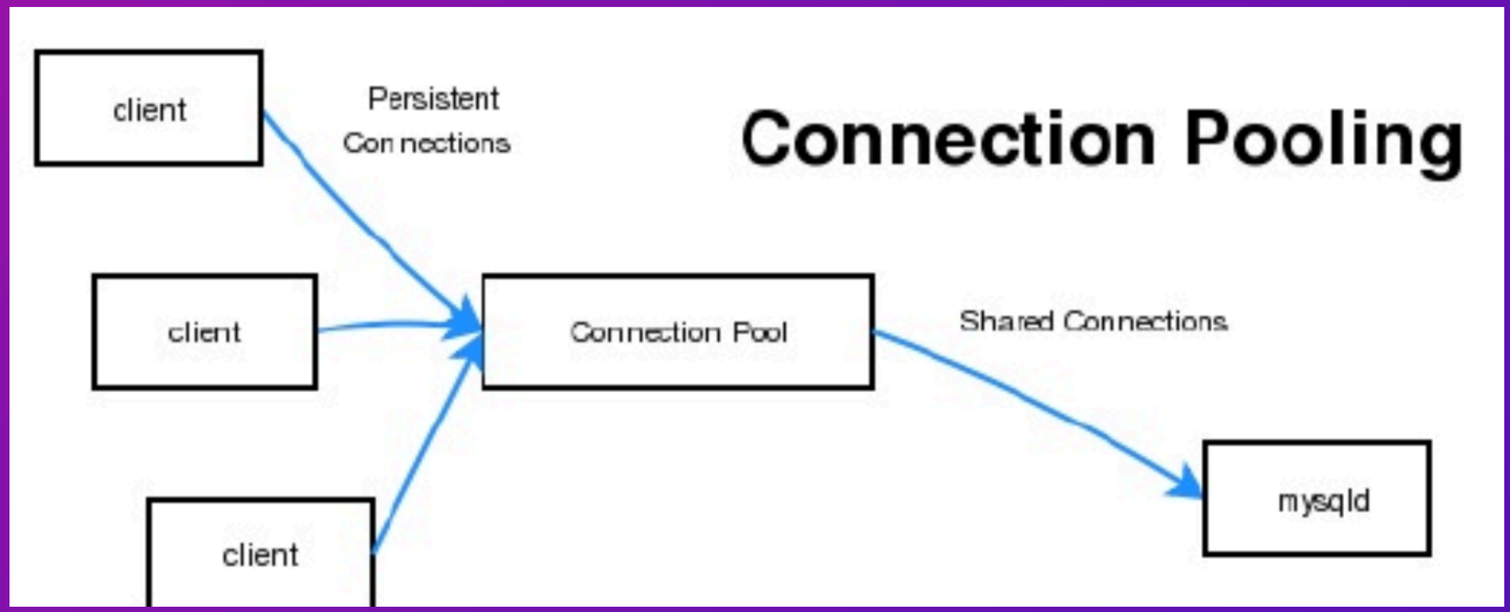
```
let channel = null;
try {
  channel = await this.createChannel(queueName, queueType);
  channel.sendToQueue( queueName, new Buffer(message), {persistent: true});
} catch (err) {
  console.log(`ERROR queue actions create channel`, err);
  throw err
}
if(channel && typeof (channel.close) === 'function' )
  channel.close();
```

Forget close connection

Mysql pool connection

- In the past, Uiza also had a problem with pool connection. In this case, many service couldn’t connect to mysql server, then all most apis response with error code 504. Detail error was “ECONNRESET”.

- This is a critical error. After few hours later, we found reason was found reason and solution. A few connection didn’t release after query done. So when number connection is equal limit connection of pool.



Forget close connection

Mysql pool connection

- Solution

```
var mysql = require('mysql');
var pool  = mysql.createPool(...);

pool.getConnection(function(err, connection) {
  if (err) throw err; // not connected!

  // Use the connection
  connection.query('SELECT something FROM sometable', function (error, results, fields) {
    // When done with the connection, release it.
    connection.release();

    // Handle error after the release.
    if (error) throw error;

    // Don't use the connection here, it has been returned to the pool.
  });
});
```

Read more: https://www.npmjs.com/package/mysql#connection-options

Blocking the event loop

- Node.js runs on a single thread, everything that will block the event loop will block everything

- That means that if you have a web server with a thousand connected clients and you happen to block the event loop, block everything.

=> So handling input-output operations asynchronously

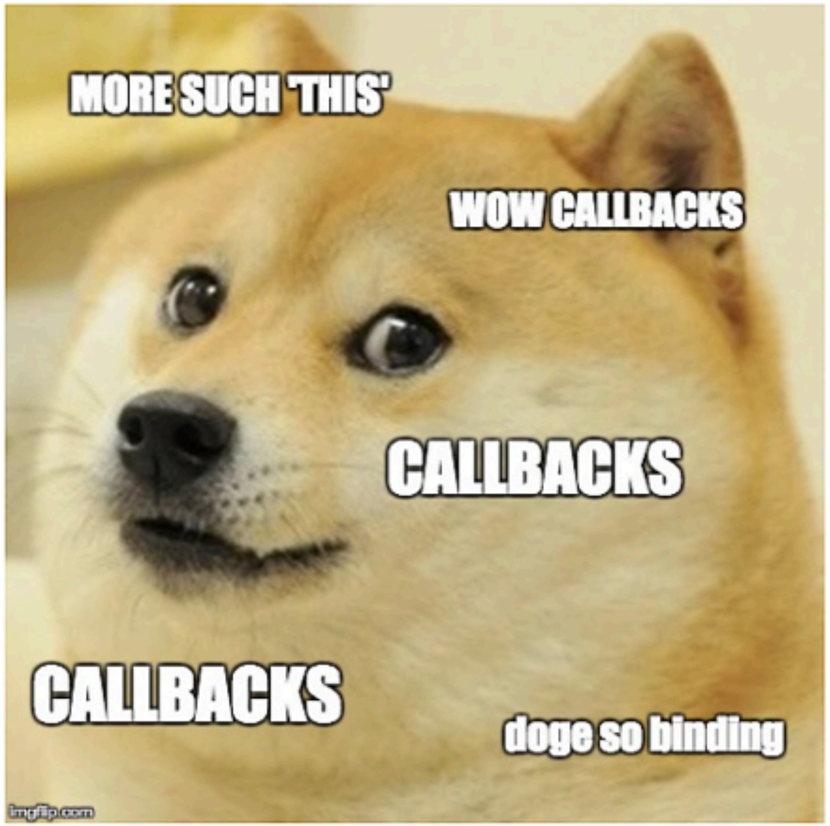
Example:
- read sync content from big file
- Parse json from big data

```
app.get('/block', (req, res) => {
  const end = Date.now() + 5000;
  while (Date.now() < end) {
    const doSomethingHeavyInJavaScript = 1 + 2 + 3;
  }
  res.send('I am done!');
});

app.get('/non-block', (req, res) => {
  // Imagine that setTimeout is IO operation
  // setTimeout is a native implementation, not the JS
  setTimeout(() => res.send('I am done!'), 5000);
});
```

Executing a callback multiple times

Have you ever seen this error ? “Error: Can't set headers after they are sent.”
If yes, you called a callback multiple times

```
request(proxyUrl, function(err, r, body) {
  if (err) {
    respond(err, {
    res: res,
    proxyUrl: proxyUrl
    });
  }

  respond(null, {
    res: res,
    body: body,
    proxyUrl: proxyUrl
  });
});
```



Executing a callback multiple times

```
let customParams = {
  where: {transcodeFileId: transcodeFileId},
  process: data.status,
  progress: data.percentage,
  adminUserId: ctx.params.adminUserId,
  appId: ctx.params.appId
}
ctx.call('v1.transcode.history.updateByCustomParams', customParams).catch((error) => {
  SlackService.error('ERROR v1.transcode.history.updateByCustomParams ERROR', error)
  return cb(error) //<------
})
if (customParams.process === 'success' && customParams.progress === 100) {
  ctx.call('v1.media.entity.updateByCustomParams', customParams).catch((error) => {
    SlackService.error('ERROR v1.media.entity.updateByCustomParams', error)
    return cb(error) //<------
  })
}
return cb() //<------
```

How many callback was called ?

Callback Hell

Callbacks are just the name of a convention for using JavaScript functions

Callback hell is caused by poor coding practices

```
function hell(win) {
 // for listener purpose
 return function() {
   loadLink(win, REMOTE_SRC+'/assets/css/style.css', function() {
     loadLink(win, REMOTE_SRC+'/lib/async.js', function() {
       loadLink(win, REMOTE_SRC+'/lib/easyXDM.js', function() {
         loadLink(win, REMOTE_SRC+'/lib/json2.js', function() {
           loadLink(win, REMOTE_SRC+'/lib/underscode.min.js', function() {
             loadLink(win, REMOTE_SRC+'/lib/backbone.min.js', function() {
               loadLink(win, REMOTE_SRC+'/dev/base_dev.js', function() {
                 loadLink(win, REMOTE_SRC+'/assets/js/deps.js', function() {
                   loadLink(win, REMOTE_SRC+'/src/' + win.loader_path + '/loader.js', function() {
                     async.eachSeries(SCRIPTS, function(src, callback) {
                       loadScript(win, BASE_URL+src, callback);
                     });
                   });
                 });
               });
             });
           });
         });
       });
     });
   });
 };
}
```

How do you feel when reading code like this?

Callback Hell

using async lib: https://github.com/caolan/async/blob/v1.5.2/README.md
Control Flow: waterfall, parallel, series, times

```
const verifyUser = function(username, password, callback){
   dataBase.verifyUser(username, password, (error, userInfo) => {
       if (error) {
           callback(error)
       }else{
           dataBase.getRoles(username, (error, roles) => {
               if (error){
                   callback(error)
               }else {
                   dataBase.logAccess(username, (error) => {
                       if (error){
                           callback(error);
                       }else{
                           callback(null, userInfo, roles);
                       }
                   })
               }
           })
       }
   })
};
```

```
async.waterfall([
    function(callback) {
        callback(null, 'one', 'two');
    },
    function(arg1, arg2, callback) {
      // arg1 now equals 'one' and arg2 now equals 'two'
        callback(null, 'three');
    },
    function(arg1, callback) {
        // arg1 now equals 'three'
        callback(null, 'done');
    }
], function (err, result) {
    // result now equals 'done'
});
```

Callback Hell

using Promise: async await

- The resulting code is much cleaner.
- Error handling is much simpler and it relies on try/catch
- Debugging is much simpler

```
const verifyUser = function(username, password, callback){
   dataBase.verifyUser(username, password, (error, userInfo) => {
       if (error) {
           callback(error)
       }else{
           dataBase.getRoles(username, (error, roles) => {
               if (error){
                   callback(error)
               }else {
                   dataBase.logAccess(username, (error) => {
                       if (error){
                           callback(error);
                       }else{
                           callback(null, userInfo, roles);
                       }
                   })
               }
           })
       }
   })
};
```

```
const verifyUser = function(username, password) {
   database.verifyUser(username, password)
       .then(userInfo => dataBase.getRoles(userInfo))
       .then(rolesInfo => dataBase.logAccess(rolesInfo))
       .then(finalResult => {
           //do whatever the 'callback' would do
       })
       .catch((err) => {
           //do whatever the error handler needs
       });
};
```

Error handling

Class Error

Error objects capture a "stack trace" detailing the point in the code at which the Error was instantiated, and may provide a text description of the error.

All errors generated by Node.js, including all System and JavaScript errors, will either be instances of, or inherit from, the Error class.

error.code# The error.code property is a string label that identifies the kind of error. error.code is the most stable way to identify an error.

error.message# The error.message property is the string description of the error as set by calling new Error(message).

error.stack# The error.stack property is a string describing the point in the code at which the Error was instantiated.

More detail: https://nodejs.org/api/errors.html#errors_error_propagation_and_interception

Error handling - Synchronous programming

The try statement lets you test a block of code for errors.
The catch statement lets you handle the error.
The throw statement lets you create custom errors.
The finally statement lets you execute code, after try and catch, regardless of the result.

```
function slugifyUsername(username) {
    if(typeof username !== ‘string’) {
        throw new TypeError('expected a string username, got '+(typeof username))
    }
    // ...
}

try {
    var usernameSlug = slugifyUsername(username)
} catch(e) {
    console.log(‘Oh no!’)
}
```

```
Oh no! TypeError: expected a string username, got number
    at slugifyUsername (/Users/nguyentvk/Site/uiza/uiza-module-cdn-controller/commonmistake.js:16:13)
    at Object.<anonymous> (/Users/nguyentvk/Site/uiza/uiza-module-cdn-controller/commonmistake.js:21:24)
    at Module._compile (module.js:635:30)
    at Object.Module._extensions..js (module.js:646:10)
    at Module.load (module.js:554:32)
    at tryModuleLoad (module.js:497:12)
    at Function.Module._load (module.js:489:3)
    at Function.Module.runMain (module.js:676:10)
    at startup (bootstrap_node.js:187:16)
    at bootstrap_node.js:608:3
```

But with asynchronous throw error in callback alway right?

Error handling - Asynchronous programming

In this case,
- CATCH not called
- You just receive a error message and Server has been stoped

=> Why?

- You are throwing error inside an asynchronous setTimeout call, which will lead to an uncaught error.
- The asynchronous code will not execute in the same context as the try-catch block.
- This has nothing to do with the promise API. This is just part of the behavior of asynchronous code execution in JavaScript.

```
function foo(a, b, cb) {
  setTimeout(() => {
    cb(new Error('Custom Error Inside Callback'));
  }, 1);
}
const getThePromise = () => {
  //throw new Error('Custom Error outside Callback');
  return new Promise((resolve, reject) => {
    foo('aaaa', 'bbbb', (err) => {
      throw err;        // catch not called
      //reject(err);  // catch called
    });
  });
}

getThePromise().then((res) => {
  console.log('getThePromise: Then...', res);
}).catch((err) => {
  console.log('getThePromise: CATCH:', err);
}
```

DEMO

Error handling - Asynchronous programming

Async caolan mixing async/await

Error handling - Asynchronous programming

```
let validateObject = (obj)=>{
  if (typeof obj !== 'object') {
    throw new Error('Invalid object');
  }
};
try {
  validateObject('123');
}catch (err) {
  console.log('Catch Thrown: ' + err.message);
}
```

```
let validateObjectB = (obj, callback)=>{
  setTimeout(()=>{
    if (typeof obj !== 'object') {
      throw new Error('Invalid object');
    }
    return callback(null);
  },1000);
};
try{
  validateObjectB('123', (err)=>{
    console.log('Callback err=: ',err);
  });
}catch(error){
  console.log('catch Callback error=: ',error);
}
```

```
let promiseEx = new Promise((resolve, reject)=>{
  setTimeout(function() {
    throw 'Uncaught Exception!';
  }, 1000);
});
promiseEx.catch((e)=>{
  console.log('Here catch error=', e);
});
```

Async caolan mixing async/await

```
let exPromise = (item)=>{
  return new Promise((resolve, reject)=>{
    setTimeout(()=>{
      console.log('exPromise item=', item);
      resolve(item);
    }, 1000);
  });
};
async.eachSeries(['aaa','bbb','ccc'], async (item, cb)=>{
  console.log('item', item);
  await exPromise(item)
  cb()
}, (err)=>{
  if(err)
    console.log(`async.each; err=`,err);
  else
    console.log(`async.each; all done`);
});
```


```
let exPromise = (item)=>{
  return new Promise((resolve, reject)=>{
    setTimeout(()=>{
      console.log('exPromise item=', item);
      resolve(item);
    }, 1000);
  });
};
async.eachSeries(['aaa','bbb','ccc'], async (item)=>{
  console.log('item', item);
  await exPromise(item)
}, (err)=>{
  if(err)
    console.log(`async.each; err=`,err);
  else
    console.log(`async.each; all done`);
});
```

Q&A

Reference

Top 10 Mistakes Node.js Developers Makehttps://www.airpair.com/node.js/posts/top-10-mistakes-node-developers-make

Common Mistakes That Node.js Developers Make
https://medium.com/swlh/common-mistakes-that-node-js-developers-make-9df7106d09f1

Thank **You**