Promises Node Js with Q module

Code Examples

A promise

```
var promise = doSomethingAsync()

promise.then(onFulfilled, onRejected)
```

> “A promise is an abstraction for asynchronous programming. It’s an object that proxies for the return value or the exception thrown by a function that has to do some asynchronous processing.” *Kris Kowal*

pseudocode

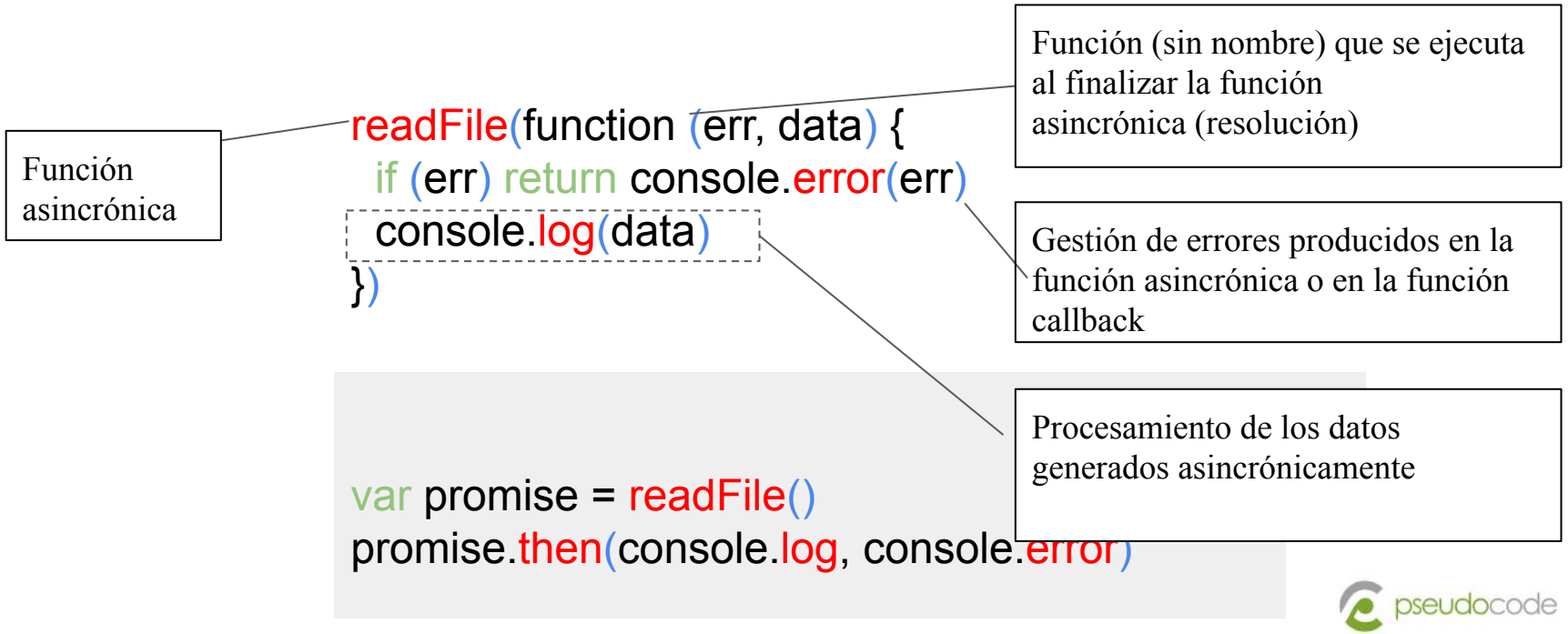
Promise vs Callback

```
readFile(function (err, data) {
  if (err) return console.error(err)
  console.log(data)
})
```

```
var promise = readFile()
promise.then(console.log, console.error)
```

pseudocode

Promise vs **Callback**



Promise vs Callback

```
readFile(function (err, data) {
  if (err) return console.error(err)
  console.log(data)
})
```

Función asincrónica

Promesa: un objeto que es un proxy de un resultado futuro

función invocada si se cumple la promesa (resolución)

método que permite asociar 2 callbacks a la compleción de la promesa

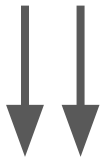
función si la promesa no se cumplió (error)

```
var promise = readFile()
promise.then(console.log, console.error)
```

Chaining promises

```
var promise = readFile()

promise.then(readAnotherFile, console.error)
```



```
var promise = readFile()

var promise2 = promise.then(readAnotherFile, console.error)
```

pseudocode

Chaining promises

```
var promise = readFile()

promise.then(readAnotherFile, console.error)
```

`promise2` es la promesa de la compleción de `readAnotherFile`

Si `readAnotherFile` hace un return o un throw entonces se puede encadenar la promesa al siguiente .then (que en este ejemplo no aparece)

```
var promise = readFile()

var promise2 = promise.then(readAnotherFile, console.error)
```

pseudocode

Chaining promises

```
var promise = readFile()

var promise2 = promise.then(function (data) {

    return readAnotherFile() // if readFile was successful, let's readAnotherFile

}, function (err) {

  console.error(err) // if readFile was unsuccessful, let's log it but still readAnotherFile

  return readAnotherFile()

})

promise2.then(console.log, console.error) // the result of readAnotherFile
```

pseudocode

Chaining promises

```
readFile()
  .then(readAnotherFile)
  .then(doSomethingElse)
  .then(...)
```

Chaining promises

```
readFile()
  .then(function (buf) {
      return JSON.parse(buf.toString())
  })
  .then(function (data) {
      // do something with `data`
  })
```

Chaining promises

```
return getUsername()
.then(function (username) {
    return getUser(username)
    .then(function (user) {
        // if we get here without an error,
        // the value returned here
        // or the exception thrown here
        // resolves the promise returned
        // by the first line
    })
});

// nested version of promises
```

pseudocode

Chaining promises

```
return getUsername()
.then(function (username) {
    return getUser(username);

})
.then(function (user) {
        // if we get here without an error,
        // the value returned here
        // or the exception thrown here
        // resolves the promise returned
        // by the first line
});
// chained version of promises
```

pseudocode

Chaining promises

```
function authenticate() {
        return getUsername()
        .then(function (username) {
            return getUser(username);
        })  // chained because we will not need the user name in the next event
        .then(function (user) {
                return getPassword(user)
                // nested because we need both user and password next
                .then(function (password) {
                if (user.passwordHash !== hash(password) )
                                throw new Error("Can't authenticate");
                });
        });
}
```

Error handling

```
try {
  doThis()
  doThat()
} catch (err) {
  console.error(err)
}
```

// Try - Catch en código síncrono

pseudocode

Error handling

```
doThisAsync()
  .then(doThatAsync)
  .then(null, console.error)
```

// gestión de errores en código asincrónico, si se produce una excepción en doThisAsync()
// entonces no se ejecutará doThatAsync().

pseudocode

Error handling

```
doThisAsync()
  .then(function (data) {
        data.foo.baz = 'bar' // throws a ReferenceError as foo is not defined
  })
  .then(null, console.error)
```

// Si se produce una excepción en la función callback que se invoca si la
// promesa se ha cumplido, será capturada en el siguiente .then

pseudocode

Error handling

```
doThisAsync()
  .then(function (data) {
        if (!data.baz) throw new Error('Expected baz to be there')
})
  .then(null, console.error)
```

// Si se lanza una excepción en la función callback que se invoca si la
// promesa se ha cumplido, será capturada en el siguiente .then

pseudocode

Error handling

```
try {
  throw new Error('never will know this happened')
} catch (e) { }
```

Excepciones no capturadas

```
readFile()
  .then(function (data) {
      throw new Error('never will know this happened')
  })
```

pseudocode

Error handling

```
readFile()
  .then(function (data) {
        throw new Error('now I know this happened')
  })
  .then(null, console.error)
```

// Se añade una promesa al final cuyo cometido es capturar todas aquellas
// excepciones sin tratar que llegan a la última promesa

pseudocode

Error handling

```
readFile()
  .then(function (data) {
        throw new Error('now I know this happened')
  })
  .fail(console.error)
```

// fail(onRejected) es lo mismo que .then(null,onRejected)
// misma semántica que .catch(onRejected)

pseudocode

Error handling

```
readFile()
  .then(function (data) {
        throw new Error('now I know this happened')
  })
  .fail(console.error)
  .done()
```

// done( ) se utiliza para terminar la cadena de promesas, dado que captura
// cualquier excepción y la lanza en un evento asíncrono que producirá una
// uncaughtException en el objeto process de Node js

Creating promises

Converting callbacks to promises

```
var Q = require(‘q’); var fs = require(‘fs’);

var fs_readFile = Q.denodeify(fs.readFile)

var promise = fs_readFile('myfile.txt')

promise.then(console.log, console.error)
```

// denodeify(function, ..args) devuelve una función para crear promesas
// al instanciar promise creamos la promesa, se genera la tarea asincrónica
// denodeify == nfbind

Creating promises

```
var Q = require('q'); var fs = require('fs');

Q.nfcall(fs.readFile, "myfile.txt", "utf-8").then(console.log, console.error)
```

// nfcall genera una promesa con la invocación de la función con callbacks al mismo tiempo
// parecida semántica que nfapply, pero esta acepta un array con los parámetros

```
Q.nfapply(fs.readFile, ["myfile.txt", "utf-8"]).then(console.log, console.error)
```

Creating promises

```
var Q = require('q'); var fs = require('fs');

function fs_readFile (file, encoding) {
  var deferred = Q.defer()
  fs.readFile(file, encoding, function (err, data) {
        if (err) deferred.reject(err) // rejects the promise with `er` as the reason
        else deferred.resolve(data) // fulfills the promise with `data` as the value
  })
  return deferred.promise // the promise is returned
}
```

U S E

```
fs_readFile('myfile.txt').then(console.log, console.error)
```

// fs_readFile devuelve una promesa de un resultado futuro cuando se invoca con parámtros
// es decir, que este código es equivalente a lo que hace denodeify(fs.readFile);

Creating promises

```
var Q = require('q'); var fs = require('fs');

function fs_readFile (file, encoding) {
  var deferred = Q.defer()
  fs.readFile(file, encoding, function (err, data) {
        if (err) deferred.reject(err) // rejects the promise with `er` as the reason
        else deferred.resolve(data) // fulfills the promise with `data` as the value
  })
  return deferred.promise // the promise is returned
}
```

U S E

```
fs_readFile('myfile.txt').then(console.log, console.error);
```

// deferred es un objeto con un campo promise, y los métodos reject, resolve, notify

Creating promises with callbacks

```
var Q = require('q'); var fs = require('fs');

function fs_readFile (file, encoding, callback) {
    var deferred = Q.defer()
    fs.readFile(file, encoding, function (err, data) {
        if (err) deferred.reject(err) // rejects the promise with `er` as the reason
        else deferred.resolve(data) // fulfills the promise with `data` as the value
    });
    return deferred.promise.nodeify(callback) // the promise is returned if callback it's not a
}                                             // function, if it is a function it is called when/if
                                              // the promise is rejected or fulfilled
```

U S E

```
fs_readFile('myfile.txt', 'utf-8', function(err,data) {    if (err) console.error(err)
                                                           else console.log(data)   })
fs_readFile('myfile.txt', 'utf-8').then(console.log, console.error)
```

Synchronizing promises

```
var Q = require('q'); var fs = require('fs');

var allPromise = Q.all([fs_readfile('file1.txt'), fs_readfile('file2.txt')])
```

U S E

```
allPromise.then(console.log,console.error);
```

// all returns a promise that is fulfilled with an array containing the fulfillment value of each
// promise, or is rejected with the same rejection reason as the first promise to be rejected.

U S E

```
allPromise.spread(function(content1, content2) {
        console.log('content of first file: ');  console.log(content1);
        console.log('content of second file: ');  console.log(content2);  }, function (err) {
        console.error(err.toString())});
```

// spreads converts array in fulfillment response to a variable number of arguments

Example

```
var mysql = require('mysql');
var connection = mysql.createConnection({
     host: 'localhost',
     user: 'user',
     password: 'password',
     database: 'db'
});

exports.getUsers = function (callback) {
     connection.connect(function () {
     connection.query('SELECT * FROM Users', function (err, result) {
          if(!err){
               callback(result);
          }
     });
     });
};
```

Example

```
exports.getUsers = function getUsers () {
  // Return the Promise right away, unless you really need to
  // do something before you create a new Promise
  return new Promise((resolve, reject) => {
      // reject and resolve are functions provided by the Promise
      // implementation. Call only one of them.
      // Do your logic here
      connection.query('SELECT * FROM Users', (err, result) => {
      // Handle errors first,
      if (err) {// Reject the Promise with an error
            return reject(err)
      }
      // Resolve (or fulfill) the promise with data
      return resolve(result)
      })
  })
}
```