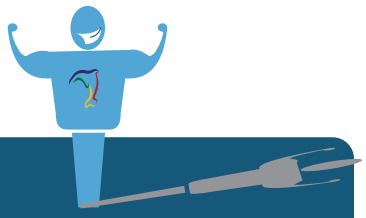
autentia

Avenida de Castilla,1 - Edificio Best Point - Oficina 21B
28830 San Fernando de Henares (Madrid)
tel./fax: +34 91 675 33 06
info@autentia.com - **www.autentia.com**

¿**Qué ofrece** Autentia Real Business Solutions S.L?

Somos su empresa de **Soporte a Desarrollo Informático**.
Ese apoyo que siempre quiso tener...

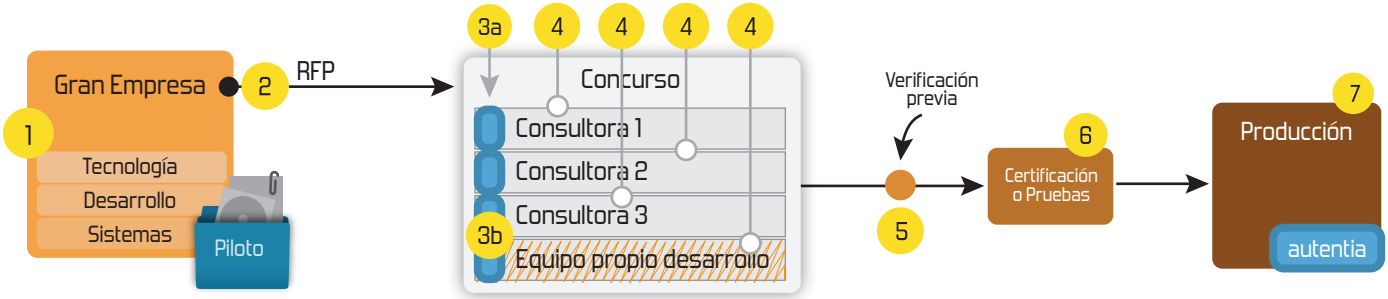
1. Desarrollo de componentes y proyectos a medida



2. Auditoría de código y recomendaciones de mejora

3. Arranque de proyectos basados en nuevas tecnologías

1. Definición de frameworks corporativos.
2. Transferencia de conocimiento de nuevas arquitecturas.
3. Soporte al arranque de proyectos.
4. Auditoría preventiva periódica de calidad.
5. Revisión previa a la certificación de proyectos.
6. Extensión de capacidad de equipos de calidad.
7. Identificación de problemas en producción.



4. Cursos de formación (impartidos por desarrolladores en activo)

- Spring MVC, JSF-PrimeFaces /RichFaces, HTML5, CSS3, JavaScript-jQuery
- Gestor portales (Liferay)
- Gestor de contenidos (Alfresco)
- Aplicaciones híbridas
- Control de autenticación y acceso (Spring Security)
- UDDI
- Web Services
- Rest Services
- Social SSO
- SSO (Cas)
- JPA-Hibernate, MyBatis
- Motor de búsqueda empresarial (Solr)
- ETL (Talend)
- Dirección de Proyectos Informáticos.
- Metodologías ágiles
- Patrones de diseño
- TDD
- Tareas programadas (Quartz)
- Gestor documental (Alfresco)
- Inversión de control (Spring)
- BPM (jBPM o Bonita)
- Generación de informes (JasperReport)
- ESB (Open ESB)

Compartimos nuestro conocimiento en:
www.adictosaltrabajo.com

Para más información visítenos en:
www.autentia.com

Paulino Corujo

Java Developer & Javascript geek. Passionate about development.

Contacta en: LinkedIn y en Twitter: @pcorujo

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2013-03-12 Tutorial visitado 94 veces Descargar en PDF

Promesas: Organiza tu código Javascript/Coffeescript

Índice

- Introducción
- Manos a la obra
- Conclusiones

Introducción

En Javascript es habitual escribir código que se ejecutará de forma asíncrona, sobre todo si nuestra aplicación hace uso de librerías de entrada/salida. Tanto si escribimos una aplicación web con Javascript corriendo en un navegador con las llamadas Ajax por ejemplo o una aplicación Javascript corriendo en servidor sobre una plataforma tipo Node.js, las funciones asíncronas suelen obligarnos a tener una anidación de funciones para tratar las diferentes callbacks que hacen poco legible nuestro código y complica el tratamiento de errores.

Las **Promesas** son un paradigma de programación en Javascript que hace que nuestro código sea más legible, evitando las callbacks anidadas, y facilitando el tratamiento de errores creando un código estilo try... catch... finally.

La definición que la propuesta de CommonJS Promises/A hace, es que una promesa es un valor que se espera en un tiempo futuro no definido, después de ejecutar una operación. Esta definición encaja en las funciones asíncronas que manejamos habitualmente cuando desarrollamos en Javascript.

Una promesa puede estar en tres estados, **no cumplida**, **cumplida**, **fallida**. Una promesa no cumplida sólo puede transicionar a cumplida o fallida, es decir, mientras ejecutamos nuestro código asíncrono y éste no finalice, nuestra promesa estará incumplida. Un vez que el código asíncrono finaliza su ejecución en un momento futuro del tiempo nuestra promesa puede haberse cumplido, con lo que obtendremos el valor esperado, o puede haber fallado con lo que obtendremos un error. En las promesas, el fallo es propagado en la cadena de promesas (si hay al menos una promesa que depende de otra) siguiendo el mismo *efecto burbuja* de los eventos.

Manos a la obra

Para seguir este tutorial necesitas:

- Node.js: Es una plataforma basada en el runtime javascript de Google Chrome (node.js).
- Grunt: Herramienta que nos simplificará la construcción de la aplicación (Grunt).

Usaremos la librería Q como implementación de CommonJS Promises/A. Para hacer más legible el código del tutorial, los ejemplos están escritos en *coffeescript*.

Para el primer ejemplo, crearemos una clase PersonDAO en el que publicaremos tres métodos que simularán un acceso a base de datos (con datos también simulados) y con un comportamiento asíncrono mediante la función *setTimeout()* de node para programar la ejecución de una función callback después de "n" milisegundos:

- getPersonByName: Retorna un objeto Person (nombre y apellidos), en este caso como podéis comprobar siempre retorna el mismo para el ejemplo que queremos construir.
- save: Simula el almacenamiento en base de datos del objeto Person recibido como parámetro.
- resolveWithErrorIfIdNotEqualsOne: Para ver cómo funcionan los errores, crearemos este método, el cual resolverá la promesa como fallida si el parámetro pasado es distinto del valor 1, en caso contrario la promesa será cumplida.

Para construir la promesa de nuestro código asíncrono, usaremos un *Deferred*. Un deferred es un envoltorio de la promesa sobre el que marcaremos en nuestro código asíncrono si la promesa ha sido cumplida o ha fallado. La diferencia entre un *Deferred* y una *Promesa* es, en definitiva, que la promesa es inmutable, es decir, ambos representan una abstracción de lo que ocurrirá en un futuro (en nuestro código asíncrono) pero solamente en el Deferred es posible cambiar el estado interno de *promesa sin cumplir* a *promesa cumplida* o *fallo*

Nuestro código de la clase **PersonDAO** quedaría entonces:

```
Q = require "Q";

class PersonDAO

    getPersonByName: (name) ->
        deferred = Q.defer()
        setTimeout ()->
            deferred.resolve
                firstName: "Homer"
                lastName: "Simpson"
        , 300
        return deferred.promise


    save: (person) ->
        deferred = Q.defer()
        setTimeout ()->
            person._id = new Date().getTime()
            deferred.resolve
                id: person._id
        , 300
        return deferred.promise

    resolveWithErrorIfIdNotEqualsOne: (someParam) ->
        deferred = Q.defer()
        setTimeout ()->
            unless someParam is 1
                deferred.reject "Error trying to invoke..."
            else
                deferred.resolve "OK"
        , 300
        return deferred.promise

module.exports = PersonDAO
```

Vamos a crear un test para probar nuestro código de la clase Person que obtenga una persona por nombre:

```
describe "Promises Q", ->
     describe "PersonDAO", ->
         beforeEach ->
             @personDao = new PersonDAO()

         describe "Cuando preguntemos por un Person de nombre Homer", ->
             it "obtendremos un objeto con la persona Homer Simpson", ->
                 endTest = false
                 runs =>
                     @personDao.getPersonByName("Homer").then (person) =>
                         expect(person.firstName).toBe "Homer"
                         expect(person.lastName).toBe "Simpson"
                         endTest = true
                 waitsFor ->
                     endTest
                 ,"endTest not asigned...timeout!"
                 ,500

                 runs ->
                     expect(endTest).toBe(true)
```

Como se puede comprobar, este código no dista mucho de lo que tendríamos si escribiéremos el test sin promesas y usando la callbacks clásicas de Javascript:

```
describe "Promises Q", ->
    describe "PersonDAO", ->
        beforeEach ->
            @personDao = new PersonDAO()

        describe "Cuando preguntemos por un Person de nombre Homer", ->
            it "obtendremos un objeto con la persona Homer Simpson", ->
                endTest = false
                runs =>
                    @personDao.getPersonByName "Homer", (person, error) =>
                        unless error
                            expect(person.firstName).toBe "Homer"
                            expect(person.lastName).toBe "Simpson"
                        else
                            expect(error).toBeUndefined()
                            endTest = true
                waitsFor ->
                    endTest
                ,"endTest not asigned...timeout!"
                ,500

                runs ->
                    expect(endTest).toBe(true)
```

Lo interesante viene cuando tratamos de encadenar diferentes llamadas asíncronas. Vamos a ampliar el ejemplo anterior para que además de obtener la persona por nombre, también le modificaremos la edad, guardaremos el cambio mediante el método *save* y, posteriormente, llamaremos al método *resolveWithErrorIfIdNotEqualsOne* con un ID igual a 1 para que la cadena de promesas no falle:

```
describe "Cuando obtengamos un Person por nombre luego cambiaremos su edad y, guardare
    it "retornará un OK", ->
        endTest = false

        runs =>
            @personDao.getPersonByName("Homer").then (person) =>
                expect(person.firstName).toBe "Homer"
                expect(person.lastName).toBe "Simpson"
                person.age = 38
                @personDao.save(person)
            .then (id) =>
                expect(id.id).toBeDefined()
                expect(id.id).toBeGreaterThan(1)
                id.id = 1
                @personDao.resolveWithErrorIfIdNotEqualsOne(id.id)
            .then (message) ->
```

Administraciones Públicas (IV): buscando en las forjas una solución.

» Gestión de expedientes en el ámbito de las Administraciones Públicas (III): BPM y la gestión de procesos de negocio.

Últimos Tutoriales del Autor

» Como interactuar con el mundo físico mediante Javascript y NodeJS

» Node, Express y MongoDB: Crea un API REST en JavaScript server-side de forma rápida, sencilla y eficiente

Categorías del Tutorial

Java Estándar

Ajax

Javascript / JQuery

Últimas ofertas de empleo

2011-09-08
Comercial - Ventas - MADRID.

2011-09-03
Comercial - Ventas - VALENCIA.

2011-08-19
Comercial - Compras - ALICANTE.

2011-07-12
Otras Sin catalogar - MADRID.

2011-07-06
Otras Sin catalogar - LUGO.

```
                expect(message).toBe("OK")
                endTest = true
            .fail (error) ->
                expect(error).toBeUndefined()
                endTest = true

        waitsFor ->
            endTest
        ,"endTest not asigned...timeout!"
        ,1000

        runs ->
            expect(endTest).toBe(true)
```

En este caso, nuestro test ya muestra la legibilidad que nos proporciona el uso de las promesas. Cada paso depués de una promesa resuelta está precedido de la palabra *then*, que recibe como parámetro el valor retornado por la promesa anterior en la cadena (si es que ha sido resuelta satisfactoriamente). Este mismo código usando las callbacks clásicas tendría este aspecto:

```
describe "Cuando obtengamos un Person por nombre luego cambiaremos su edad y, guardare
   it "retornará un OK", ->
       endTest = false

       runs =>
           @personDao.getPersonByName "Homer", (person, error) =>
               unless error
                   expect(person.firstName).toBe "Homer"
                   expect(person.lastName).toBe "Simpson"
                   person.age = 38
                   @personDao.save person, (id, error) ->
                       unless error
                           expect(id.id).toBeDefined()
                           expect(id.id).toBeGreaterThan(1)
                           id.id = 1
                           @personDao.resolveWithErrorIfIdNotEqualsOne id.id, (message,
                               unless err
                                   expect(message).toBe("OK")
                                   endTest = true
                               else
                                   expect(error).toBeUndefined()
                                   endTest = true
                       else
                           expect(error).toBeUndefined()
                           endTest = true
               else
                   expect(error).toBeUndefined()
                   endTest = true

       waitsFor ->
           endTest
       ,"endTest not asigned...timeout!"
       ,1000

       runs ->
           expect(endTest).toBe(true)
```

En caso de fallo, y según lo que se ha comentado previamente de la propagación de los errores (efecto burbuja), la función *fail* recogerá el fallo de cualquiera de las promesas, y, la funcion *fin* se ejecutará siempre al final de la cadena de promesas, por eso hemos transladado la asignación de la variable endTest a ese bloque y evitar código repetido.

Tendremos por tanto un código del estilo try... catch... finally. Vamos a escribir un test para comprobar este comportamiento:

```
describe "Cuando obtengamos un Person por nombre luego cambiaremos su edad y, posteric
   it "obtendremos un fallo porque el Id no es igual a 1", ->
       endTest = false

       runs =>
           @personDao.getPersonByName("Homer").then (person) =>
               expect(person.firstName).toBe "Homer"
               expect(person.lastName).toBe "Simpson"
               person.age = 38
               @personDao.save(person)
               .then (id) =>
                   expect(id.id).toBeDefined()
                   expect(id.id).toBeGreaterThan(1)
                   @personDao.resolveWithErrorIfIdNotEqualsOne(id)
               .then (message) ->
                   expect(message).toBeUndefined()
               .fail (error) ->
                   expect(error).toBeDefined()
               .fin () ->
                   endTest = true

       waitsFor ->
           endTest
       ,"endTest not asigned...timeout!"
       ,1000

       runs ->
           expect(endTest).toBe(true)
```

Hasta ahora hemos visto cómo encadenar promesas (*chaining*) y la propagación. Veamos ahora un ejemplo de promesas que se *combinan* para formar una única promesa. Para ello vamos a reutilizar la clase PersonDAO y vamos a modelar una clase **Puppy**, el comportamiento de nuestra clase *Puppy* es simple, nuestro perro ladra pero solamente si está contento y para hacer que nuestro perro esté contento hay que acariciarle un rato.

```
Q = require "Q";
class Puppy

    constructor: ->
        @happy = false

    stroke: ->
```

```
        deferred = Q.defer()
        setTimeout ()=>
            @happy = true
            deferred.resolve @happy
        , 500
        return deferred.promise

    bark: ->
        deferred = Q.defer()
        setTimeout ()=>
            unless @happy
                deferred.reject "Grrrr!"
            else
                deferred.resolve "Woof!"
        , 500
        return deferred.promise

module.exports = Puppy
```

Vamos a escribir un test donde combinaremos dos promesas, por un lado *buscaremos una persona por nombre* y por otro, *esperaremos que nuestro perro ladre*, el problema es que como no hemos acariciado a nuestro perro, la promesa, como un todo, fallará:

```
describe "Cuando obtengamos un Person por nombre y esperemos que nuestro perro ladre",
    beforeEach ->
        @puppy = new Puppy()
    it "obtendremos un fallo porque nuestro perro no está contento y no quiere ladrar",
        endTest = false

        runs =>
            promises = Q.all [
                @personDao.getPersonByName "Homer"
                @puppy.bark()
            ]
            promises.spread (person, bark) ->
                expect(person.firstName).toBeUndefined()
                expect(person.lastName).toBeUndefined()
                expect(bark).toBeUndefined()
            .fail (error) ->
                expect(error).toBe "Grrrr!"
            .fin () ->
                endTest = true


        waitsFor ->
            endTest
        ,"endTest not asigned...timeout!"
        ,1000

        runs ->
            expect(endTest).toBe(true)
```

Con la funcion **all** combinamos varias promesas en una sola haciendo que si una de ellas falla el resto no se evalúe. Si queremos esperar por todas las promesas independientemente de si fallan o se cumplen, podemos usar la función **allResolved**.

En nuestro test la promesa no se ha cumplido, lástima que nuestro perro no esté contento, por eso hemos recibido un **Grrrr!** que ha hecho fallar toda nuestra promesa combinada. Vamos a escribir otro test pero esta vez asegurándonos de acariciar antes a nuestro perro durante un rato (asincronía), así nuestro perro estará contento y nuestra promesa combinada se cumplirá:

```
describe "Cuando obtengamos un Person por nombre y esperemos que nuestro perro ladre des
    beforeEach ->
        @puppy = new Puppy()
    it "obtendremos una persona con nombre Homer Simpson, y nuestro perro ladrará porque
        endTest = false

        runs =>

            promises = Q.all [
                @puppy.stroke()
                @personDao.getPersonByName "Homer"
                @puppy.bark()
            ]
            promises.spread (happy, person, bark) ->
                expect(happy).toBeTruthy()
                expect(person.firstName).toBe "Homer"
                expect(person.lastName).toBe "Simpson"
                expect(bark).toBe "Woof!"
            .fail (error) ->
                expect(error).toBeUndefined()
            .fin () ->
                endTest = true


        waitsFor ->
            endTest
        ,"endTest not asigned...timeout!"
        ,3000

        runs ->
            expect(endTest).toBe(true)
```

Como podéis comprobar, la promesa se ha cumplido y, como cada promesa retorna un valor cuando se resuelve, hemos usado la función *spread* que asigna un valor a sus parámetros según el orden establecido en el array de promesas pasadas a la funcion *all* para comprobar las expectativas de nuestro test.

Conclusiones

Las promesas en Javascript forman un paradigma de programación orientado principalmente a hacer nuestro código más legible y ordenado, evitando el código spaghetti propio de la anidación de callbacks en la programación asíncrona (pyramid of doom). Existen otras implementaciones de Promesas como por ejemplo rsvp. En este ejemplo **Q** lo hace apropiado para desarrollar sobre *node.js* ya que reconoce su patrón de callbacks, aunque también lo hace compatible

para usar en front con el sistema de promesas de jQuery.

Si quieres descargar el código, puedes hacerlo en: git@bitbucket.org:pcorujo/promises-tutorial.git