PRACTICA 3:  
BBDD orientadas a documentos

Modelos Avanzados de Bases de datos

**Profesor: Antonio Moratilla Ocaña**

**Autores: Adrián Sagredo Arnaiz**

**David Varela García**

**Alin Nicolae Giurca**

**Sergio Sanz García**

**Fernando García Fernández**

# Creación de la Aplicación

Cuando se ejecuta la aplicación web, llama al servlet SimularTrabajo.java. En este servlet se instancia la clase Aplicación y se ejecuta su método simular trabajo, en el que se ejecutarán diferentes funciones simples que generarán mensajes para luego introducirlos a nuestra base de datos MongoDB previo envío del mensaje al servidor REST. (Implementado actualmente solo para mongoDB en local que este en escucha y con las líneas de código que hacen la inserción comentadas por defecto). Los mensajes nos los responderá el servidor REST escritos por consola en formato JSON.

# Creación de los trabajos “Creación y envío de las Clases mensajes a nuestro Cliente”

En esta clase es donde se ejecutan las funciones de manera aleatoria dentro de la función simular trabajo. Simplemente es un bucle con un switch donde se ejecutan 3 funciones que se verán a continuación.

También hay que recalcar que esta clase Aplicación, contiene un atributo Cliente, que llamando a la función postRequest() manda al servidor REST un objeto Mensaje, y gracias a la librería de Jackson convierte la Clase Mensaje en un formato Json para después insertarlo como un documento dentro de la base de datos Mongo DB.

Los mensajes que se envían es uno al principio de la ejecución, uno al final de la ejecución, un mensaje de aviso cuando en una función se cumplen ciertas condiciones que establecemos (opcional) y un mensaje de error cuando la función sufre algún tipo de error en su ejecución (opcional). Las funciones son las siguientes:

* DivisiónSencilla():.

La función realiza una división que con una probabilidad de 1/10 produce una excepción de división por cero, y se envíe un, mensaje de error.

* posiciónLiga():

La estructura es muy parecida a la anterior, solo que en esta función se simula la asignación de una posición a un equipo dentro de una liga. Se crea el mensaje de inicio, posteriormente se simula un aviso de la función, en la que implicaría el descenso en la posición que esté. Concretamente den las posiciones 20, 19 y 18.

* funcionLarga()

El objetivo de esta función es simular que se está tardando en ejecutar la función (Gracias al sleep del hilo principal que se está ejecutando, que realmente es la espera que se realiza.)

Si la ejecución de la función es superior a 45 ms se crea un mensaje de warning con lo que hemos especificado anteriormente.

También como es lógico previamente se crea el mensaje de inicio y después de la parte de la condición del mensaje de warning, se crea un mensaje de fin independientemente de si se ha producido el aviso o no.

Si se produce un error, también en esta función se ejecuta una excepción para la creación del mensaje de error.

# Lógica de negocio (Clase Mensaje).

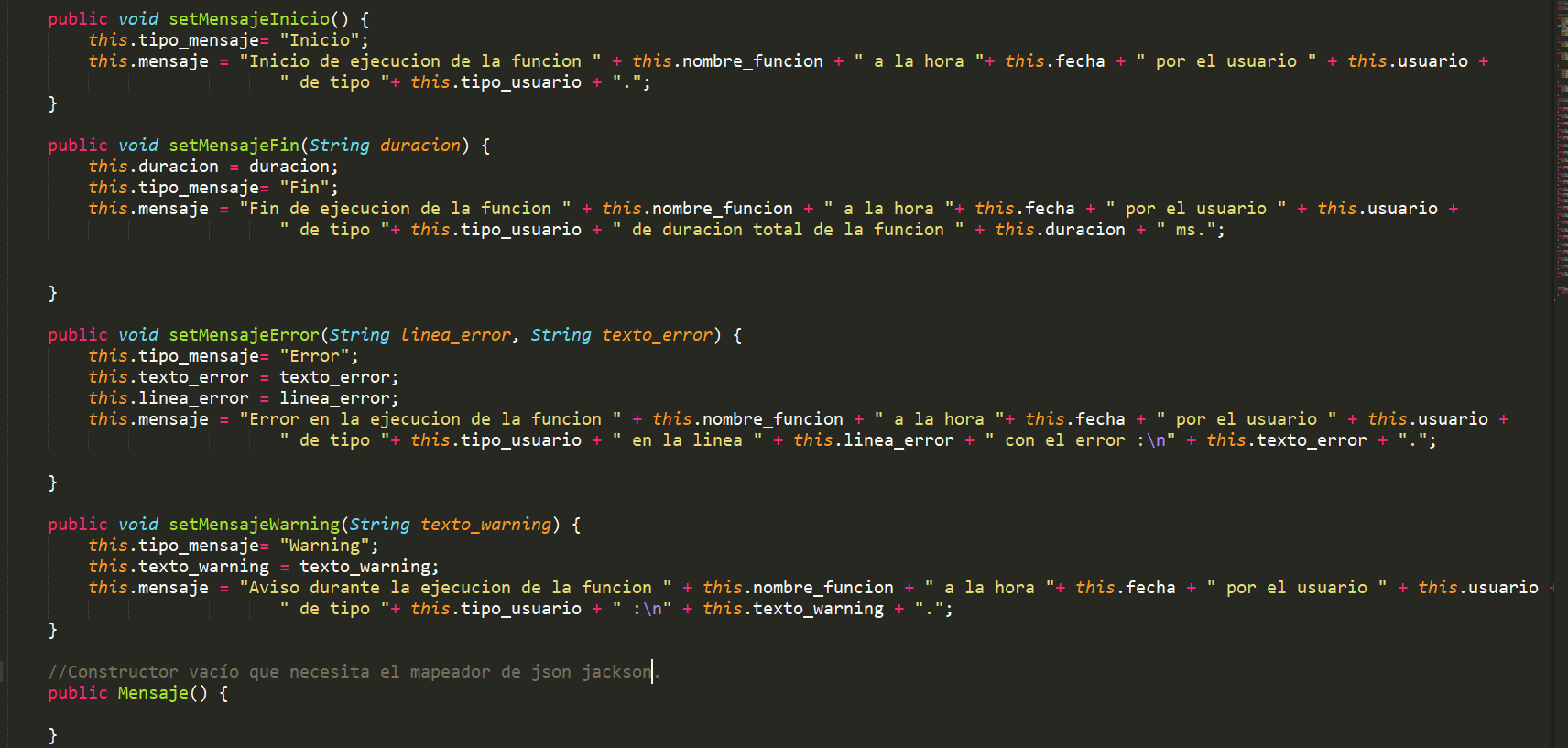
En esta parte observaremos la estructura de los mensajes que se crean en nuestra aplicación, como se observa en la siguiente captura:

## 

Al principio, se establece que el json (Clase transformada por Jackson) no incluirá los campos que sean nulos en ese momento en el mensaje.

Como se puede observar, dentro de la clase se guardan todos los datos que necesitamos para construirlo, usarios, tipo de usario la fecha, nombre de la función que se está ejecutando ... y el atributo mensaje como String (que es el propio mensaje) y algunos atributos útiles para la creación de mensajes de warning y error.

Dentro del constructor, se dan valor a todos los atributos, siendo algunos por defecto como la fecha. También hay un constructor vacío sin argumentos que es necesario para que Jackson pueda hacer la conversión a JSON internamente.

Por finalizar para establecer los distintos tipos de mensajes, se realiza mediante la ejecución de los diferentes sets del tipo de mensaje.

Cada uno cambia el atributo mensaje de la clase, con una contracción del string diferente, para cada tipo de mensaje concatenando con los atributos de la clase, especificados al principio en este apartado.

# Despliegue del cluster y configuración

Para el despliegue de la base de datos hemos utilizado Docker. Para realizar la replicación y el balanceo de carga de los datos que se van a almacenar se ha optado por crear un nodo enrutador, a través del cual se podrá leer y escribir en la base de datos. De él, cuelgan 3 nodos donde se almacenarán todos los datos de nuestro sistema. Entre estos dos puntos también se han implementado 3 nodos intermediaros en los cuales se almacenarán los metadatos y las diferentes configuraciones del clúster.

Por lo tanto, nuestro sistema está compuesto por 7 nodos desplegados con Docker. Para desplegarlos hemos creado un archivo Docker-compose para poder lanzar todos los nodos simplemente ejecutando un único archivo.

Una vez creados los contenedores de Docker con nuestros nodos hemos realizado la configuración de estos.

*Tanto el Docker-compose.yml como los archivos de configuración de los nodos se han adjuntado junto a este documento.*

* *Config. enrutador: configEnrutador.js*
* *Config. nodos configuración: configServer.js*
* *Conig. Nodos Shard: configShard.js*

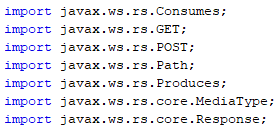
# Conexión a Servidor local

La realización de nuestro servicio dentro de la aplicación se realiza dentro del archivo 

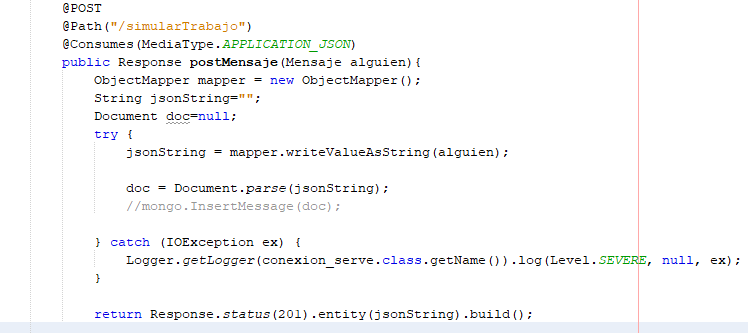
Importamos los datos mensajes para poder utilizarlos.



Esta clase funciona a través de java rs.



El siguiente método Post funciona cuando están en la dirección “/mensaje/data” para recoger ese mensaje en formato JSON y lo devuelvo en una cadena de texto.

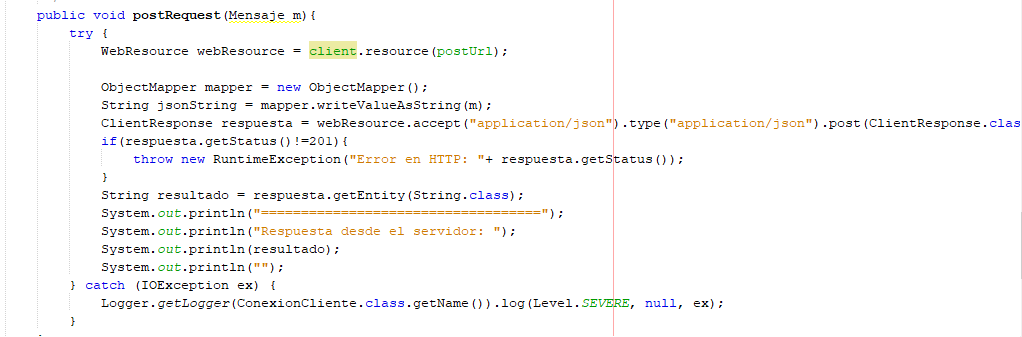


Para comprobar que la conexión funciona, creamos un cliente que se conecte al servidor.

El código del cliente se encuentra en el archivo . Establecemos las url que necesitamos para conectarnos.



Establecemos un método que envíe un JSON a través de un servicio REST utilizando el método POST.



Por último, también tenemos un método get en el servidor REST que posiblemente en el futuro se utilice para implementar consultas a la BBDD de MongoDB y ver las respuestas.

# Consultas MongoDB

Las funciones expuestas a continuación, y las cuales se han probado en una instancia MongoDB individualizada y con unos datos de ejemplo añadidos a mano en esta misma instancia, pueden tener los nombres de las variables distintos a los que finalmente se implementen para la PECL4 y distintos respecto a los atributos antes mostrados en Java, en la que ya si estará implementada la conexión de mongo con el aplicativo Java de modo que los valores de las variables corresponderán en cada caso con el mismo atributo del aplicativo.

* Fecha mayor que: db.PECL3.find({"fecha\_inicio" : {"$gt" :ISODate("2001-01-01 00:00.000")}})
  + $gt: *greater than* (*mayor que*)
  + $gte: *greater than or equal* (*mayor o igual que*)
  + $lt: *less than* (*menor que*)
  + $lte: *less than or equal* (*menor o igual que*)
  + $ne: *not equal* (*distinto de*)
  + $in: *in* (*dentro de*(un array)
  + $nin: *not in* (*no dentro de* (un array))
* Filtra por id: db.PECL3.find({\_id: ObjectId('5ad46a48cd730b05f4ca62a5')})
* Filtra por campo String: db.PECL3.find({id\_fun:"f001"})
* Filtro multiple string: db.PECL3.find({code:"warning", id\_dev: "dev02"})
* Filtro con selección de campos activos:

db.PECL3.find**(**

**{**id\_fun**:**"f001"**},**

**{**id\_fun**:**1**,**code**:**1**,**fecha\_inicio**:**1**,**\_id**:**0**}** //campos a mostrar

**)**.pretty**()**

* Contar totales: db.PECL3.find**({})**.count**()**
* Contar totales Log Inicios: db.PECL3.find({code:" Inicio"}).count()
* Porcentaje Inicios:

var total= db.PECL3.find**({})**.count**()**

var inicios = db.PECL3.find**({**code**:**"Inicio"**})**.count**();**

var ini = inicios / total\*100

ini

* Contar totales Log Fin: db.PECL3.find({code:" Fin"}).count()
* Porcentaje Fin:

var total= db.PECL3.find**({})**.count**()**

var final = db.PECL3.find**({**code**:**"Fin"**})**.count**();**

var fin = final / total\*100

fin

* Contar totales WARNING: db.PECL3.find({code:"warning"}).count()
* Porcentaje WARNING :

var total= db.PECL3.find**({})**.count**()**

var warning = db.PECL3.find**({**code**:**"warning"**})**.count**();**

var war= warning / total\*100

war

* Contar totales ERROR: db.PECL3.find({code:"error"}).count()
* Porcentaje ERROR :

var total= db.PECL3.find**({})**.count**()**

var errors = db.PECL3.find**({**code**:**"error"**})**.count**();**

var err= errors / total\*100

err

* Duración media por tipo de código:

db.PECL3.aggregate**([**

**{**

$group**:** **{** \_id**:** "$code"**,** duracion media**:** **{** $avg**:** "$Duracion" **}}**

**}])**

* Agrupación por desarrollador, muestra código, duración total, media y cantidad

db.PECL3.aggregate**(** **[**

**{** $match**:** **{** id\_dev**:** "dev01" **}** **},**

**{** $group**:** **{** \_id**:** "$code"**,** d\_total**:** **{** $sum**:** "$Duracion" **},**

avg**:** **{** $avg**:** "$Duracion" **},**

count**:** **{** $sum**:** 1 **}}**

**}])**

* Agrupación por funciones, duración media de ejecución y cantidad de ejecuciones por función

db.PECL3.aggregate**(** **[**

**{** $group**:** **{** \_id**:** "$id\_fun"**,** d\_total**:** **{** $sum**:** "$Duracion" **},**

avg**:** **{** $avg**:** "$Duracion" **},**

count**:** **{** $sum**:** 1 **}}**

**}])**