

# ¿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

Tareas programadas (Quartz)  
 Gestor documental (Alfresco)  
 Inversión de control (Spring)

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

BPM (jBPM o Bonita)  
 Generación de informes (JasperReport)  
 ESB (Open ESB)



Entra en Adictos a través de

E-mail

Contraseña

Entrar

[Regístrate](#)  
[Olvidé mi contraseña](#)

[Inicio](#) [Quiénes somos](#) [Formación](#) [Comparador de salarios](#) [Nuestros libros](#) [Más](#)

» Estás en: [Inicio](#) [Tutoriales](#) [Introducción a Apache Storm](#)



Juan Alonso Ramos

Consultor tecnológico de desarrollo de proyectos informáticos.

Ingeniero en Informática, especialidad en Ingeniería del Software

Puedes encontrarme en [Autentia](#): Ofrecemos de servicios soporte a desarrollo, factoría y formación

Somos expertos en Java/J2EE



[Ver todos los tutoriales del autor](#)



Fecha de publicación del tutorial: 2014-09-22

Tutorial visitado 1 veces [Descargar en PDF](#)

## Introducción a Apache Storm

### 0. Índice de contenidos.

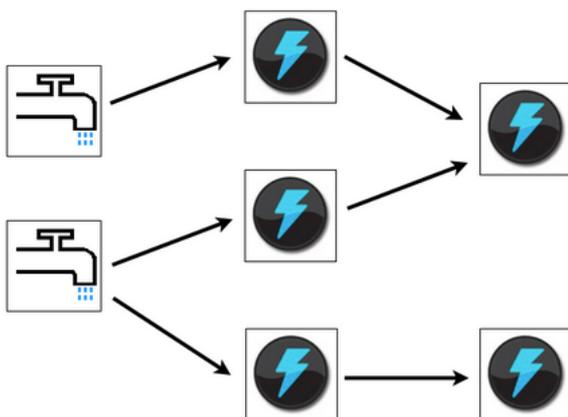
- 1. Introducción.
- 2. Entorno.
- 3. Componentes de un cluster de Storm.
  - 3.1 Modos de funcionamiento.
  - 3.2 Topología.
  - 3.3 Spout.
  - 3.4 Bolt.
  - 3.5 Stream grouping.
- 4. Implementar un contador de palabras recogidas de Twitter con Storm.
- 5. Conclusiones.

### 1. Introducción.

Apache Storm es un sistema que sirve para recuperar streams de datos en tiempo real desde múltiples fuentes de manera distribuida, tolerante a fallos y en alta disponibilidad. Storm está principalmente pensado para trabajar con datos que deben ser analizados en tiempo real, por ejemplo datos de sensores que se emiten con una alta frecuencia o datos que provengan de las redes sociales donde a veces es importante saber qué se está compartiendo en este momento.

Se compone de dos partes principalmente. La primera es la que se denomina Spout y es la encargada de recoger el flujo de datos de entrada. La segunda se denomina Bolt y es la encargada del procesado o transformación de los datos.

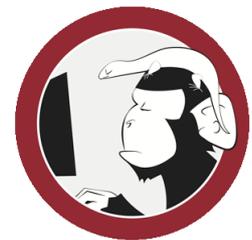
En la documentación oficial representan los Spouts con grifos simulando la entrada de un stream de datos al sistema y a los Bolts con un rayo que es donde se realizan las acciones pertinentes con los datos de entrada.



Uno de los puntos fuertes de Storm es que podemos crear una topología donde añadimos instancias de Bolts y Spouts para que escale el sistema desplegándola en el cluster de Storm que es quién se encargará de particionar los datos de entrada y redistribuirlos por los diferentes componentes.

Puedes descargar el código del tutorial desde [aquí](#).

### Catálogo de servicios Autentia



### Síguenos a través de:



### Últimas Noticias

» [Curso JBoss de Red Hat](#)

» Si eres el responsable o líder técnico, considérate desafortunado. No puedes culpar a nadie por ser gris

» [Portales, gestores de contenidos documentales y desarrollos a medida](#)

» [Comentando el libro Start-up Nation, La historia del milagro económico de Israel, de Dan Senor & Salu Singer](#)

» [Screencasts de programación narrados en Español](#)

[Histórico de noticias](#)

### Últimos Tutoriales

» [Canon AX10: Una cámara de video para profesionales y aficionados.](#)

» [Cómo crear tu CV en formato europeo](#)

» [Primeros pasos con Elasticsearch](#)

» [iTunes Backup: Copia de seguridad en disco externo](#)

» [Emmet.io - el toolkit esencial para los desarrolladores Web](#)

## 2. Entorno.

El tutorial se ha realizado con el siguiente entorno:

- MacBook Pro 15' (2.4 GHz Intel Core i5, 8GB DDR3 SDRAM).
- Oracle Java SDK 1.7.0\_60
- Apache Storm 0.9.2-incubating
- Twitter4j 4.0.1

## 3. Componentes de un cluster de Storm

Un cluster de Storm es muy parecido a un cluster Hadoop. El equivalente al Job MapReduce en Hadoop sería el concepto de topología en Storm. La principal diferencia es que mientras un Job MapReduce termina cuando finaliza la tarea, una topología se queda esperando datos de entrada eternamente, mientras no mates el proceso claro está.

La arquitectura de Storm es bastante sencilla, se divide en los siguientes componentes:

- El **master node** ejecuta el demonio llamado Nimbus responsable de distribuir el código a través del cluster (similar al JobTracker de Hadoop). Realiza también la asignación y monitorización de las tareas en las distintas máquinas del cluster.
- Los **worker nodes** ejecutan el demonio Supervisor encargado de recoger y procesar los trabajos asignados en la máquina donde corre. Estos nodos ejecutan una porción de la topología para que así se puedan distribuir los trabajos a lo largo del cluster. Si fallara un worker node el demonio Nimbus se daría cuenta y redirigiría el trabajo a otro worker node.
- **Zookeeper**: aunque no es un componente como tal de Storm, si que es necesario montar un [Apache Zookeeper](#) ya que será el encargado de la coordinación entre el Nimbus y los Supervisors. También es el encargado de mantener el estado ya que el Nimbus y los Supervisors son stateless.

### 3.1 Modos de funcionamiento

Storm puede funcionar en dos modos: local y cluster. El modo local es muy útil para probar el código desarrollado en la topología de Storm ya que corre en una única JVM por lo que podemos hacer pruebas integradas de nuestro sistema, depurar código, etc. y así poder ajustar los parámetros de configuración. En este modo Storm simula con threads los distintos nodos del cluster.

El modo cluster de Storm como su propio nombre indica, ejecuta la topología en cluster, es decir distribuye y ejecuta nuestro código en las distintas máquinas. Es el considerado 'modo producción'.

### 3.2 Topología

Una topología en Storm es similar a un grafo. Cada nodo se encarga de procesar una determinada información y le pasa el testigo al siguiente nodo. Esto se configura previamente en la topología. La topología se compone de Spouts y Bolts.

Construimos la topología con la clase TopologyBuilder:

```
final TopologyBuilder builder = new TopologyBuilder();
```

### 3.3 Spout

El componente Spout de Storm es el encargado de la ingesta de los datos en el sistema, por ejemplo si tenemos que leer un fichero de texto y contar las palabras, el componente que recibiría los streams del fichero sería el Spout. Otro típico ejemplo sería Twitter. Si queremos recoger determinados tweets para posteriormente procesarlos o realizar algún tipo de analítica sobre ellos, el encargado de conectar con el API de Twitter y recoger los datos sería el Spout.

Añadimos el Spout a la topología:

```
builder.setSpout("mySpout", new MySpout());
```

### 3.4 Bolt

El Bolt es encargado de consumir las tuplas que emite el Spout, las procesa en función de lo que dicte el algoritmo que programamos sobre los streams de entrada y puede emitirlos a otro Bolt. Es recomendable que cada Bolt realice una única tarea. Si necesitamos realizar varios cálculos o transformaciones sobre los datos que le llegan al Bolt, lo mejor es que se dividan en distintos Bolt para mejorar la eficiencia y la escalabilidad. Tanto el Spout como el Bolt emiten tuplas que serán enviadas a los Spouts que estén suscritos a ese determinado stream configurado en la topología. Por ejemplo si queremos contar las veces que aparece cada palabra en un texto, podemos hacer un Bolt que se encargue de contar las que empiezan por vocal y otro para las que empiezan por consonante. El Spout sería el encargado de redirigir a uno u otro Bolt

Añadimos el Bolt a la topología:

```
builder.setBolt("myBolt", new MyBolt()).shuffleGrouping("mySpout");
```

La forma en que se va a hacer la compartición de streams entre spout y bolt es de forma aleatoria ya que hemos definido el stream grouping como shuffleGrouping. Esto se explica con más detalle en el siguiente punto.

También podemos indicarle el número de bolts paralelos que se quieren lanzar. Por ejemplo queremos 10:

```
builder.setBolt("myBolt", new MyBolt(), 10).shuffleGrouping("mySpout");
```

### 3.5 Stream grouping

Un aspecto importantísimo de Storm es la forma en que se van a compartir los datos entre los distintos componentes por lo que la elección de la topología es crucial, a esto se le denomina stream groupings. Como modelo de datos Storm utiliza tuplas que básicamente son listas de valores con un nombre específico. El valor asociado puede ser un objeto de cualquier tipo, por ello podemos pasar lo que queramos entre nodos implementando un serializador.

- **Shuffle grouping**: Storm decide de forma aleatoria la tarea a la que se va a enviar la tupla de manera que la distribución se realiza equitativamente entre todos los nodos
- **Fields grouping**: Se agrupan los streams por un determinado campo de manera que se distribuyen los valores que cumplen una determinada condición a la misma tarea. Similar a una hash.

## Últimos Tutoriales del Autor

» [Primeros pasos con Neo4j](#)

» [Testing de Hadoop con MRUnit](#)

» [Introducción a Spring Data Hadoop](#)

» [Implementar una función UDF de Apache Pig](#)

» [Primeros pasos con Apache Pig](#)

## Categorías del Tutorial

 [Big Data](#)

- **All grouping:** El stream se pasa a todas las tareas del cluster haciendo multicast.
- **Global grouping:** El stream se envía al bolt con ID más bajo.
- **None grouping:** Bastante similar a shuffle grouping donde el orden no es importante.
- **Direct grouping:** La propia tarea es la encargada de decidir a qué bolt emitir la tupla indicando el ID de ese emisor. Esta forma dota de mayor lógica de distribución en los nodos para que puedan decidir hacia donde redirigir los streams.
- **Local grouping:** Se utiliza el mismo bolt si tiene una o más tareas en el mismo proceso.

Un aspecto a tener en cuenta es que se pueden encadenar los tipos de paso de streams, es decir podemos configurar la topología para que se envíen los streams mediante un shuffle grouping y adicionalmente hacer un fields grouping por ejemplo para enviar a un determinado bolt las tuplas que cumplan una determinada hash.

#### 4. Implementar un contador de palabras recogidas de Twitter con Storm.

Para ilustrar con un ejemplo los conceptos de Storm vamos a crear una topología que haga un clásico contador de palabras. Como flujo de entrada a la topología vamos a utilizar el API de Streaming de Twitter.

La topología tendrá un Spout que se conectará a Twitter a través de la librería `Twitter4j` que nos facilita la comunicación con el API de Twitter. Este Spout recogerá los tweets que se vayan publicando y que nos sirve en streaming y se los pasará a un Bolt que recoge el texto de tweet, lo divide en palabras y lo emite al siguiente Bolt haciendo un **fields grouping** por la palabra de manera que se puedan agrupar palabras para distribuir mejor la carga y llevar el número de apariciones. Por último se va imprimiendo por pantalla en un nuevo Bolt la cuenta resultante. A medida que van entrando nuevos tweets se actualiza e imprime una nueva cuenta.

```
4.0.0
com.autentia.tutoriales
storm-word-count
0.0.1-SNAPSHOT
```

```
UTF-8
1.7
```

```
maven-compiler-plugin
```

```
  <version>${java.version}</version>
  <version>${java.version}</version>
  <encoding>${project.build.sourceEncoding}</encoding>
```

```
org.apache.storm
storm-core
0.9.2-incubating
```

```
org.twitter4j
twitter4j-core
4.0.1
```

```
org.twitter4j
twitter4j-stream
4.0.1
```

#### WordCountTopology

Configura la topología y la arranca en modo local para pruebas. Se configura cada Spout para que sea arrancado con 10 hilos de ejecución. En un cluster correspondería a una máquina dedicada a cada tarea.

```
public class WordCountTopology {

    public static void main(String... args) throws AlreadyAliveException, InvalidTopologyException {
        final TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("twitterSpout", new TweetsStreamingConsumerSpout());
        builder.setBolt("tweetSplitterBolt", new TweetSplitterBolt(), 10).shuffleGrouping("twitterSpout");
        builder.setBolt("wordCounterBolt", new WordCounterBolt(), 10).fieldsGrouping("tweetSplitterBolt", new
        builder.setBolt("countPrinterBolt", new CountPrinterBolt(), 10).fieldsGrouping("wordCounterBolt", new

        final Config conf = new Config();
        conf.setDebug(false);

        final LocalCluster cluster = new LocalCluster();
        cluster.submitTopology("wordCountTopology", conf, builder.createTopology());
    }
}
```

#### TweetsStreamingConsumerSpout

El Spout encargado de recoger de Twitter los tweets a través del API 'streaming sample' que devuelve tweets aleatorios.

Es recomendable utilizar una cola para comunicar flujos de entrada de datos multihilo con los procesos que los consumen por temas de sincronización por ello se configura una cola en memoria para guardar los tweets como paso intermedio antes de ser emitidos a los Bolts.

Se implementan los métodos 'open' para configurar el Bolt, 'nextTuple' para emitir por la topología las tuplas que se van generando, 'activate' para arrancar el spout y empezar a recibir tweets, 'declareOutputFields' para definir los campos que salen del Spout y por último el 'deactivate' y 'close' para liberar y cerrar conexiones.

```
public class TweetsStreamingConsumerSpout extends BaseRichSpout {

    private SpoutOutputCollector collector;
    private LinkedBlockingQueue queue;
    private TwitterStream twitterStream;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector collector) {
        this.collector = collector;
        this.twitterStream = new TwitterStreamFactory().getInstance();
        this.queue = new LinkedBlockingQueue();

        final StatusListener listener = new StatusListener() {

            @Override
            public void onStatus(Status status) {
                queue.offer(status);
            }

            @Override
            public void onDeletionNotice(StatusDeletionNotice sdn) {
            }

            @Override
            public void onTrackLimitationNotice(int i) {
            }

            @Override
            public void onScrubGeo(long l, long ll) {
            }

            @Override
            public void onException(Exception e) {
            }

            @Override
            public void onStallWarning(StallWarning warning) {
            }

        };

        twitterStream.addListener(listener);
    }

    @Override
    public void nextTuple() {
        final Status status = queue.poll();
        if (status == null) {
            Utils.sleep(50);
        } else {
            collector.emit(new Values(status));
        }
    }

    @Override
    public void activate() {
        twitterStream.sample();
    };

    @Override
    public void deactivate() {
        twitterStream.cleanup();
    };

    @Override
    public void close() {
        twitterStream.shutdown();
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("status"));
    }
}

```

#### **TweetSplitterBolt**

Este Bolt recoge el tweet completo emitido por el Spout, se queda con el texto del tweet y lo divide en palabras. Cada palabra la emitirá de nuevo por la topología al siguiente Bolt configurado para recoger la salida.

```
public class TweetSplitterBolt extends BaseRichBolt {

    private OutputCollector collector = null;

    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {

```

```

        this.collector = collector;
    }

    @Override
    public void execute(Tuple tuple) {
        final Status tweet = (Status) tuple.getValueByField("status");
        final String[] words = tweet.getText().split(" ");

        for (String word : words) {
            collector.emit(new Values(word));
        }
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word"));
    }
}

```

### WordCounterBolt

Este Bolt se encarga de recoger cada palabra emitida por el Bolt anterior y realizar la cuenta por el número de apariciones almacenando el estado en el mapa de palabras. El resultado lo emitirá por la topología para que lo recoja el siguiente Bolt.

```

public class WordCounterBolt extends BaseRichBolt {

    private OutputCollector collector = null;
    private Map words = null;

    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        this.collector = collector;
        this.words = new HashMap();
    }

    @Override
    public void execute(Tuple input) {
        final String word = input.getStringByField("word");
        MutableInt count = words.get(word);
        if (count == null) {
            count = new MutableInt();
        }
        count.increment();

        words.put(word, count);
        collector.emit(new Values(word, count));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        declarer.declare(new Fields("word", "count"));
    }
}

```

### CountPrinterBolt

Por último se añade a la topología un Bolt para sacar los resultados que se van registrando a la hora de contar el número de apariciones de una palabra de todos los tweets.

```

public class CountPrinterBolt extends BaseRichBolt {

    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
    }

    @Override
    public void execute(Tuple tuple) {
        final String word = tuple.getStringByField("word");
        final MutableInt count = (MutableInt) tuple.getValueByField("count");

        System.out.println(String.format("%s:%s", word, count.toString()));
    }

    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
    }
}

```

Antes de poder hacer llamadas al API de Twitter necesitamos registrar nuestra aplicación con un usuario de Twitter en la sección de [aplicaciones](#).

# Create an application

## Application details

**Name \***

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

**Description \***

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

**Website \***

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens. (If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Una vez registrada tenemos que generar las claves OAuth desde la pestaña 'API Keys' que habrá que copiar en el fichero `twitter4j.properties` en el directorio `src/main/resources` de nuestra aplicación. Twitter4j se encargará de leer en el classpath este directorio y configurar el conector para autenticarnos con nuestras credenciales:

### twitter4j.properties

```
oauth.consumerSecret=[your consumer secret]
oauth.consumerKey=[your consumer key]
oauth.accessToken=[your access token]
oauth.accessTokenSecret=[your access token secret]
```

Para arrancar la topología basta con arrancar la clase `WordCountTopology` que tiene un método `main`. Por consola deberán ir saliendo las palabras con su número de apariciones.

```
iphone:8
you:84
don't:9
this:27
everyone:4
so:33
over:6
de:86
que:70
dar:3
to:114
...
```

## 5. Conclusiones.

Hadoop es un gran sistema para el procesado de un gran volumen de datos pero no está pensado para hacerlo en tiempo real ya que tiene una alta latencia debido a las operaciones de lectura/escritura que realiza. Apache Storm está siendo una revolución para procesar grandes cantidades de información en tiempo real ya que es capaz de procesar hasta un millón de tuplas por nodo por segundo!!!

Ejemplos para el uso de Storm se me ocurren muchos, tantos como de datos dispongamos y lo que está claro es que estamos en la era de los datos!. Se pueden procesar con Storm en tiempo real los logs de nuestras aplicaciones para ver el uso que se hace de los distintos servicios y gestión de errores; para extraer información de redes sociales a través de sus APIs; recoger y procesar datos de sensores; buscadores verticales, web analytics, etc, etc.

Si estás familiarizado con los Jobs MapReduce de Hadoop encontrarás similitudes en Storm por lo que aprender esta nueva tecnología te resultará sencillo. Personalmente creo que es mucho más sencillo Storm que Hadoop.

Puedes descargarte el código del tutorial desde [aquí](#).

Espero que te haya sido de ayuda.

Un saludo.

Juan

## A continuación puedes evaluarlo:

[Regístrate para evaluarlo](#)

**Por favor, vota +1 o compártelo si te pareció interesante**

| [Share](#)

Ánimate y coméntanos lo que pienses sobre este **TUTORIAL**:

» **Regístrate** y accede a esta y otras ventajas «



Esta obra está licenciada bajo licencia [Creative Commons de Reconocimiento-No comercial-Sin obras derivadas 2.5](#)

Copyright 2003-2014 © All Rights Reserved | [Texto legal y condiciones de uso](#) | [Banners](#) | [Powered by Autentia](#) | [Contacto](#)

