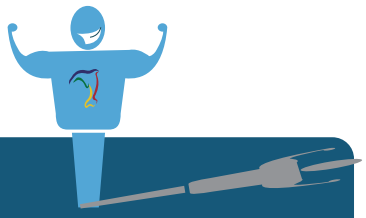
autentia

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...

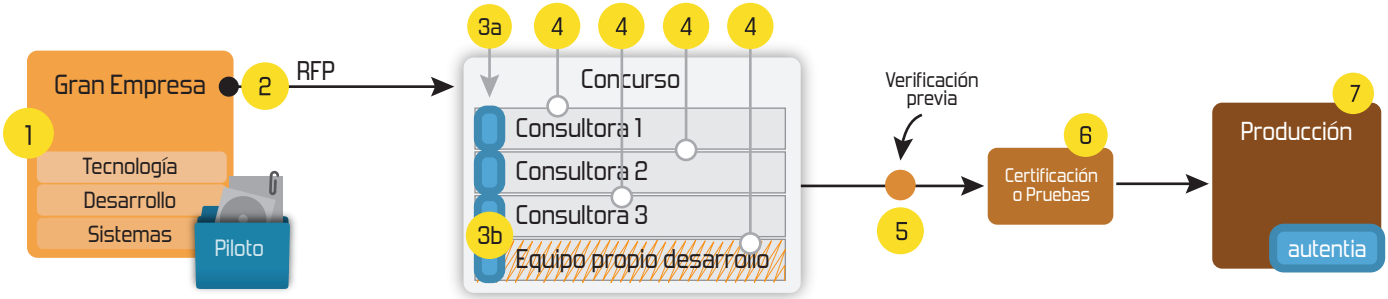
1. Desarrollo de componentes y proyectos a medida



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

Miguel Arlandy Rodríguez

Consultor tecnológico de desarrollo de proyectos informáticos.

Puedes encontrarme en Autentia: Ofrecemos servicios de soporte a desarrollo, factoría y formación

Somos expertos en Java/JEE

Ver todos los tutoriales del autor

Fecha de publicación del tutorial: 2009-02-26

Tutorial visitado 8 veces Descargar en PDF

Introducción a Apache ActiveMQ.

0. Índice de contenidos.

- 1. Introducción.
- 2. Entorno.
- 3. Características de ActiveMQ.
- 4. Instalación.
- 5. Configuración.
  - 5.1 Transport Connector.
  - 5.2 Persistencia de mensajes.
  - 5.3 Control de flujo.
  - 5.4 Uso del sistema.
- 6. Ejemplo: productor y consumidor de mensajes.
  - 6.1 Productor.
  - 6.2 Consumidor.
- 7. Referencias.
- 8. Conclusiones.

1. Introducción

El número de aplicaciones que tienen las compañías va siendo cada vez mayor. Los sistemas son cada vez más complejos y un mal diseño puede llevar a desaprovechar su potencial y generar importantes gastos por falta de flexibilidad.

En Autentia ya hemos tratado en muchas ocasiones temas como: la arquitectura SOA, Web Services, REST, ESB´s, plataformas de integración, etc... que son algunos de los actores y conceptos que nos ayudarán a diseñar sistemas flexibles con componentes con bajo nivel de acoplamiento.

Hay un componente que me parece muy interesante en este tipo de arquitecturas que es conocido como MOM (Message Oriented Middleware), que no es más que un intermediario de mensajes que usan dos o más sistemas o aplicaciones para intercambiar mensajes. Un ejemplo de MOM es ActiveMQ.

En este tutorial comentaremos las características de ActiveMQ, un intermediario de mensajes entre diferentes aplicaciones o sistemas, describiremos algunos de los escenarios donde puede ser interesante su uso, veremos cómo se instala, configura y un ejemplo de funcionamiento.

2. Entorno.

El tutorial está escrito usando el siguiente entorno:

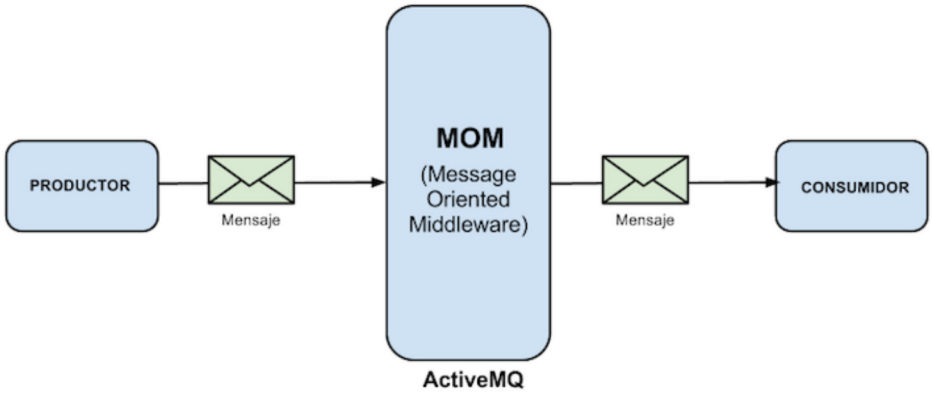
- Hardware: Portátil MacBook Pro 15' (2.2 Ghz Intel Core I7, 8GB DDR3).
- Sistema Operativo: Mac OS Snow Leopard 10.6.7
- Entorno de desarrollo: Intellij Idea 11.1 Ultimate.
- ActiveMQ 5.6.0
- MySQL 5.1

3. Características de ActiveMQ.

Entre las características de Apache ActiveMQ, me gustaría destacar las siguientes:

- Es Open Source. Se distribuye bajo Licencia Apache.

- Actúa como mediador de mensajes entre aplicaciones emisoras y receptoras.
- Proporciona comunicación asíncrona entre aplicaciones.
- Pese a ser una implementación de la especificación JMS (Java Message Service), proporciona una gran cantidad de **APIs para diferentes lenguajes** como PHP, C/C++, .Net, Ruby, etc... Lo que reduce notablemente el nivel de acoplamiento entre componentes de nuestro sistema (uno de los pilares de SOA).
- Soporta diferentes protocolos de conexión como HTTP, TCP, SSL, etc...
- Interface gráfica de administración.



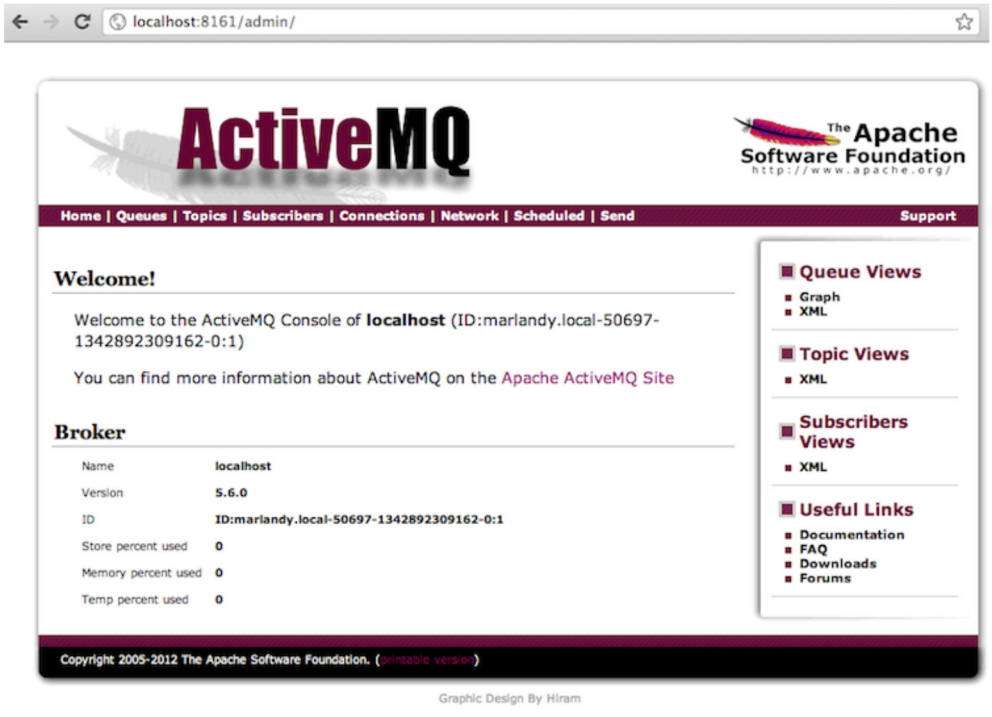
4. Instalación.

Antes de nada quiero destacar que como requisitos previos a la instalación necesitamos: Java 1.5 o superior y JAVA_HOME como variable de entorno.

La instalación de Apache ActiveMQ es muy sencilla. Lo primero que debemos hacer es descargarnos la distribución deseada desde http://activemq.apache.org/download.html. En nuestro caso lo haremos con la última versión que, a fecha de publicación de este tutorial, es la 5.6.0. Una vez descargado el fichero haremos lo siguiente.

- Descomprimimos la distribución descargada en el directorio deseado. Para entornos Unix lo haremos de la siguiente forma: **tar zxvf apache-activemq-X.X.X-bin.tar.gz**
- Comprobamos que el script de arranque de la aplicación tiene persimos de ejecución para ello en el directorio de instalación, en el directorio bin, asignamos al script "activemq" permisos de ejecución. En entorno Unix sería: **chmod 755 activemq**
- Arrancamos Apache ActiveMQ de la siguiente forma: **[directorio_instalacion_activemq]/bin/activemq start**. También podemos parar la aplicación, reiniciarla o comprobar su estado con: stop, restart o status.
- Para comprobar que nuestro intermediario de mensajes ha arrancado correctamente: **netstat -an|grep 61616**, donde 61616 es el puerto por defecto.

También podemos acceder a la consola de administración en **http://localhost:8161/admin/**



Últimos Tutoriales del Autor

» Transiciones y animaciones con CSS3

» Comparando diferencias entre ficheros con java-diff-utils

» JQuery Waypoints: realizando acciones al llegar a un punto de la página con el scroll.

» Creando un videojuego con HTML5 y Javascript

» Trabajando con Intellij IDEA 11

Categorías del Tutorial

SOA

Patrones de Integración

Para Windows el proceso de instalación es muy similar.

5. Configuración.

La configuración del gestor de mensajes ActiveMQ se basa en el fichero **activemq.xml** situado en el directorio [directorio_instalacion_activemq]/conf/. No obstante, se puede cambiar el fichero de configuración con el que arrancar ActiveMQ de la siguiente forma:

```
> [directorio_instalacion_activemq]/bin/activemq start xbean:[nombre_del_fichero_configuracion].x
```

A continuación veremos distintos comportamientos que podemos configurar en el fichero activemq.xml

5.1 Transport Connector.

Define la conexión con el broker de mensajería. Se configura en el tag **transportConnectors** (dentro del tag broker) de activemq.xml de la siguiente forma:

```
<broker>
    ...
    <transportConnectors>
        <transportConnector name="openwire" uri="uri_en_función_de_tipo_de_conexion"/>
    </transportConnectors>
    ...
</broker>
```

Se pueden definir varios tipos de conexiones dependiendo del protocolo de transporte (ver lista completa en http://activemq.apache.org/configuring-transports.html).

A continuación se muestran dos ejemplos por vm y tcp:

VM Transport permite la conexión con un cliente Java que corra bajo la misma JVM. Se define de la siguiente forma: **vm://nombre_del_broker?transportOptions**, donde:

- nombre_del_broker: es el atributo brokerName del tag broker del activemq.xml
- transportOptions: distintas opciones de la comunicación. Se puede ver la lista de opciones en http://activemq.apache.org/vm-transport-reference.html

Ejemplo:

```
<broker brokerName="mibroker">
    ...
    <transportConnectors>
        <transportConnector name="openwire" uri="vm://mibroker?marshal=false&broker.persistent=fals
    </transportConnectors>
    ...
</broker>
```

TCP Transport permite la conexión entre el cliente y el gestor de mensajería mediante protocolo TCP. Se define de la siguiente forma: **tcp://nombre_o_ip_de_la_maquina:puerto?options**, donde:

- nombre_o_ip_de_la_maquina: máquina o IP donde reside ActiveMQ
- puerto: puerto de la máquina donde escucha ActiveMQ (por defecto suele ser el 61616)
- options: opciones de la conexión. Se puede ver la lista entera de opciones en http://activemq.apache.org/tcp-transport-reference.html

Ejemplo:

```
<broker>
    ...
    <transportConnectors>
        <transportConnector name="openwire" uri="tcp://localhost:61616?trace=false&soTimeout=60000"
    </transportConnectors>
    ...
</broker>
```

5.2 Persistencia de mensajes.

Define la manera de persistir los mensajes en caso de que sea necesario (ver especificación JMS).

Persistencia de colas de mensajes con KahaDB:

KahaDB es un sistema de almacenamiento propio de ActiveMQ diseñado específicamente para las necesidades de un intermediario de mensajes. Proporciona un rendimiento muy alto. Más que cualquier base de datos relacional. Es un sistema de almacenamiento en ficheros combinado con una caché en memoria.

A continuación un ejemplo de configuración de broker con presistencia con KahaDB:

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="mibroker" persistent="true"
    useShutdownHook="false" destroyApplicationContextOnStop="false">
    ...
    <persistenceAdapter>
        <kahaPersistenceAdapter directory="${activemq.base}/activemq-data" maxDataFileLength="33554
    </persistenceAdapter>
    ...
</broker>
```

Donde:

- directory: directorio donde se almacenará la información.
- maxDataFileLength: máximo número de bytes que tendrán los ficheros que compondrán KahaDB.

Persistencia de colas de mensajes en base de datos con JDBC:

ActiveMQ tiene soporte para poder almacenar mensajes en la mayoría de bases de datos. A continuación un ejemplo de configuración con MySQL:

```
<broker xmlns="http://activemq.apache.org/schema/core" brokerName="mibroker" persistent="true"
    useShutdownHook="false" destroyApplicationContextOnStop="false">
    ...
    <persistenceAdapter>
        <jdbcPersistenceAdapter dataDirectory="${activemq.base}/data" dataSource="#mysql-ds"/>
    </persistenceAdapter>
    ...
</broker>

<!-- Fuera del tag 'broker' -->
<bean id="mysql-ds" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost/activemq?relaxAutoCommit=true"/>
    <property name="username" value="usuario"/>
    <property name="password" value="passwordBBDD"/>
    <property name="poolPreparedStatements" value="true"/>
</bean>
```

Importante: es necesario añadir el driver JDBC de conexión de cualquier base de datos al classpath de ActiveMQ, para ello basta con copiarlo en el directorio [directorio_instalacion_activemq]/lib/

Más información sobre persistencia de mensajes en http://activemq.apache.org/persistence.html.

5.3. Control de flujo.

Existe un problema muy típico en ActiveMQ cuando se usan mensajes no persistentes (almacenados en memoria) y es que el buffer de almacenamiento termine con toda la RAM. Para solucionar esto existe un sistema de control de flujo, mediante el cual, es posible limitar el número de mensajes en memoria de uno o varios productores de mensajes no persistentes.

A continuación un ejemplo de configuración:

```
<broker>
    ...
    <policyEntry queue=">" producerFlowControl="true" memoryLimit="1mb">
        <pendingQueuePolicy>
            <vmQueueCursor/>
        </pendingQueuePolicy>
    </policyEntry>
    ...
</broker>
```

El valor del atributo queue igual a >, significa que este control de flujo se aplicará a cualquier cola de mensajes. Podríamos aplicar distintas políticas de control de flujo según el nombre de la cola. Ejemplo:

- queue="TEST.FOO" : para todos los mensajes de la cola TEST.FOO
- queue="TEST.>" : para todos los mensajes de las colas que comiencen por TEST.

Más info sobre control de flujo en http://activemq.apache.org/producer-flow-control.html

5.4. Uso del sistema.

Es posible realizar una configuración más genérica de determinadas partes del sistema como puede ser el uso de memoria o el límite de almacenamiento físico o temporal.

Aquí vemos ejemplo:

```
<broker>
    ...
    <systemUsage>
        <systemUsage>
            <memoryUsage>
                <memoryUsage limit="45mb"/>
            </memoryUsage>
            <storeUsage>
                <storeUsage limit="20gb"/>
            </storeUsage>
            <tempUsage>
                <tempUsage limit="150mb"/>
            </tempUsage>
        </systemUsage>
    </systemUsage>
    ...
</broker>
```

Nótese que el límite de almacenamiento no influye en bases de datos externas aunque sí en KahaDB.

6. Ejemplo: productor y consumidor de mensajes.

Vamos a ver un ejemplo de **un escenario donde podría ser interesante el uso de ActiveMQ**. Supongamos que tenemos una aplicación (o varias) con un elevado número de usuarios. En dicha aplicación queremos poder "trazar" la actividad del usuario, o lo que es lo mismo, queremos saber cómo interactúa el usuario con la aplicación para que, posteriormente, el departamento de negocio pueda explotar esa información.

Como hemos dicho, la aplicación tiene un elevado número de usuarios por lo que se decide que el responsable de procesar y almacenar la información de la actividad de los usuarios sea otra aplicación. De esta forma liberamos a la aplicación principal de carga de trabajo.

Para implementar esta solución haremos uso de ActiveMQ. Cada vez que la aplicación principal detecte una acción de un usuario (ej: cuando el usuario vaya a "Opciones de configuración") enviará un mensaje a nuestro intermediario de mensajes con la información de dicha acción.

Estos mensajes quedarán almacenados en nuestro broker de mensajería a la espera de que la aplicación que procesa los datos los consuma para su posterior tratamiento. Si NO se necesitase un tratamiento instantáneo de la información, podría ser un proceso nocturno quien se encargase de esto (se presupone que la cantidad de mensajes generados es muy elevada).

Dicho esto, vamos a ver cómo creamos un productor y un consumidor de mensajes.

6.1 El productor de mensajes.

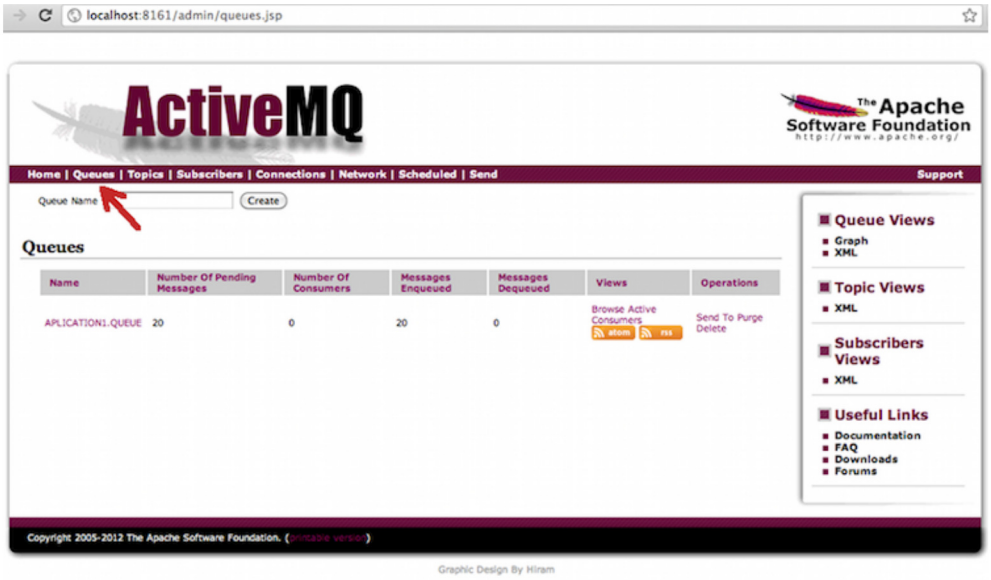
En el código que viene a continuación podemos ver cómo un emisor de mensajes envía 20 mensajes a nuestro intermediario con la información de las acciones que realizan los usuarios.

Para este ejemplo es necesaria la librería **activemq-all-X.X.X.jar**. Viene incluida en la distribución de Apache ActiveMQ.

```
import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;

import javax.jms.*;
import java.util.Random;

public class MessageSender {

    public enum UserAction {

        CONFIGURACION("IR A OPCIONES DE CONFIGURACION"),
        PORTADA("VER PORTADA"),
        LOGIN("ACCEDER A LA APLICACION"),
        SUGERENCIA("ENVIAR SUGERENCIA");

        private final String userAction;

        private UserAction(String userAction) {
            this.userAction = userAction;
        }

        public String getActionAsString() {
            return this.userAction;
        }
    }

    private static final Random RANDOM = new Random(System.currentTimeMillis());

    private static final String URL = "tcp://localhost:61616";

    private static final String USER = ActiveMQConnection.DEFAULT_USER;

    private static final String PASSWORD = ActiveMQConnection.DEFAULT_PASSWORD;

    private static final String DESTINATION_QUEUE = "APLICATION1.QUEUE";

    private static final boolean TRANSACTED_SESSION = true;

    private static final int MESSAGES_TO_SEND = 20;

    public void sendMessages() throws JMSException {

        final ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(USER, PA
        Connection connection = connectionFactory.createConnection();
        connection.start();

        final Session session = connection.createSession(TRANSACTED_SESSION, Session.AUTO_ACKNOWLE
        final Destination destination = session.createQueue(DESTINATION_QUEUE);

        final MessageProducer producer = session.createProducer(destination);
        producer.setDeliveryMode(DeliveryMode.PERSISTENT);

        sendMessages(session, producer);
        session.commit();

        session.close();
        connection.close();

        System.out.println("Mensajes enviados correctamente");
    }

    private void sendMessages(Session session, MessageProducer producer) throws JMSException {
        final MessageSender messageSender = new MessageSender();
        for (int i = 1; i <= MESSAGES_TO_SEND; i++) {
            final UserAction userActionToSend = getRandomUserAction();
            messageSender.sendMessage(userActionToSend.getActionAsString(), session, producer);
        }
    }

    private void sendMessage(String message, Session session, MessageProducer producer) throws JMS
        final TextMessage textMessage = session.createTextMessage(message);
        producer.send(textMessage);
    }

    private static UserAction getRandomUserAction() {
        final int userActionNumber = (int) (RANDOM.nextFloat() * UserAction.values().length);
        return UserAction.values()[userActionNumber];
    }

    public static void main(String[] args) throws JMSException {
        final MessageSender messageSender = new MessageSender();
        messageSender.sendMessages();
    }

}
```

Si nos fijamos en el código observamos que lo que se está haciendo es enviar un número de mensajes con acciones del usuario al azar. No es exactamente lo que hemos descrito en el punto anterior pero creo que así se entiende mejor.

Con ActiveMQ arrancado lanzamos el ejemplo (método main) y vemos en la consola de administración cómo la cola de mensajes destino (APLICATION1.QUEUE) guarda la información de los mensajes que le acabamos de enviar.

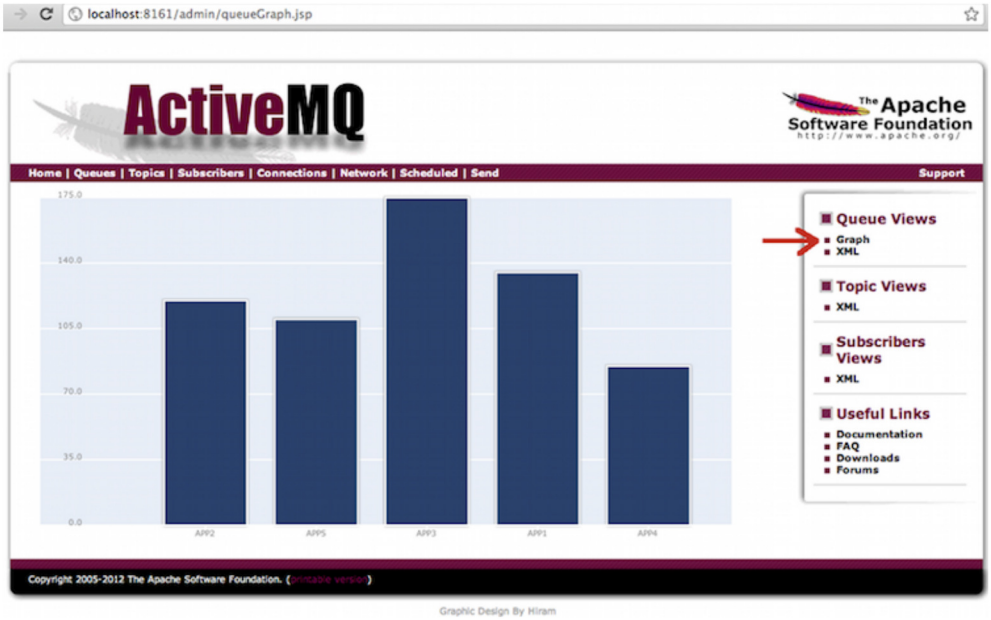


Si además tenemos configurado que los mensajes se almacenen en una base de datos relacional (en nuestro caso MySQL) podemos ver cómo en la tabla "activemq_msgs" tenemos guardados nuestros mensajes.

'tutorial_activemq.activemq_msgs' Table

	ID	CONTAINER	MSGID_PROD	MSGID_SEQ	EXPIRATION	MSG	PRIORITY
1	161	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	1	0	[0000014A 1C000000 0600017B 01002949 443A6...	0
2	162	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	2	0	[0000014A 1C000000 0700017B 01002949 443A6...	0
3	163	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	3	0	[0000013D 1C000000 0800017B 01002949 443A6...	0
4	164	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	4	0	[00000143 1C000000 0900017B 01002949 443A6...	0
5	165	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	5	0	[00000137 1C000000 0A00017B 01002949 443A6...	0
6	166	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	6	0	[0000014A 1C000000 0B00017B 01002949 443A6...	0
7	167	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	7	0	[00000137 1C000000 0C00017B 01002949 443A6...	0
8	168	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	8	0	[0000014A 1C000000 0D00017B 01002949 443A6...	0
9	169	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	9	0	[00000137 1C000000 0E00017B 01002949 443A6...	0
10	170	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	10	0	[0000014A 1C000000 0F00017B 01002949 443A6...	0
11	171	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	11	0	[0000013D 1C000000 1000017B 01002949 443A6...	0
12	172	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	12	0	[0000013D 1C000000 1100017B 01002949 443A6...	0
13	173	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	13	0	[00000143 1C000000 1200017B 01002949 443A6...	0
14	174	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	14	0	[0000014A 1C000000 1300017B 01002949 443A6...	0
15	175	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	15	0	[0000014A 1C000000 1400017B 01002949 443A6...	0
16	176	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	16	0	[00000143 1C000000 1500017B 01002949 443A6...	0
17	177	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	17	0	[00000137 1C000000 1600017B 01002949 443A6...	0
18	178	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	18	0	[00000137 1C000000 1700017B 01002949 443A6...	0
19	179	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	19	0	[0000013D 1C000000 1800017B 01002949 443A6...	0
20	180	queue://APLICATION1.QUEUE	ID:marlandy.local-53005-1342864298659-1:1:1:1	20	0	[0000014A 1C000000 1900017B 01002949 443A6...	0

En caso de que tuviésemos varias colas de mensajes (imaginemos varios productores donde cada uno envía mensajes a una cola), podemos ver la cantidad de mensajes de un vistazo con la consola de administración.



6.2 El consumidor de mensajes.

Nuestro consumidor procesará los todos mensajes que haya en la cola "a demanda" (cuando se ejecute). Si quisiésemos que procesase cada mensaje cuando llega a la cola (no es nuestro caso) deberíamos implementar la interface javax.jms.MessageListener y lanzar nuestro proceso como un hilo que se queda en espera (Thread).

```
1  import org.apache.activemq.ActiveMQConnection;
2  import org.apache.activemq.ActiveMQConnectionFactory;
3
4  import javax.jms.*;
5  import java.util.HashMap;
6  import java.util.Map;
7
8  public class UserActionConsumer {
9
```

```
10      private static final String URL = "tcp://localhost:61616";
11
12      private static final String USER = ActiveMQConnection.DEFAULT_USER;
13
14      private static final String PASSWORD = ActiveMQConnection.DEFAULT_PASSWORD;
15
16      private static final String DESTINATION_QUEUE = "APLICATION1.QUEUE";
17
18      private static final boolean TRANSACTED_SESSION = false;
19
20      private static final int TIMEOUT = 1000;
21
22      private final Map<String, Integer> consumedMessageTypes;
23
24      private int totalConsumedMessages = 0;
25
26      public UserActionConsumer() {
27          this.consumedMessageTypes = new HashMap<String, Integer>();
28      }
29
30      public void processMessages() throws JMSException {
31
32          final ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(USER, PA
33          final Connection connection = connectionFactory.createConnection();
34
35          connection.start();
36
37          final Session session = connection.createSession(TRANSACTED_SESSION, Session.AUTO_ACKNOWLE
38          final Destination destination = session.createQueue(DESTINATION_QUEUE);
39          final MessageConsumer consumer = session.createConsumer(destination);
40
41          processAllMessagesInQueue(consumer);
42
43          consumer.close();
44          session.close();
45          connection.close();
46
47          showProcessedResults();
48      }
49
50      private void processAllMessagesInQueue(MessageConsumer consumer) throws JMSException {
51          Message message;
52          while ((message = consumer.receive(TIMEOUT)) != null) {
53              proccessMessage(message);
54          }
55      }
56
57      private void proccessMessage(Message message) throws JMSException {
58          if (message instanceof TextMessage) {
59              final TextMessage textMessage = (TextMessage) message;
60              final String text = textMessage.getText();
61              incrementMessageType(text);
62              totalConsumedMessages++;
63          }
64      }
65
66      private void incrementMessageType(String message) {
67          if (consumedMessageTypes.get(message) == null) {
68              consumedMessageTypes.put(message, 1);
69          } else {
70              final int numberOfTypeMessages = consumedMessageTypes.get(message);
71              consumedMessageTypes.put(message, numberOfTypeMessages + 1);
72          }
73      }
74
75      private void showProcessedResults() {
76          System.out.println("Procesados un total de " + totalConsumedMessages + " mensajes");
77          for (String messageType : consumedMessageTypes.keySet()) {
78              final int numberOfTypeMessages = consumedMessageTypes.get(messageType);
79              System.out.println("Tipo " + messageType + " Procesados " + numberOfTypeMessages + " (
80                      (numberOfTypeMessages * 100 / totalConsumedMessages) + "%)");
81          }
82      }
83
84      public static void main(String[] args) throws JMSException {
85          final UserActionConsumer userActionConsumer = new UserActionConsumer();
86          userActionConsumer.processMessages();
87      }
88  }
```

Como vemos, lo que hace nuestro consumidor es procesar los mensajes de la cola y mostrar el número de acciones de usuario de cada tipo. Con ActiveMQ arrancado lanzamos el ejemplo y el resultado es el siguiente.

```
/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/H
Procesados un total de 20 mensajes
Tipo VER PORTADA Procesados 5 (25%)
Tipo IR A OPCIONES DE CONFIGURACION Procesados 7 (35%)
Tipo ACCEDER A LA APLICACION Procesados 3 (15%)
Tipo ENVIAR SUGERENCIA Procesados 5 (25%)

Process finished with exit code 0
```

Si consultásemos la consola de administración o la base de datos, veríamos que todos los mensajes de la cola han desaparecido puesto que ya han sido consumidos.

7. Referencias.

- Apache ActiveMQ

8. Conclusiones.

En este tutorial hemos visto el papel que puede jugar un intermediario de mensajes, en concreto ActiveMQ, en el diseño de una plataforma distribuida. Su uso está especialmente recomendado para el intercambio de mensajes entre aplicaciones de manera asíncrona.

También hemos comentado (y esto me parece muy interesante) que, aunque sea una implementación de la especificación JMS (Java Message Service), proporciona la posibilidad de que otras aplicaciones no escritas en Java puedan consumir y producir mensajes (normalmente vía Stomp).

Espero que este tutorial os haya sido de ayuda. Un saludo.

Miguel Arlandy

marlandy@autentia.com

Twitter: @m_arlandy