

APACHE
kafka®

A distributed streaming platform

Kafka
Development

Rubén Gómez García

Version 2.0.1 2020-10-23

Contenidos

1. Introducción a Kafka	1
2. Zookeeper	3
2.1. Introducción	3
2.2. Configuración	4
2.3. Ejecución	6
2.4. Operaciones de Zookeeper	7
2.4.1. Ejemplos	7
2.4.2. Creación del grupo	8
2.4.3. Unirse a un grupo	11
2.4.4. Encontrar a los miembros en un grupo	13
2.4.5. Eliminar un miembro	15
2.4.6. Eliminación de un grupo	16
3. Topics	18
4. Productores y consumidores	25
4.1. Configuración del Topic	29
5. Configuración de Producers	32
6. Configuración de Consumers	34
7. Desarrollo con kafka	35
7.1. Implementación kafka	35
7.2. Lenguaje De programación	35
7.3. Objetivo	36
7.4. Lab: Instalación del entorno de desarrollo de Kafka	37
7.4.1. Docker	37
7.4.2. Ejecución de prueba de Visual Studio Code	38
7.4.3. Atajos	39
7.4.4. Comprobación de la nueva Instalación	40
7.4.5. Preparación de proyecto plantilla	41
8. Configuración de Producers	46
9. Configuración de Consumers	48
10. Operaciones con Kafka	49
10.1. Utilidades y herramientas	49
10.1.1. kafka-preferred-replica-election.sh (deprecada)	49
10.1.2. kafka-leader-election.sh	49
10.1.3. kafka-mirror-maker.sh	50
10.1.4. kafka-replay-log-producer.sh	50
10.1.5. kafka-replica-verification.sh	50
11. Kafka Java API	52
11.1. Dependencias	52

11.2. API para Producer	52
11.3. API para Consumer	57
11.4. Lab: Invocando productores y consumidores.....	61
11.4.1. Creación de productor.....	62
11.4.2. Simple Consumer	66
11.4.3. Prueba de ejecución.....	68
11.4.4. PartitionProducer	69
11.4.5. Groups	72
11.4.6. Autocommit	75
11.4.7. PartitionConsumer	77
11.4.8. SeekConsumer	79
11.4.9. Consumer Info	80
12. Esquemas en Kafka	83
12.1. Tipos de serialización en kafka	83
12.1.1. Ejemplos Esquemas-IDL	84
12.2. Avro	84
12.3. Esquemas de Avro	85
12.4. Tipos de datos	85
12.4.1. Compactación	86
12.4.2. Integración	86
12.5. Avro y Java	89
12.5.1. Avro Tools	89
12.5.2. Maven	89
12.5.3. Lab: Generando las clases avro	91
12.6. Esquemas en Avro	102
12.6.1. Lab: Esquemas en Avro	105
13. Schema Registry	131
13.1. Integración de datos	131
13.2. Flujo de trabajo	132
13.3. Ejemplos	132
13.4. Lab: Schema Registry	134
13.4.1. Inicio de los servicios	134
13.4.2. Adaptación del productor	134
13.4.3. Generación del consumidor	138
13.4.4. Prueba de ejecución	142
13.4.5. Comprobación del Schema Registry	143
13.5. Estrategias de nombres	144
13.6. Lab: Estrategias de Schema Registry	145
13.6.1. Topic strategy	145
13.6.2. Record Name Strategy	147
14. Log Compaction	154

15. Kafka Streams API	159
15.1. Características	159
15.2. Streams	159
15.3. KStreams y KTables	159
15.3.1. KStream	159
15.3.2. KTable	159
15.3.3. GobalKTable	160
15.4. Ventanas (Windows)	160
15.5. Transformaciones	161
15.5.1. Transformaciones sin estado	161
15.5.2. Transformaciones con estado	164
15.6. Salida de datos	168
15.7. Estado en Streams	170
15.7.1. Transform Processor	170
15.8. Streams en java	171
15.8.1. Aproximaciones	171
15.8.2. Librerías	171
15.8.3. Ejemplo básico	172
15.9. Configuración	172
15.10. Creación de Serde	173
15.11. Topology	174
15.12. Serdes personalizados	177
15.13. Branching	179
15.14. Filtering	180
15.15. Desarrollo con topologías	181
15.16. Lab: Kafka Streams en Java	182
15.16.1. Ejemplo stream básico	182
15.16.2. Uso de Avro con Kafka Streams	187
15.16.3. Topology	198
15.16.4. Stateful Stream	205
15.16.5. Uso de Join	211
15.16.6. Avro	211
16. Kafka Connect	221
16.1. Tipos	221
16.2. Modos de ejecución	221
17. Seguridad en Kafka	223
17.1. Cómo funciona la seguridad	223
17.2. Certificados, Keystores y Trustores	226
17.3. Seguridad en clientes	227
17.3.1. SASL	228
17.4. Lab: Aplicando seguridad en Kafka	229

17.4.1. Creación del keystore y la pareja de claves pública/privada	229
17.4.2. Securizando los Brokers	231
17.4.3. Securizando los Clientes	233
17.4.4. Autenticar clientes	234

Capítulo 1. Introducción a Kafka

- **Kafka** es un sistema **distribuido** para el procesamiento de **streams**, escrito en **Scala y Java**.
- El objetivo de **Kafka** es ofrecer una plataforma de baja latencia y alto rendimiento para gestionar feedings en tiempo real. Para ello, dispone de una capa de almacenamiento de tipo publicador/suscriptor altamente escalable (basado en transaction logs).
- **Kafka** permite conectar a múltiples sistemas para importar o exportar información, y ofrece un **API de Java** para procesar los streams.
- Para usar **Kafka**, es imprescindible disponer de **Zookeeper**, ya que es vital para descubrir los **brokers**, y también para guardar la configuración a nivel de **topic**
- **Kafka** es un proyecto desarrollado por **LinkedIn**.



- En 2011, **Kafka** es liberado y se convierte en un proyecto de **código abierto**, gestionado por la **Apache software foundation**



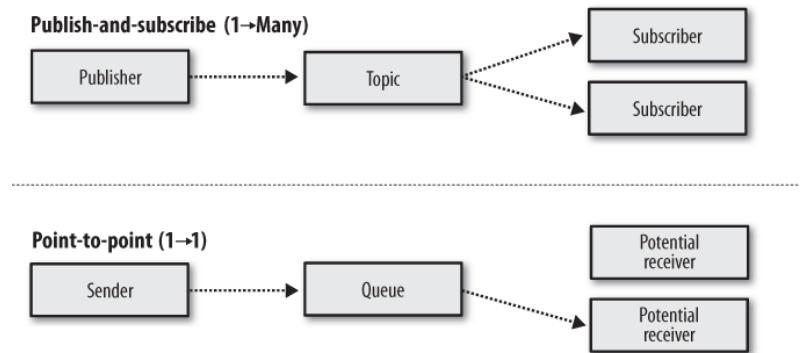
- En noviembre de 2014, varios ingenieros que trabajaban desarrollando **Kafka** en **LinkedIn**, crearon una nueva compañía llamada **Confluent**, centrada en dicho proyecto.



- El nombre del proyecto, se debe al escritor **Franz Kafka**, puesto que es un "sistema optimizado para escribir"
- **Kafka** está compuesto por tres tipos de componentes:
 - **Productores**
 - **Consumidores**
 - **Colas** (o topics)

- **Kafka** Puede usarse de dos formas:

- Como un modelo de colas, donde los mensajes son distribuidos a los clientes
- Como un modelo de publicador/suscriptor, donde el mismo mensaje es enviado a los distintos clientes



- Para ello, **Kafka** dispone de un **Broker**. Este es el servicio principal de **Kafka**, sus funciones son:
 - Almacena los distintos topics
 - Se encarga de gestionar las particiones
 - Se encarga de gestionar dónde se realizan las escrituras en disco
 - También controla la seguridad
 - Se utiliza para crear clústers y poder escalar el servicio (para ello, usa **Zookeeper**)

Capítulo 2. Zookeeper

2.1. Introducción

- ZooKeeper es un servicio de coordinación distribuido para la construcción de aplicaciones distribuidas generales.
- Escribir aplicaciones distribuidas es difícil. Es duro principalmente debido al fracaso parcial.
- ZooKeeper no puede hacer que los fallos parciales desaparezcan, ya que son intrínsecos a los sistemas distribuidos.
- Lo que hace ZooKeeper es proporcionar un grupo de herramientas para construir aplicaciones distribuidas que pueden manejar de forma segura los fallos.
- Características:
 - **Simple:** es, en su núcleo, un sistema de archivos desnudo que expone algunas simples operaciones y algunas extracciones extra, tales como ordenaciones y notificaciones.
 - **Expresivo:** las primitivas de ZooKeeper son un grupo rico de bloques de construcción que se pueden utilizar para construir una gran clase de estructuras de datos de coordinación y protocolos. Por ejemplo se incluyen colas distribuidas, bloqueos distribuidos y elección de líderes entre un grupo de compañeros.
 - **Disponible:** se ejecuta en una colección de máquinas y está diseñado para tener alta disponibilidad. Puede ayudarnos a evitar la introducción de puntos individuales de fallos en su sistema.
 - **Facilita las interacciones poco acopladas:** las interacciones se apoyan en los participantes, que no necesitan saber unos de otros, utilizando un mecanismo de encuentro.
 - **Es una librería:** que proporciona un repositorio compartido de implementaciones y recetas de código abierto y patrones comunes de coordinación. A los programadores individuales se les ahorra la carga de escribiendo protocolos comunes ellos mismos.
 - **Es de alto rendimiento.** En Yahoo!, donde se creó, el rendimiento de un grupo de ZooKeeper se ha comparado con más de 10.000 operaciones por segundo para escritura dominantes generadas por cientos de clientes.

2.2. Configuración

Los archivos de configuración se llama convencionalmente **zoo.cfg** y se coloca en el subdirectorio **conf** (normalmente se colocar en **/etc/zookeeper**, o en el directorio definido por la variable **ZOOCFGDIR**, si se establece).

Ejemplo de variables de zoo.conf

```
tickTime=2000
dataDir=/disk1/zookeeper
dataLogDir=/disk2/zookeeper
clientPort=2181
initLimit=5
syncLimit=2
server.1=zookeeper1:2888:3888
server.2=zookeeper2:2888:3888
server.3=zookeeper3:2888:3888
```

- Es un archivo de propiedades estándar de Java, y las tres propiedades siguientes son las minimas requeridas para ZooKeeper:
 - **tickTime** : es la unidad de tiempo básica en ZooKeeper (especificada en milisegundos). Se usa para los heartbeats, la sesión mínima es dos veces el tickTime
 - **dataDir** : es la ubicación local del sistema de archivos donde ZooKeeper almacena datos persistentes
 - **clientPort** : es el puerto donde escucha las conexiones del cliente (2181 es una opción común).
- Cada servidor del grupo de servidores de ZooKeeper tiene un identificador numérico que es único dentro del grupo y debe estar comprendido entre 1 y 255.
- El número de servidor se especifica en **texto sin formato** en un archivo denominado **myid** en el directorio especificado por la propiedad **dataDir**.
- También necesitamos dar a todos los servidores las identidades y ubicaciones de red de los demás servidores pertenecientes al grupo.
- En Zookeeper, el archivo de configuración debe incluir una línea para cada servidor:

Formato

```
server.n = hostname:port:port
```

- El valor de **n** se sustituye por el **número de servidor**.
- Hay dos configuraciones para los puertos:
 - **puerto que los seguidores utilizan para conectar con el líder**
 - **puerto para la elección del líder**.
- Los servidores escuchan en tres puertos:

- El **2181** para conexiones de cliente;
 - El **2888** para conexiones seguidoras, si ellos son el líder;
 - El **3888** para otras conexiones de servidor durante la fase de elección de líder.
- Cuando un servidor ZooKeeper se inicia, lee el archivo **myid** para determinar qué tipo de servidor es, y luego lee el **archivo de configuración** para determinar los puertos que debe escuchar y descubrir las direcciones de red de los otros servidores del grupo.
 - Los clientes que se conecten a este grupo ZooKeeper deben usar
 - Zookeeper1: 2181
 - Zookeeper2: 2181
 - Zookeeper3: 2181
 - Es similar a la cadena del host en el constructor para el objeto ZooKeeper.
 - En el grupo de replica (**replicaSet**), **hay dos propiedades obligatorias adicionales:**
 - **initLimit**
 - **syncLimit**
 - **Medidas múltiples de tickTime.**
 - **InitLimit**
 - Cantidad de tiempo que permite a los seguidores conectarse y sincronizarse con el líder.
 - Si la mayoría de seguidores no sincronizan dentro de este período, el líder renuncia a su liderazgo y otra elección de líder tiene lugar.
 - Si esto ocurre a menudo (se puede ver en el registro), es un signo de que la **configuración es demasiado baja**.
 - **SyncLimit** es la cantidad de tiempo que permite a un seguidor sincronizar con el líder.
 - Si un seguidor no se sincroniza dentro de este período, se reiniciará.
 - Los clientes que estuvieron vinculados a este seguidor se conectarán a otro.
 - Estas son las configuraciones mínimas necesarias para iniciar y ejecutar con un grupo de ZooKeeper Servidores.
 - Hay, sin embargo, más opciones de configuración, particularmente para ajustar rendimiento, que se documentan en la **Guía del administrador de ZooKeeper**.

2.3. Ejecución

- Para ejecutarlo, solo necesitamos usar el siguiente comando:

```
[kafka@kafka-server ~]$ zkServer.sh start
```

o bien:

1. Ejecución para la versión 3.4.14, para otras, hay que cambiar las versiones de paquetes

```
[kafka@kafka-server zookeeper-3.4.14]$ java -cp zookeeper-3.4.14.jar:lib/slf4j-  
a.25.jar:lib/slf4j-log4j12-1.7.25.jar:lib/log4j-1.2.17.jar:conf  
org.apache.zookeeper.server.quorum.QuorumPeerMain conf/zoo.cfg
```

- Para comprobar si ZooKeeper se está ejecutando, ejecutamos comando ruok ("¿Está bien?") al puerto cliente utilizando **nc** (telnet también funciona):

```
[kafka@kafka-server ~]$ echo ruok| nc localhost 2181  
imok
```

- ZooKeeper nos indica, "Estoy bien".

2.4. Operaciones de Zookeeper

- Aquí podemos observar el listado de comandos que podemos ejecutar en Zookeeper

Category	Command	Description
Server status	ruok	Prints imok if the server is running and not in an error state.
	conf	Prints the server configuration (from zoo.cfg).
	envi	Prints the server environment, including ZooKeeper version, Java version, and other system properties.
	srvr	Prints server statistics, including latency statistics, the number of znodes, and the server mode (standalone, leader, or follower).
	stat	Prints server statistics and connected clients.
	srst	Resets server statistics.
Client connections	isro	Shows whether the server is in read-only (ro) mode (due to a network partition) or read/write mode (rw).
	dump	Lists all the sessions and ephemeral znodes for the ensemble. You must connect to the leader (see srvr) for this command.
	cons	Lists connection statistics for all the server's clients.
Watches	crst	Resets connection statistics.
	wchs	Lists summary information for the server's watches.
	wchc	Lists all the server's watches by connection. Caution: may impact server performance for a large number of watches.
Monitoring	wchp	Lists all the server's watches by znode path. Caution: may impact server performance for a large number of watches.
	mntr	Lists server statistics in Java properties format, suitable as a source for monitoring systems such as Ganglia and Nagios.

Figure 1. Listado de operaciones

- Además del comando **mntr**, ZooKeeper expone estadísticas a través de JMX.



En la documentación de zookeeper explica como se exponen las estadísticas JMX

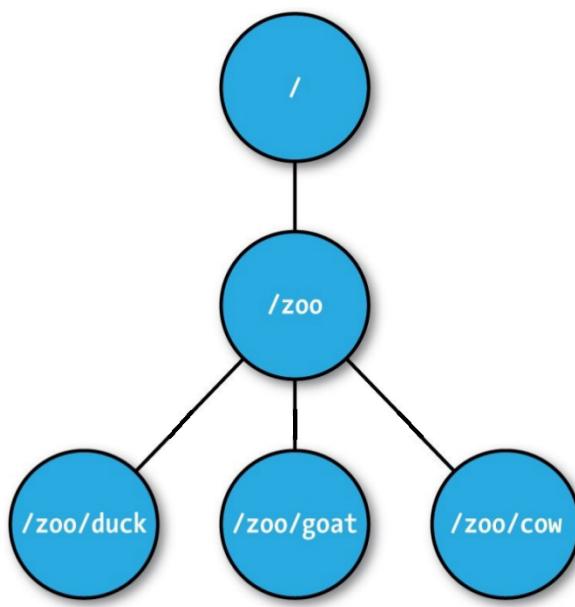
- Tenemos las herramientas de monitorización y recipientes en **src/contrib** de la distribución.
- Desde la versión 3.5.0 de ZooKeeper, hay un servidor web incorporado para proporcionar la misma información, en <http://localhost:8080/commands> para obtener una lista de los comandos de la versión.

2.4.1. Ejemplos

- Imaginemos un grupo de servidores que proporcionan algún servicio a los clientes.
- Queremos:
 - Que los clientes puedan localizar a uno de los servidores para que puedan utilizar el servicio y mantener la lista de servidores perteneciente al grupo.
 - La lista de miembros no puede almacenarse en un solo nodo de la red, ya que un fallo de ese nodo significaría la caída de todo el sistema (**alta disponibilidad**).
 - ¿Cómo eliminamos un servidor de la lista del grupo si falla?. Algunos procesos deben ser los responsables de la eliminación de servidores caídos, pero no pueden ser los servidores, ya

que no están en ejecución!

- En definitiva es una **estructura de datos distribuida activa**, y puede cambiar su estado con una entrada cuando se produce algún evento externo.
- ZooKeeper nos proporciona este servicio.
- **Pertenencia al grupo en ZooKeeper**
 - Pensemos que ZooKeeper proporciona un sistema de archivos con una alta disponibilidad.
 - No tiene archivos y directorios, sino un concepto unificado de un **nodo**, llamado **znode**, que actúa tanto como un contenedor de datos (como un archivo) y un contenedor de otros **znodes** (como un directorio).
 - Los **znodes** forman un espacio de nombres jerárquico, y la de crear la lista de miembros es un **znode padre** con el nombre del grupo y **znodes secundarios** con los nombres de los miembros del grupo (servidores).



- En una aplicación real debemos imaginarnos el almacenamiento de datos sobre los miembros, como nombres de host, con sus **znodes**.

2.4.2. Creación del grupo

- Vemos mediante el API JavaKeeper de Java la escritura de un programa para **crear un znode** de ejemplo

```

package com.kafka.zookeeper;

import java.io.IOException;
import java.util.concurrent.CountDownLatch;

import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.KeeperState;
import org.apache.zookeeper.ZooDefs.Ids;
import org.apache.zookeeper.ZooKeeper;

public class CreateGroup implements Watcher {

    private static final int SESSION_TIMEOUT = 5000;
    private ZooKeeper zk;
    private CountDownLatch connectedSignal = new CountDownLatch(1);

    public void connect(String hosts) throws IOException, InterruptedException {
        zk = new ZooKeeper(hosts, SESSION_TIMEOUT, this);
        connectedSignal.await();
    }

    @Override
    public void process(WatchedEvent event) { // Interfaz observadora
        if (event.getState() == KeeperState.SyncConnected) {
            connectedSignal.countDown();
        }
    }

    public void create(String groupName) throws KeeperException,
        InterruptedException {
        String path = "/" + groupName;
        String createdPath = zk.create(path, null/*data*/, Ids.OPEN_ACL_UNSAFE,
            CreateMode.PERSISTENT);
        System.out.println("Created " + createdPath);
    }

    public void close() throws InterruptedException {
        zk.close();
    }

    public static void main(String[] args) throws Exception {
        CreateGroup createGroup = new CreateGroup();
        createGroup.connect(args[0]);
        createGroup.create(args[1]);
        createGroup.close();
    }
}

```

- Cuando se ejecuta el método **main()**, se crea una instancia de **CreateGroup** y luego llama al método **Connect()** que instancia un nuevo objeto **ZooKeeper**, que es el objeto principal del cliente y la que **mantiene la conexión entre el cliente y el servicio ZooKeeper**.
- El constructor toma **3 argumentos**:
 - Dirección del host (y opcionalmente su puerto 2181) del servicio ZooKeeper
 - Tiempo de espera de la sesión en milisegundos (establecido a 5 segundos)
 - Una instancia de un objeto **Watcher** (observador).
 - El objeto Watcher recibe devoluciones de llamada de ZooKeeper para informarle de varios eventos.
- En este escenario, **CreateGroup** es un **Watcher**, así que pasamos esto al **constructor** de ZooKeeper.
- Cuando se crea una instancia de ZooKeeper, **se inicia un subprocesso para conectarse al servicio ZooKeeper**.
- La llamada al constructor debe volver de inmediato, por lo que es importante esperar la conexión antes de utilizar el objeto ZooKeeper.
- Se hace uso de Java **CountDownLatch** (del paquete **java.util.concurrent**) para el **bloqueo** hasta que la instancia de ZooKeeper está lista.
- Aquí es donde entra el **Watcher**.
- La interfaz **Watcher** tiene un solo método:

```
public void process(Evento WatchedEvent);
```

- Cuando el cliente se ha conectado al servidor de ZooKeeper, el **watcher** recibe una llamada a su método **process()** con un evento que indica que se ha conectado.
- Al recibir un evento de conexión (representado por **enum Watcher.Event.KeeperState**, con el valor **SyncConnected**), nosotros decrementamos el contador en **CountDownLatch**, utilizando su método **countDown()**.
- El **CountDownLatch** (cerrojo) se creó con un recuento a 1, que representa el número de eventos que deben ocurrir antes de que libere todos los hilos en espera.
- Después de llamar a **countDown()** una vez, el contador alcanzará 0 y devolverá el **método await()**.
- El método **connect()** ahora ha regresado, y el método siguiente a ser invocado es el método **create()** en la instancia de ZooKeeper.
- Los argumentos que toma son:
 - El camino (representado por una cadena recibida en **arg[1]**)
 - Contenido del znode (una matriz de bytes nula en este caso)
 - Un acceso a la lista de control (o ACL para abreviar, que aquí está completamente abierta, permitiendo que cualquier cliente las lea o escriba en el znode)
 - Naturaleza del znode que se va a crear: efímeros, **ephemeral** o persistentes, **persistent**.

- Un **znode efímero** será borrado por el servicio de ZooKeeper cuando el cliente que lo creó se desconecta, explícitamente o porque el cliente lo termina por cualquier razón.
- Un **znode persistent**, por otro lado, **NO** es eliminado cuando el cliente se desconecta ya que queremos que vida más tiempo el znode que representa que la vida del programa que lo crea.
- El valor de retorno del método **create()** es la camino creado por ZooKeeper.
- Lo imprimimos como mensaje de que la ruta de acceso se creó correctamente.
- Notaremos cómo el camino devuelto por **create()** puede diferir del pasado al método cuando vemos znodes secuenciales.
- Para ver el programa en acción, necesitamos tener ZooKeeper corriendo en la máquina local, y ejecutamos:

```
[kafka@kafka-server ~]$ export CLASSPATH=/home/kafka/Desktop/software/libs/*:$ZOOKEEPER_HOME/*:\$ZOOKEEPER_HOME/lib/*:$ZOOKEEPER_HOME/conf
[kafka@kafka-server ~]$ java com.kafka.zookeeper.CreateGroup localhost zoo
```

2.4.3. Unirse a un grupo

- Una vez creado el Grupo necesitamos registrar a un miembro en el grupo.
- **Cada miembro se ejecutará como un programa y se unirá al grupo.**
- Cuando el programa salga, debe ser eliminado del grupo, creandolo como **znode efímero** en el espacio de nombres de ZooKeeper.
- El programa **JoinGroup** implementa esta idea:

JoinGroup.java

```
package com.kafka.zookeeper;

import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.KeeperException;
import org.apache.zookeeper.ZooDefs.Ids;

public class JoinGroup extends ConnectionWatcher {

    public void join(String groupName, String memberName) throws KeeperException,
        InterruptedException {
        String path = "/" + groupName + "/" + memberName;
        String createdPath = zk.create(path, null/*data*/, Ids
        .OPEN_ACL_UNSAFE,
            CreateMode.EPHEMERAL);
        System.out.println("Created " + createdPath);
    }

    public static void main(String[] args) throws Exception {
        JoinGroup joinGroup = new JoinGroup();
        joinGroup.connect(args[0]);
        joinGroup.join(args[1], args[2]);

        // Se mantiene durmiendo hasta que el proceso sea matado o
        // interrumpido el hilo de ejecución
        Thread.sleep(Long.MAX_VALUE);
    }
}
```

- La lógica para crear y conectarse ha sido refactorizada en **ConnectionWatcher** :

```
package com.kafka.zookeeper;

import java.io.IOException;
import java.util.concurrent.CountDownLatch;

import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.Watcher.Event.KeeperState;
import org.apache.zookeeper.ZooKeeper;

public class ConnectionWatcher implements Watcher {

    private static final int SESSION_TIMEOUT = 5000;
    protected ZooKeeper zk;
    private CountDownLatch connectedSignal = new CountDownLatch(1);
    public void connect(String hosts) throws IOException, InterruptedException {
        zk = new ZooKeeper(hosts, SESSION_TIMEOUT, this);
        connectedSignal.await();
    }

    @Override
    public void process(WatchedEvent event) {
        if (event.getState() == KeeperState.SyncConnected) {
            connectedSignal.countDown();
        }
    }

    public void close() throws InterruptedException {
        zk.close();
    }
}
```

- El código para **JoinGroup** es muy similar a **CreateGroup**.
- Crea un **znode efímero** como hijo del grupo znode en su método **join()**, luego simula hacer trabajo de algún tipo durmiendo hasta que el proceso se termine por fuerza.
- Más adelante, vemos que un **znode efímero** es eliminado por ZooKeeper.

2.4.4. Encontrar a los miembros en un grupo

ListGroup

```
package com.kafka.zookeeper;

import java.util.List;

import org.apache.zookeeper.KeeperException;

public class ListGroup extends ConnectionWatcher {
    public void list(String groupName) throws
KeeperException,
        InterruptedException {
        String path = "/" + groupName;

        try {
            List<String> children = zk.getChildren(path, false);
            if (children.isEmpty()) {
                System.out.printf("No members in group %s\n",
groupName);
                System.exit(1);
            }
            for (String child : children) {
                System.out.println(child);
            }
        } catch (KeeperException.NoNodeException e) {
            System.out.printf("Group %s does not exist\n", groupName);
            System.exit(1);
        }
    }

    public static void main(String[] args) throws Exception {
        ListGroup listGroup = new ListGroup();
        listGroup.connect(args[0]);
        listGroup.list(args[1]);
        listGroup.close();
    }
}
```

- En el método **list()**, llamamos **getChildren()** con la ruta del **znode** y un indicador **watch** para recuperar una lista de rutas secundarias para el znode, que imprimimos.
- Colocamos un **watch** en un **znode** para hacer que el **watcher** quede registrado como activo si el znode cambia de estado.
- Viendo a los hijos de un znode se permitirá a un programa recibir notificaciones de los miembros que se unan o abandonan el grupo, o del grupo que se elimina.
- Capturamos **KeeperException.NoNodeException**, que se lanza en el caso cuando el **grupo del znode no existe**.
- Vemos **ListGroup** en ejecución, y comprobamos que el **grupo zoo** está vacío, ya que no tenemos agregado a ningún miembro aún:

```
[kafka@kafka-server ~]$ java ListGroup localhost zoo  
no members in group zoo
```

- Vamos usar el programa **JoinGroup** para agregar algunos miembros al grupo.
- Los lanzamos como procesos en background, ya que no terminan por sí mismos (debido a la declaración de sleep):

```
[kafka@kafka-server ~]$ java JoinGroup localhost zoo pato &  
[kafka@kafka-server ~]$ java JoinGroup localhost zoo vaca &  
[kafka@kafka-server ~]$ java JoinGroup localhost zoo cabra &  
[kafka@kafka-server ~]$ nuestra_cabra_pid = $!
```

- En la última línea nos **guardamos el ID de proceso del proceso Java que ejecuta el programa que agrega** como un miembro.
- Necesitamos recordar el ID para poder matar el proceso en un momento dado,
- comprobamos los miembros:

```
[kafka@kafka-server ~]$ java ListGroup localhost zoo  
cabra  
pato  
vaca
```

2.4.5. Eliminar un miembro

- Para ello matamos su proceso:

```
[kafka@kafka-server ~]$ kill $nuestra_cabra_pid
```

- Y unos segundos más tarde, ha desaparecido del grupo porque el proceso de la sesión de ZooKeeper ha finalizado (el tiempo de espera se ha establecido en 5 segundos)
- y su **znode efímero** ha sido eliminado:

```
[kafka@kafka-server ~]$ java ListGroup localhost zoo  
pato  
vaca
```

- Resumen
 - Sabemos como construir una lista del grupo de nodos que están participando en un sistema distribuido.
 - Los nodos **NO** tienen que tener conocimiento mutuo de su existencia.
 - Un cliente que desea utilizar los nodos de la lista para realizar algunos trabajo, por ejemplo,

puede descubrir los nodos, sin que ellos sean conscientes de su existencia.

- La pertenencia a un grupo no implica una sustitución sobre el manejo de errores de red cuando nos comunicamos con un nodo.
- Incluso si un nodo es un miembro del grupo, las comunicaciones con él pueden fallar, y tales fallos deben ser manejados de la manera habitual (reintentar, probar con un miembro del grupo, etc.).
- **Las herramientas de línea de comandos de ZooKeeper son para interactuar con el espacio de nombres de ZooKeeper.**
- Podemos usarlo para *listar los znodes bajo el znode /zoo como sigue:

```
[kafka@kafka-server ~]$ zkCli.sh -server localhost ls /zoo
vaca, pato
```

- Si lo ejecutamos sin argumentos nos muestra la ayuda.

2.4.6. Eliminación de un grupo

- La clase ZooKeeper proporciona una método **Delete()** que toma una ruta de acceso y un número de versión.
- Se eliminará un **znode** sólo si el **número de versión especificado** es el mismo que el **número de versión del znode** que es tratado de eliminar, pues se trata de un mecanismo de bloqueo optimista que permite a los clientes detectar conflictos sobre la modificación de un **znode**.
- Sin embargo, podemos omitir la comprobación de versiones mediante una versión con **valor -1** para eliminar el znode independientemente de su número de versión.

No hay ninguna operación de borrado recursivo en ZooKeeper, por lo que tenemos que eliminar los **znodes hijos antes de los padres**.

- Esto es lo que hace la clase **DeleteGroup**, que eliminará un grupo y todos sus miembros:

```

package com.kafka.zookeeper;

import java.util.List;

import org.apache.zookeeper.KeeperException;

public class DeleteGroup extends ConnectionWatcher {

    public void delete(String groupName) throws KeeperException, InterruptedException {
        String path = "/" + groupName;

        try {
            List<String> children = zk.getChildren(path, false);
            for (String child : children) {
                zk.delete(path + "/" + child, -1);
            }
            zk.delete(path, -1);
        } catch (KeeperException.NoNodeException e) {
            System.out.print("El grupo " + groupName + " no existe \n");
            System.exit(1);
        }
    }

    public static void main(String[] args) throws Exception {
        DeleteGroup deleteGroup = new DeleteGroup();
        deleteGroup.connect(args[0]);
        deleteGroup.delete(args[1]);
        deleteGroup.close();
    }
}

```

- Finalmente, podemos eliminar el grupo de zoo que creamos anteriormente:

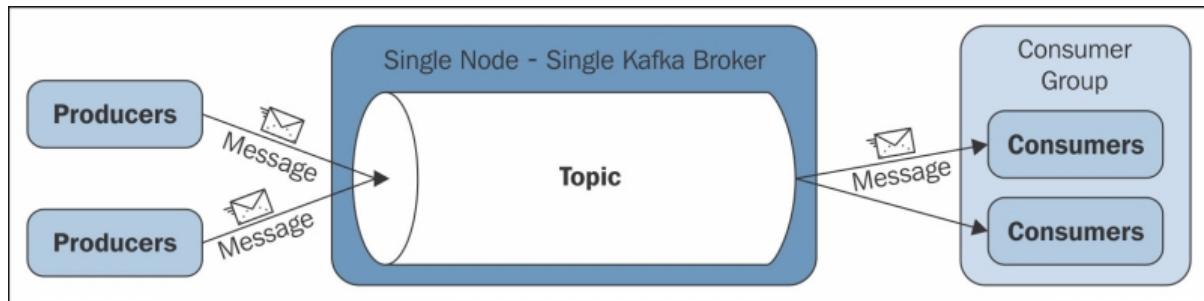
```

[kafka@kafka-server ~]$ java DeleteGroup localhost zoo
[kafka@kafka-server ~]$ java ListGroup localhost zoo
Grupo zoo no existe

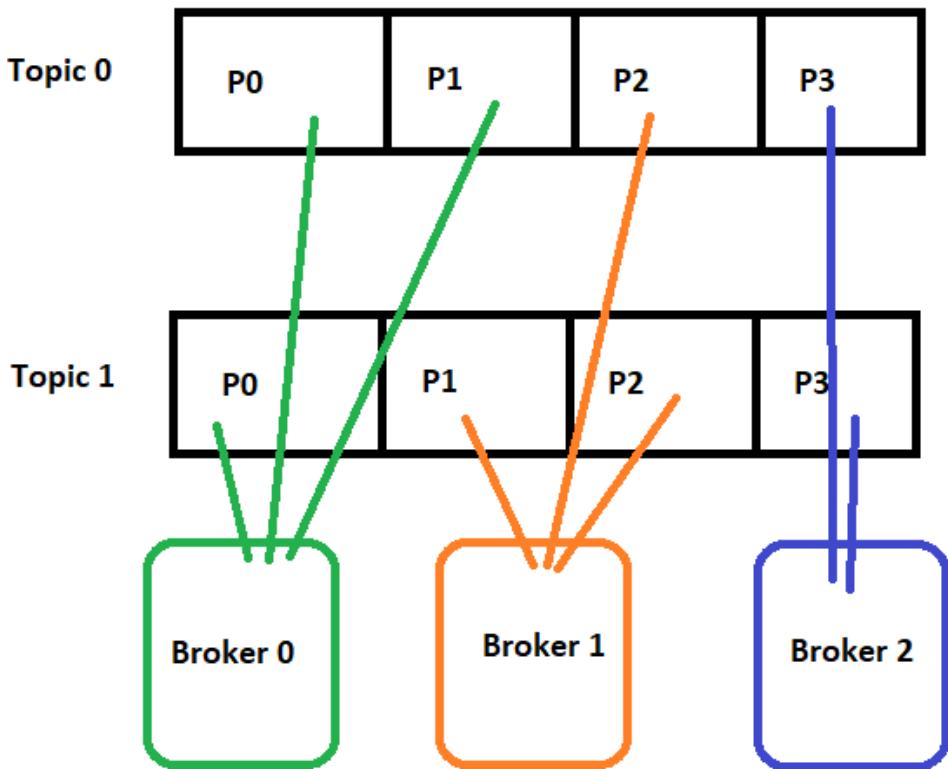
```

Capítulo 3. Topics

- Los **Topics** son la base de **Kafka**, son los equivalentes a las colas de mensajería (se inserta o leen mensajes).
- El **Broker** es el encargado de guardar las distintas colas (**topics**), se utilizan también para crear los **clusters**, y se sincronizan mediante **Zookeeper**.

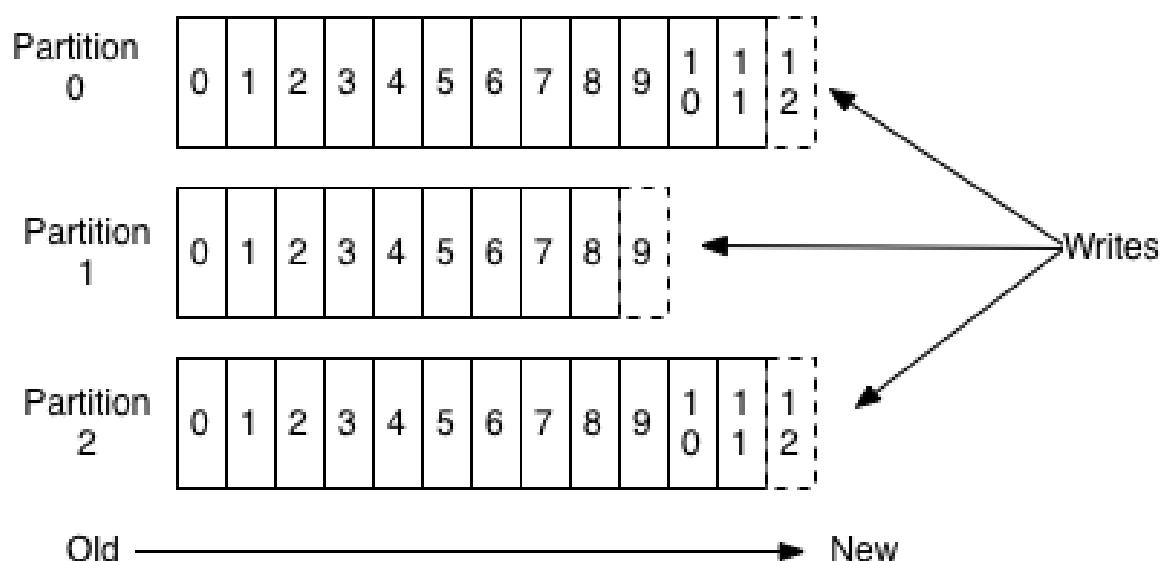


- Los **Topics** son las distintas colas de mensajes que se encuentran en **Kafka**.
- Un **Topic** está dividido en múltiples particiones. Las particiones se asignan a los distintos **Brokers** para poder distribuir y escalar el sistema (aunque un único **Broker** puede gestionar varias particiones).
- Dentro de las particiones se encuentran nuestros mensajes, que es el objeto final de nuestro sistema.

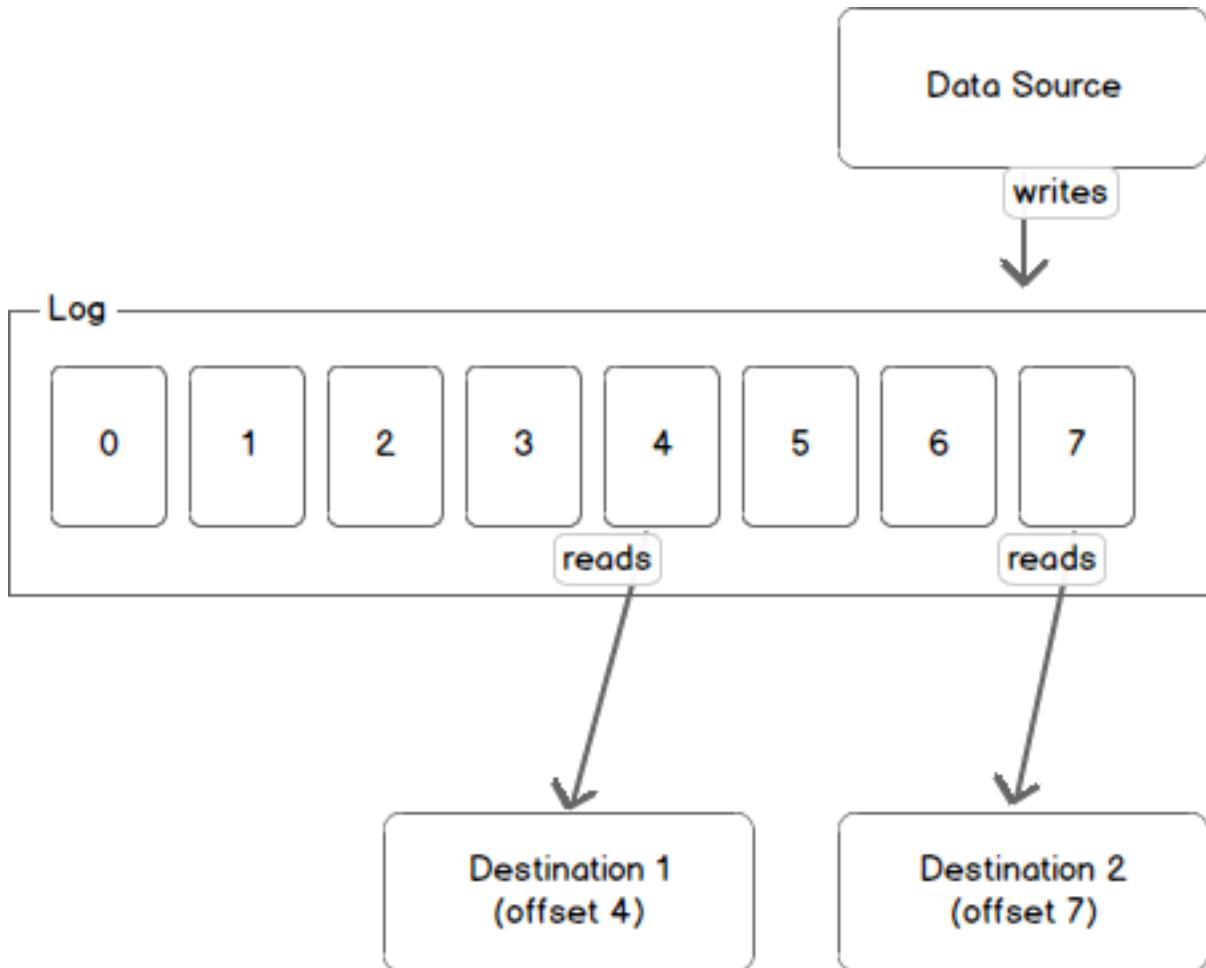


- Las particiones son ficheros que se encuentran en el disco.
- Estos ficheros se denominan **logs**.
- Cada mensaje dentro de un fichero **log** es identificado por un **offset**. Este **offset** sirve de ordenamiento, y es generado automáticamente por **kafka**.
- Los consumidores pueden leer los mensajes a partir de un **offset** específico, por lo que los consumidores pueden unirse al clúster en cualquier momento (empezando en el **offset** que consideren).
- En **Kafka**, un mensaje se identifica de manera única mediante su **topic**, su **partición** y su **offset** (dentro de dicha partición)

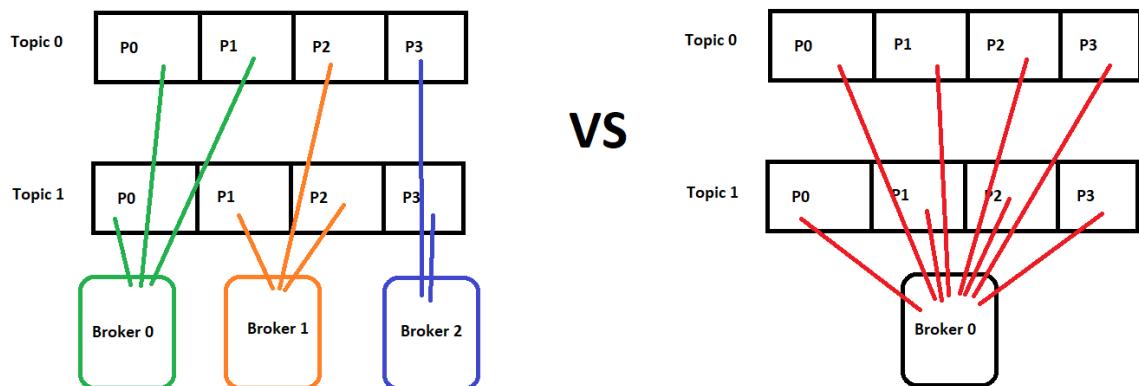
Anatomy of a Topic



- Ya hemos dicho que internamente, una partición se guarda en disco como un fichero de tipo **log**. Un productor escribe un mensaje en dicho fichero y los consumidores leen el fichero desde el **offset** que ellos quieran.
- **Kafka** mantiene estos mensajes durante un periodo de tiempo (configurable), y es el consumidor el que debe ajustarse a dicho comportamiento (e.g., si un consumidor está caído durante un tiempo mayor a dicho periodo, perderá mensajes, pero en caso contrario, podrá continuar donde lo había dejado).
- Es decir, **Kafka** no guarda información de qué ha leído cada consumidor.



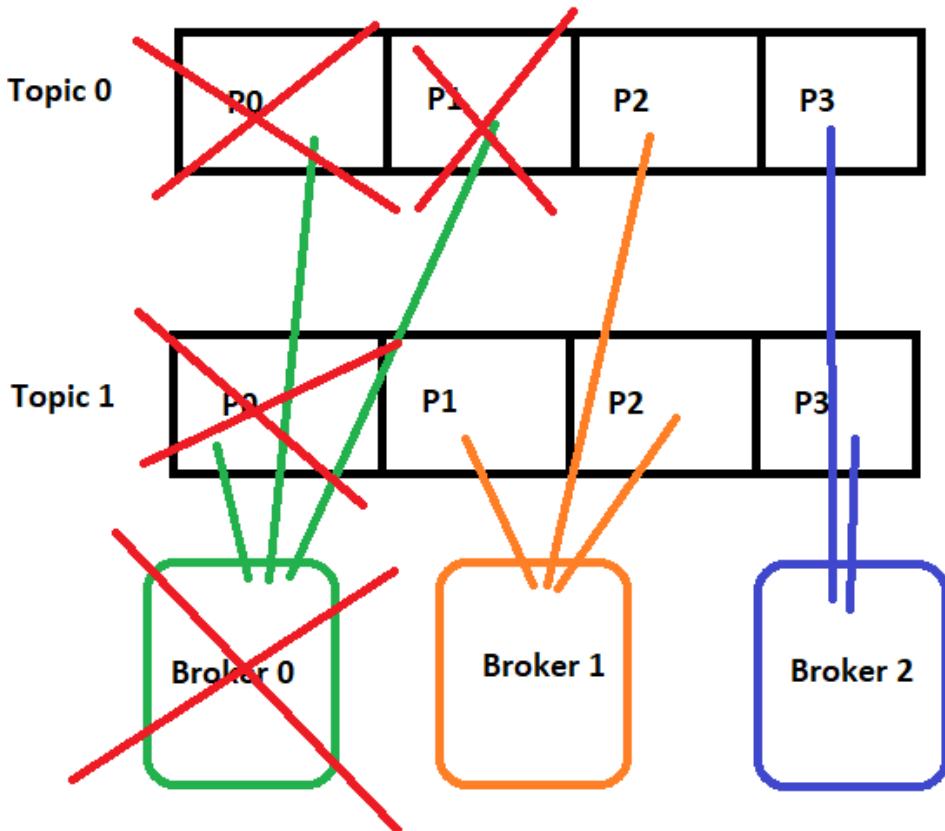
- Bueno, ya hemos dicho que podemos particionar un **Topic** para poder distribuirlo a más de un **Broker**.
- Tener en cuenta que si vuestro sistema está saturado, y sólo disponéis de un **Broker**, crear más particiones no va a solucionar el problema (ya que el **Broker** tendrá todas las particiones).
- Para poder distribuir las particiones a múltiples nodos, debemos disponer de múltiples **Brokers**



- Bien, sabemos cómo escalar si múltiples productores están escribiendo a un ritmo superior al que es capaz de gestionar un único **Broker**, pero ¿qué sucede en el caso anterior si se cae un

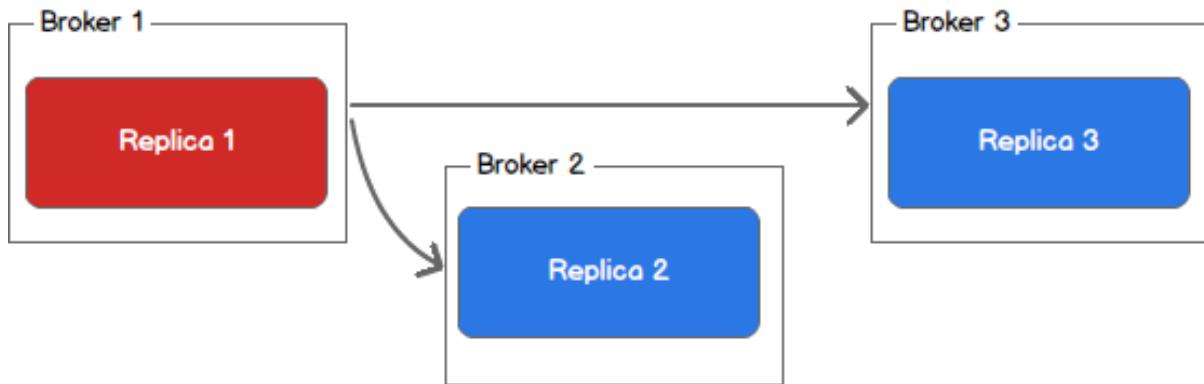
nodo?

- Si en nuestro ejemplo anterior, se cayera el nodo que gestiona el **Broker 0**, dejaríamos de tener acceso a tres particiones de dos **topics** distintos, con lo que no seríamos capaz de funcionar correctamente



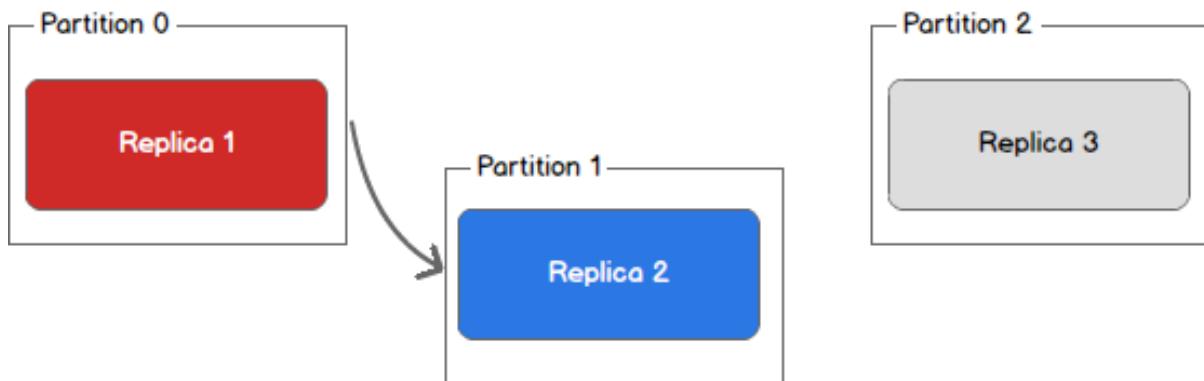
- Para asegurar la alta disponibilidad de nuestro sistema, **Kafka** permite gestionar **réplicas** de nuestras particiones.
- Una **réplica** es una copia de una partición asignada a otro **Broker** y por lo tanto ubicada en otro nodo distinto.
- Con ello, nos aseguramos de que en caso de caída de algún, podemos seguir trabajando ya que la información seguirá estando disponible.
- Cada partición de cada **Topic** tiene un único **Leader**.
- Cuando realizamos una escritura, esta se realiza siempre sobre la partición **Leader**, cuyo **Broker** se encarga de persistir el dato y de sincronizarlo con las otras réplicas (los **Brokers** que contienen las otras réplicas deben confirmar la escritura).

Leader (red) writes to replicas (blue)



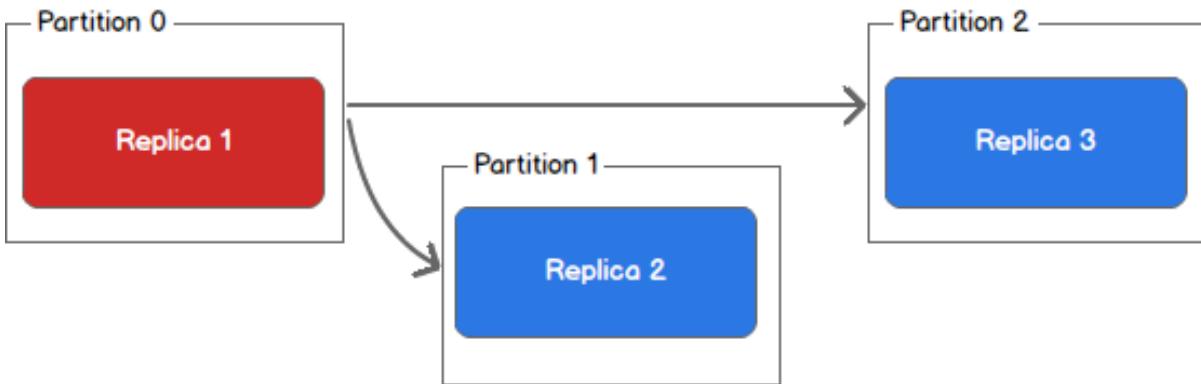
- Es importante recordar que tanto las escrituras como las lecturas se realizan siempre desde la partición **Leader**
- Cuando todo funciona correctamente, los datos se replican sin problemas. pero ¿qué sucede cuando cae un nodo?
 - Si cae un nodo que contiene una de las réplicas no líderes, estos quedarán **out of sync**, y cuando se recuperen el líder se encargará de sincronizarlas.

Leader (red) writes to live replicas (blue)



- Si el nodo que cae es el que contiene la réplica **Leader**, el controlador de **Kafka** detectará la caída del líder, y elegirá un nuevo **Leader** de las réplicas que estén sincronizadas.

Leader (red) fails

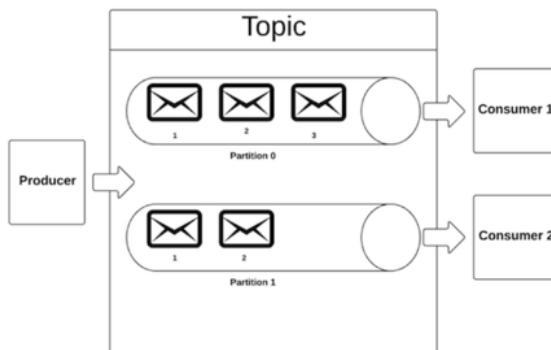


- Hay varios puntos que tenemos que tener en cuenta
 - Nunca debemos poner un número de **réplicas** superior al número de **Brokers**, ya que no tiene sentido que un único **Broker** contenga dos réplicas
 - Aumentar las réplicas aumentamos la disponibilidad, nuestro sistema es más robusto ante caídas
 - Incrementar las réplicas aumenta el consumo de red, ya que el líder debe enviar los datos a las réplicas
 - Las réplicas disminuyen el rendimiento, porque el líder debe enviar los datos a las réplicas y recibir confirmaciones antes de dar la operación por buena (se puede configurar para no esperar tantos ack)
- Por último, vamos a hablar de los mensajes.
- Los mensajes en **Kafka** están formados por tres partes:
 - Un Timestamp
 - Una Clave
 - Un valor

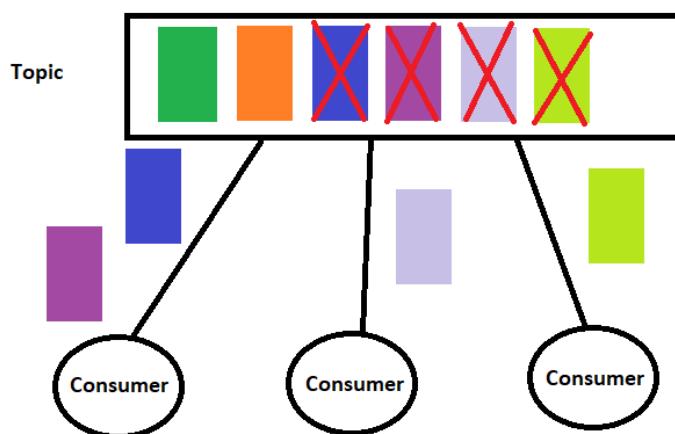
Tanto la clave como el valor pueden ser de muchos tipos (ya que son conjuntos de bytes). La Clave puede ser usada para realizar el particionado (envías un mensaje a un **Topic**, pero podemos usar la clave para conseguir que el mensaje acabe en una partición determinada).

Capítulo 4. Productores y consumidores

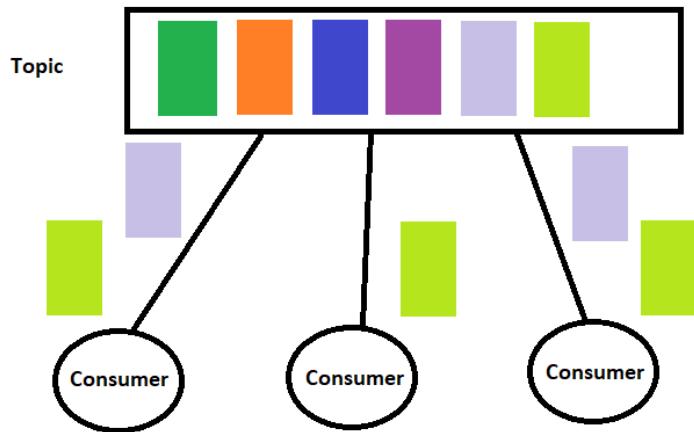
- Ya hemos hablado de que en **Kafka** existen **Productores** y **Consumidores**.
 - Los **Productores** son los que crean los mensajes y los mandan a las colas (**topics**)
 - Los **Consumidores** son los que recogen esos mensajes



- Los **consumidores** pueden funcionar de dos maneras, siguiendo un **modelo de colas** o un **modelo de publicador/suscriptor**
- En el modelo de cola, los mensajes son repartidos entre las instancias del consumidor



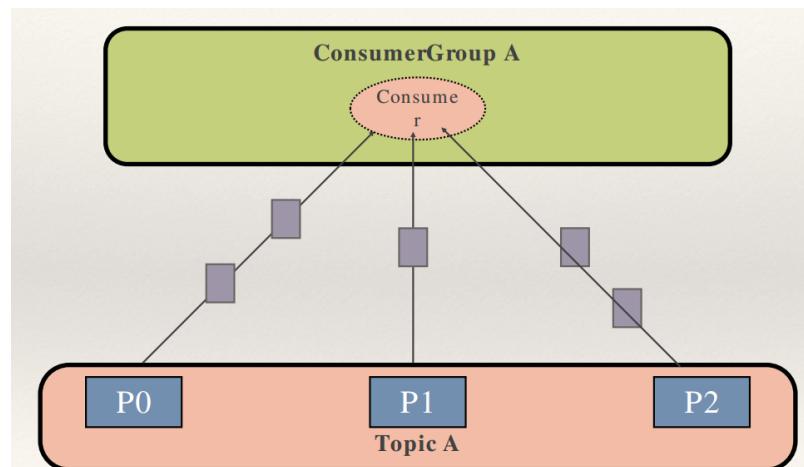
- En el modelo de publicador/suscriptor, las instancias de los consumidores recibirán todos los mismos mensajes



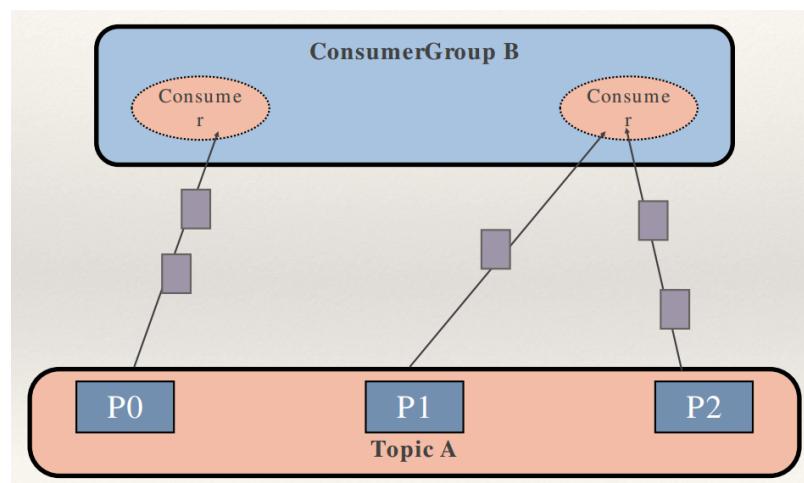
- Los **Productores** son los encargados de producir los mensajes y enviarlos a los **topics**.
- Estos mensajes deben ser enviados siempre a la partición líder, el **productor** conoce cuál es el líder de cada partición, porque al iniciarse, se conecta a uno de los **broker** y le pide el mapa del particionado.
- Tras obtener este mapa del particionado, ya sabe en qué **broker** está cada partición líder
- Una cosa importante, cada mensaje se va a guardar en una partición concreta.
- Esto, a priori, es aleatorio, aunque a veces es interesante que no hagamos un reparto aleatorio, sin conocer (o decidir) de antemano en qué partición acaba cada mensaje (por ejemplo, queremos tener todos los mensajes asociados al id de un cliente en una misma partición).
- Para saber en qué partición acaba nuestro mensaje, usaremos su **clave**
- Por temas de rendimiento, los mensajes no se envían de uno en uno, se agrupan en paquetes. Estos paquetes se definen mediante dos condiciones:
 - Por número → Si llegas a un número determinado de mensajes, envía el paquetes
 - Por tiempo → Si durante un intervalo de tiempo definido, no se llega a acumular el número de mensajes indicado, realizamos el envío igualmente.
- Por su parte, los **consumidores** serán los encargados de leer los mensajes de los distintos **topics** (pero recordar, siempre se leen de las particiones líderes).
- En versiones antiguas de **Kafka** (0.8 y anteriores), se hacía uso de **Zookeeper** para saber por dónde estaban leyendo, ahora ya no (lo gestiona **kafka** internamente).
- Para llevar esta gestión, **Kafka** hace uso de un **topic** especial denominado **_consumer_offsets**.
- Este **topic** tiene el identificador de cada **grupo de consumidores** y el **offset** por el que va leyendo.
- Un **grupo de consumidores** es un identificador compartido por varios consumidores. Un **grupo**

de consumidores puede tener 1 o varias instancias de consumidores.

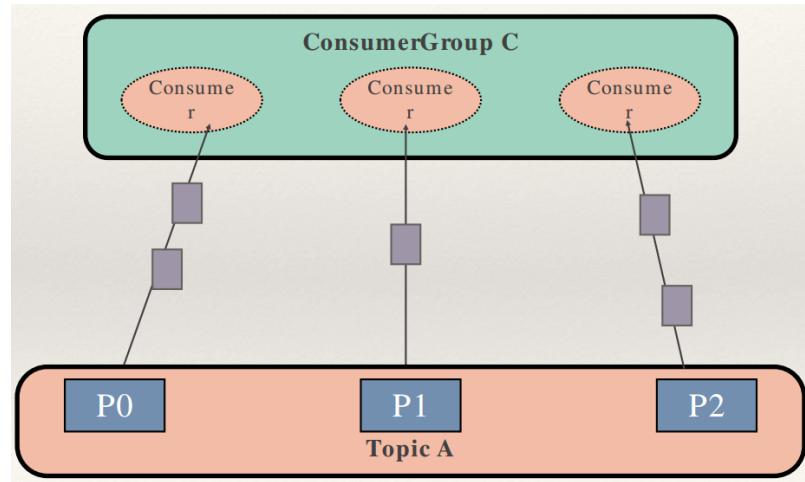
- Si hay más de uno, los consumidores balancean las particiones.
- Si algún consumidor se cae, su partición se asigna a otro consumidor.
- **No se pueden tener más consumidores que particiones**, pero si podemos tener varios **grupos de consumidores** leyendo de una misma partición (pero sólo un cliente de cada grupo puede leer de una partición concreta)
- Un **Grupo de consumidores** puede tener una única instancia de consumidor, en cuyo caso procesa los mensajes de todas las particiones del **topic**



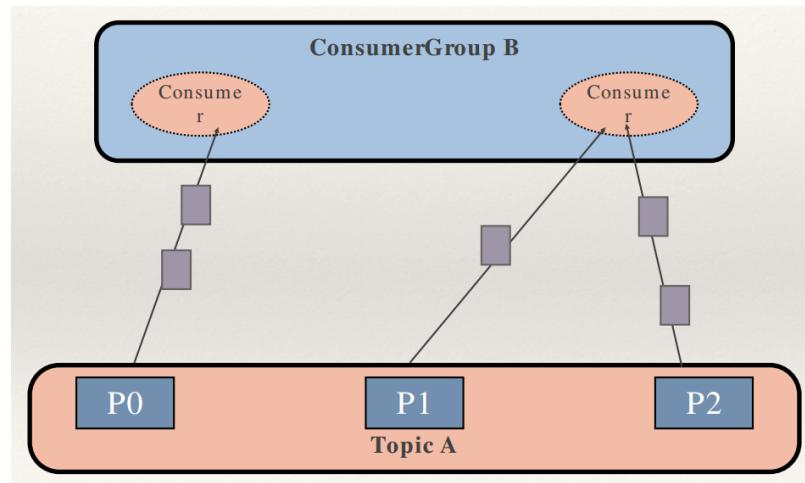
- Si para el mismo grupo de consumidores, creo una nueva instancia del consumidor, se balancean las particiones



- Con otra instancia adicional, se vuelven a balancear las particiones (recordad, no puede haber más consumidores que particiones)



- Si alguna instancia cae, la partición (o particiones) asignadas al consumidor que ha caído se reasignan a los consumidores existentes.



4.1. Configuración del Topic

- Ya hemos visto anteriormente cómo crear un **Topic**, mediante el uso del comando **kafka-topics.sh**, vamos a ver con más detalle cómo funciona el comando:

```
[kafka@kafka-server ~]$ kafka-topics.sh --bootstrap-server localhost:9092 --create  
--topic nombre-topic --partitions 4 --replication-factor 2 --config propiedad=valor
```

- Argumentos:
 - **--zookeeper host:puerto** → Es necesario indicar dónde está el servidor **Zookeeper**
 - **--create** → Indicamos que vamos a crear un **Topic**
 - **--topic nombre** → Damos nombre al **Topic** (cuidado, los guiones bajos y pueden colisionar)
 - **--partitions n** → Indicamos el número de particiones que queremos crear para dicho **Topic**
 - **--replication-factor n** → Indicamos el número de réplicas que queremos
 - **--config propiedad=valor** → permite cambiar los valores de otras propiedades
- Los **Topics** se pueden eliminar (aunque para ello, debemos tener activada en el fichero de configuración **server.properties** la propiedad **delete.topic.enable=true**, si no, lo ignora).
- Para borrar un **Topic**, usaremos la opción **--delete** del comando **kafka-topics.sh**.
- El siguiente ejemplo muestra cómo se borraría un **Topic** de nombre **topic-a-borrar**

```
[kafka@kafka-server ~]$ kafka-topics.sh --bootstrap-server localhost:9092 --delete  
--topic topic-a-borrar
```

Una vez hemos creado el **Topic**, podemos cambiar su configuración mediante la opción **--alter**, a la que debemos añadir el cambio que queramos hacer (con **--config propiedad=valor**)

```
[kafka@kafka-server ~]$ kafka-topics.sh --bootstrap-server localhost:9092 --alter  
--topic un-topic --config propiedad=valor
```

- Cuando no hemos especificado el valor de una propiedad, usa los valores por defecto para la misma. Si queremos quitar la configuración que hemos añadido y dejar sus valores por defecto, podemos usar **--alter** con la opción **--delete-config propiedad**



En versiones anteriores a Kafka 0.9, era **--deleteConfig**)

```
[kafka@kafka-server ~]$ kafka-topics.sh --bootstrap-server localhost:9092 --alter  
--topic un-topic --delete-config propiedad
```

- Para añadir particiones, podemos hacer uso de **--alter** con la opción **--partitions n**, siendo **n** el número de particiones que queremos añadir.

```
[kafka@kafka-server ~]$ kafka-topics.sh --bootstrap-server localhost:9092 --alter  
--topic un-topic --partitions 20
```

- Un punto importante es que **No se pueden disminuir particiones.**
- Disminuir particiones implicaría la pérdida de datos, y por lo tanto no se contempla esto.
- También tenemos que tener en cuenta que ampliar las particiones hace que los mensajes asignados anteriormente, puedan no estar asignados a la partición correspondientes.
- Es recomendable crear particiones de más al crear un **Topic**, para evitar estos problemas.
- Cambiar las réplicas de un **Topic** es más complicado.
- Hay que usar la utilidad **kafka-reassign-partitions.sh** y exige aportar un documento en el que especificamos para cada partición, en qué brokers queremos depositar las réplicas.
- Por ejemplo, para un **Topic** que tuviera dos particiones y factor de replicación 1, podría aumentar las réplicas del **topic** para que estuvieran en los brokers 0, 1 y 2 con el siguiente JSON:

```
{"version":1, "partitions":  
    [{"topic":"topicAlgo","partition":0, "replicas":[0,1,2]},  
     {"topic":"topicAlgo","partition":1, "replicas":[0,1,2]}]  
}
```

- Para aplicar estos cambios (imaginando que nuestro fichero se llama cambio-replicas.json), haríamos:

```
[kafka@kafka-server ~]$ kafka-reassign-partitions.sh --zookeeper localhost:2181  
--reassignment-json-file cambio-replicas.json --execute
```

- Podemos verificar que esto se ha realizado correctamente a través de la misma utilidad, con la opción **--verify** en lugar de **--execute**

```
[kafka@kafka-server ~]$ kafka-reassign-partitions.sh --zookeeper localhost:2181  
--reassignment-json-file cambio-replicas.json --verify
```

- En el tema anterior hablamos del **Log Compaction**.
- Esto, se debe configurar a nivel de **topic**, pero para poder hacer uso de ello hay que tener activado en el fichero de configuración **server.properties** la opción **log.cleaner.enable=true**.
- Podemos crear directamente un **topic** con dicha configuración mediante el parámetro **--create** con la opción **--config cleanup.policy=compact**, o bien alterar uno existente con el parámetro **--alter** con la opción **--config cleanup.policy=compact**

```
[kafka@kafka-server ~]$ kafka-topics.sh --bootstrap-server localhost:9092 --create  
--topic topicNuevo --config cleanup.policy=compact --partitions 4 --replication-factor  
3  
[kafka@kafka-server ~]$ kafka-topics.sh --bootstrap-server localhost:9092 --alter  
--topic topicExistente --config cleanup.policy=compact
```

Capítulo 5. Configuración de Producers

- Las configuraciones que vamos a ver para los **Productores** son:
 - **bootstrap.servers** → Lista de Brokers
 - **key.serializer** → Clase para serializar la clave
 - **value.serializer** → Clase para serializar el valor
 - **retries** → Reintentos de envío
 - **ack** → Número de respuestas ack que esperamos
 - **compression.type** → Tipo de compresión
 - **batch.size** → Mensajes que se acumulan antes del envío
 - **linger.ms** → Tiempo máximo de espera para envío
 - **client.id** → Identificador del cliente
 - **partitioner.class** → Clase para el particionado
- **bootstrap.servers**: La lista de **Brokers**, separados por coma, indicando los puertos en los que están escuchando. Nuestro **productor** se pondrá en contacto con un **broker** de esta lista para crear el mapa de particionados.
- **key.serializer**: La clase que se debe utilizar por este **productor** para serializar la clave del mensaje. En nuestros ejemplos serializábamos los mensajes directamente como **Strings**. Kafka posee varios serializadores ya incluídos:
 - String → org.apache.kafka.common.serialization.StringSerializer
 - Long → org.apache.kafka.common.serialization.LongSerializer
 - Integer → org.apache.kafka.common.serialization.IntegerSerializer
 - Double → org.apache.kafka.common.serialization.DoubleSerializer
 - Bytes → org.apache.kafka.common.serialization.BytesSerializer
 - ByteArray → org.apache.kafka.common.serialization.ByteArraySerializer
 - ByteBuffer → org.apache.kafka.common.serialization.ByteBufferSerializer
- Podemos crear nuestro propio serializador, implementando el interfaz **Serializer<T>** (org.apache.kafka.common.serialization.Serializer), si queremos usar nuestros propios tipos en los mensajes.
- **value.serializer**: La clase que se usa para serializar el valor de los mensajes. Al igual que en el caso anterior, podemos hacer uso de una de las clases predefinidas o implementar nuestra propia serialización
- **retries**: Número de intentos que se van a realizar de enviar un mensaje. Cuidado, podemos obtener un fallo y eso no quiere decir que nuestro mensaje no se haya enviado (sencillamente que no se nos ha confirmado), con lo que se podría dar la situación de que enviáramos el mismo mensaje (podríamos duplicar mensajes)
- **ack**: Podemos configurar nuestros **productores** para esperar un número concreto de respuestas antes de dar por buena la escritura. Algunos valores son:

- 0 → "fire and forget", no esperamos confirmación. Es el que mayor rendimiento ofrece, pero el más susceptible a errores.
 - 1 → Sólo esperamos la respuesta de la partición líder
 - all → Esperamos la respuesta de todas las réplicas
- **compression.type** → Podemos realizar compresión a la hora de enviar nuestros mensajes (aunque por defecto no se realiza). Por cierto, los topics también se pueden configurar para que guarden comprimidos los datos
 - none → No se realiza compresión
 - gzip → Compresión gzip
 - snappy → Compresión snappy
 - lz4 → Compresión lz4
 - **batch.size**: Número de mensajes que van a agruparse antes de realizar el envío. Un valor de 1 envía todos los mensajes automáticamente según se generen.
 - **linger.ms**: Tiempo máximo de espera antes del envío. Si este tiempo se cumple, los mensajes se envían aunque no hayan alcanzado el número definido por **batch.size**
 - **client.id**: Es el identificador de cliente. Aunque existan varios **productores**, si todos comparten el mismo **client.id**, **Kafka** los considera el mismo cliente. Es de especial utilidad para definir cuotas.
 - En el siguiente ejemplo vemos cómo se define una cuota de tasa de producción para el cliente "clienteAlgo" de 1024 bytes por segundo (si supera esta tasa, **kafka** relentizará el cliente para no superarla):

```
[kafka@kafka-server ~]$ kafka-configs.sh --zookeeper localhost:2181 --alter --add -config 'producer_byte_rate=1024' --entity-name clienteAlgo --entity-type clients
```

- **partitioner.class**: Por último, indicar que a la hora de enviar mensajes, estos van a acabar en una partición. Tenemos dos opciones:
 - Indicar explícitamente para cada mensaje en qué partición queremos que acabe.
 - Usar un sistema de particionado.
- **Kafka** posee un particionador por defecto (basado en la clave hash obtenida mediante murmur3), pero podemos implementar nuestro propio particionador implementando el interfaz **Partitioner** (org.apache.kafka.clients.producer.Partitioner), y por supuesto especificando su uso en **partitioner.class**.

Capítulo 6. Configuración de Consumers

- Las configuraciones que vamos a ver para los **Consumidores** son:
 - **bootstrap.servers** → Lista de Brokers (análogo a la del productor)
 - **key.serializer** → Clase para serializar la clave (análogo a la del productor)
 - **value.serializer** → Clase para serializar el valor (análogo a la del productor)
 - **group.id** → Identificador del grupo de consumidores
 - **enable.auto.commit** → Control automático del offset
 - **auto.commit.interval.ms** → Cada cuantos milisegundos se actualiza el control del offset
 - **auto.offset.reset** → Para indicar dónde empieza un cliente nuevo
 - **client.id** → Identificador del cliente
- **group.id**: Servía para identificar un grupo de consumidores. Las instancias del consumidor que posean un mismo **group.id**, se reparten las particiones para procesar los mensajes de forma escalable. Los consumidores con distintos **group.id**, leerán los mismos mensajes.
- **enable.auto.commit**: Los clientes llevan un control de los mensajes leídos, para ello sencillamente tienen que recordar el último **offset** procesado para continuar (en caso de caída o parada, cuando vuelven a ejecutarse) donde lo habían dejado. Este control puede ser manual (hacemos desde la aplicación el control y guardamos el **offset** en momentos concretos), pero normalmente se hace de forma automática. Activando esta opción, hacemos que cada cierto tiempo, se guarde información del último **offset** leído.
- **auto.commit.interval.ms**: Cada cuánto tiempo se actualiza la información de los offsets
- **auto.offset.reset**: Cuando un cliente nuevo empieza a leer mensajes por primera vez, no hay información del último **offset** procesado. Tenemos dos opciones, leer desde el principio de los tiempos, o leer sólo los nuevos mensajes
 - **smallest** → Leemos desde el primer mensaje (como cuando usábamos la opción "--from-beginning" en "kafka-console-consumer.sh")
 - **largest** → Sólo leemos los mensajes nuevos (desde que mi cliente se levantó).
- **client.id**: Es el identificador de cliente. Aunque existan varios **consumidores**, si todos comparten el mismo **client.id**, Kafka los considera el mismo cliente. Es de especial utilidad para definir cuotas.
- En el siguiente ejemplo vemos cómo se define una cuota de tasa de consumo para el cliente "clienteAlgo" de 1024 bytes por segundo (si supera esta tasa, Kafka relentizará el cliente para no superarla):

```
[kafka@kafka-server ~]$ kafka-configs.sh --zookeeper localhost:2181 --alter --add-config 'consumer_byte_rate=1024' --entity-name clienteAlgo --entity-type clients
```

- Como se puede ver, es análogo al ejemplo en el que poníamos una cuota al productor, pero en este caso al cliente le añadimos la opción "consumer_byte_rate"

Capítulo 7. Desarrollo con kafka

- Para desarrollar con kafka, hay que tener en cuenta para kafka:
 - Cual va a ser la implementación utilizada de kafka (OpenSource/Confluent)
 - Que versión vamos a utilizar (2.6.0/6.0.0)
 - Que instalación vamos a usar (Entorno centralizado/entorno particular)
 - En caso de uso de un entorno particular, como vamos a instalarlo (Instalación común/Docker)
 - Tenemos memoria suficiente para albergar todo lo que necesitamos en el equipo?
- En cuanto al desarrollo hay que tener en cuenta:
 - Que lenguaje/lenguajes de programación vamos a utilizar para comunicarnos con kafka
 - Que IDE vamos a utilizar
 - Que librerías de apoyo vamos a utilizar
 - Que arquitecturas vamos a implementar.

7.1. Implementación kafka

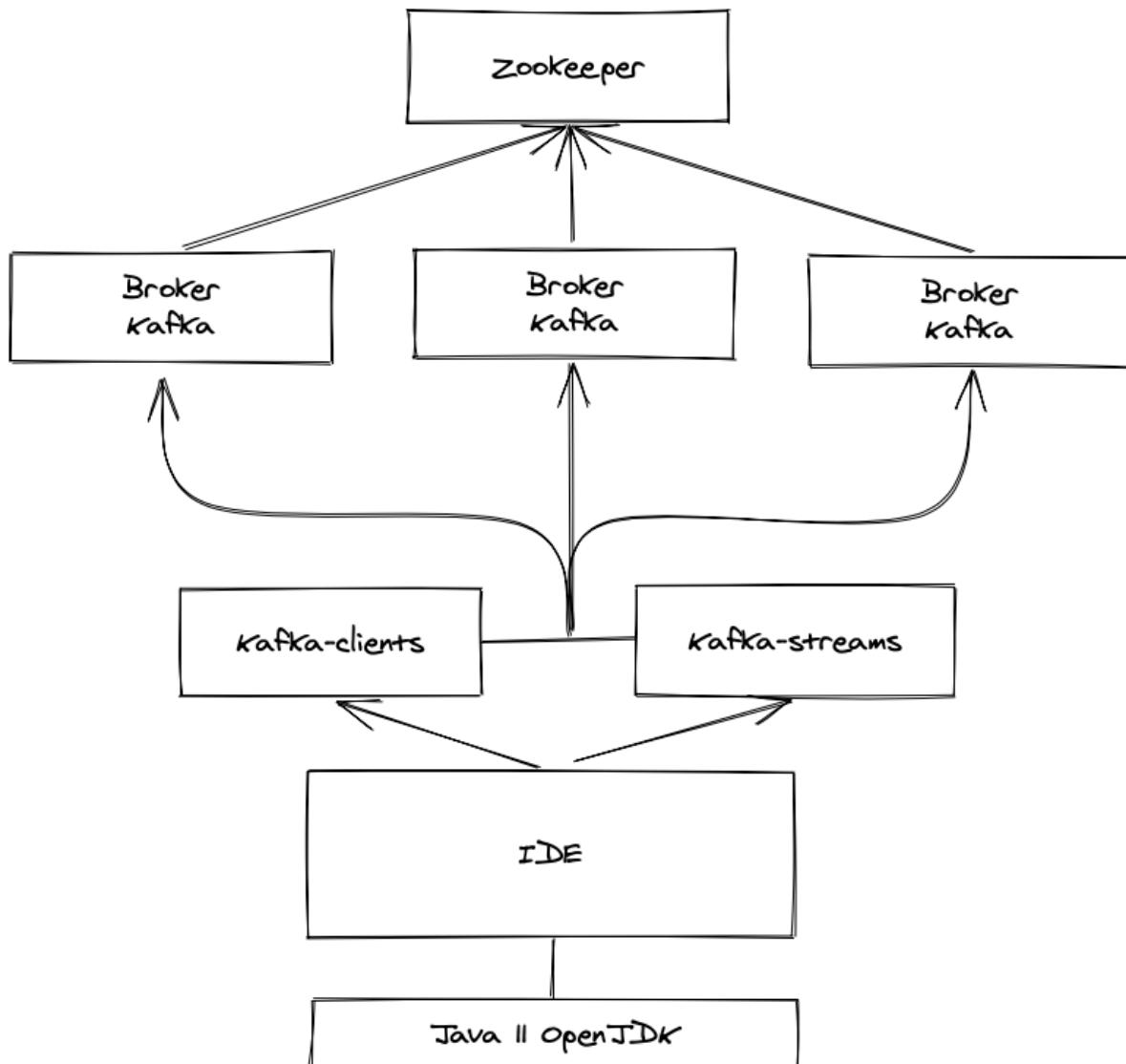
- En cuanto a la implementación de Kafka, en este curso nos centraremos en la versión OpenSource de Kafka, la versión 2.6.0
- Usaremos Docker para crear los siguientes contenedores:
 - Contenedor de Zookeeper como base de metadatos de los brokers
 - Tres contenedores de Kafka que se usarán como brokers
- Se usará la interfaz host para que se alojen en local y funcione correctamente las comunicaciones entre brokers.

7.2. Lenguaje De programación

- En nuestro caso, usaremos Eclipse o STS según lo cómodo que esté cada alumno para usarlos.
- Como lenguaje de programación, usaremos Java
- Usaremos los frameworks de:
 - kafka-clients: Para uso de conexiones a kafka
 - kafka-streams: Para el uso de streams en Java
 - io.confluent: como librerías de confluent para uso de ciertos recursos con licencia confluent.

7.3. Objetivo

- El objetivo es poseer un cluster de Kafka suficientemente versatil para poder trabajar con él
- El aspecto final será el siguiente:



7.4. Lab: Instalación del entorno de desarrollo de Kafka

- Para desarrollar en kafka vamos a realizar las siguientes operaciones.

7.4.1. Docker

- Vamos a usar docker-compose para desplegar una solución completa de Kafka.
- Para ello, vamos a generar una carpeta dentro de documents:

```
[kafka@kafka-server .]$ cd Documents  
[kafka@kafka-server Documents]$ mkdir docker-kafka  
[kafka@kafka-server Documents]$ cd docker-kafka
```

- Dentro crearemos el siguiente fichero llamado docker-compose.yml

- Podemos utilizar cualquier editor conocido para Docker, como por ejemplo visual studio code o vi.
- Si queremos abrir rápidamente visual studio code, podemos lanzar el siguiente comando, lo que lo abrirá en el mismo directorio y podremos crear ficheros nuevos en la sección izquierda del ide.



```
[kafka@kafka-server docker-kafka]$ code .
```

Contenido del fichero docker-compose.yml

```
version: "3"
services:
  zookeeper:
    image: 'bitnami/zookeeper:latest'
    ports:
      - '2181:2181'
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
    network_mode: host
  kafka1:
    image: 'bitnami/kafka:latest'
    network_mode: host
    environment:
      - KAFKA_BROKER_ID=1
      - KAFKA_LISTENERS=PLAINTEXT://:9091
      - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://127.0.0.1:9091
      - KAFKA_ZOOKEEPER_CONNECT=127.0.0.1:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
    depends_on:
      - zookeeper
  kafka2:
    image: 'bitnami/kafka:latest'
    network_mode: host
    environment:
      - KAFKA_BROKER_ID=2
      - KAFKA_LISTENERS=PLAINTEXT://:9092
      - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://127.0.0.1:9092
      - KAFKA_ZOOKEEPER_CONNECT=127.0.0.1:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
    depends_on:
      - zookeeper
  kafka3:
    image: 'bitnami/kafka:latest'
    network_mode: host
    environment:
      - KAFKA_BROKER_ID=3
      - KAFKA_LISTENERS=PLAINTEXT://:9093
      - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://127.0.0.1:9093
      - KAFKA_ZOOKEEPER_CONNECT=127.0.0.1:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
    depends_on:
      - zookeeper
```

7.4.2. Ejecución de prueba de Visual Studio Code

- Para comprobar que todo ha ido correctamente, vamos a iniciar cada uno de los contenedores de forma individual

- Primero iniciamos el servicio de Zookeeper.

```
[kafka@kafka-server docker-kafka]$ docker-compose up -d zookeeper
```

- Esperamos unos segundos e iniciamos ya los contenedores de Kafka

```
[kafka@kafka-server docker-kafka]$ docker-compose up -d kafka1
[kafka@kafka-server docker-kafka]$ docker-compose up -d kafka2
[kafka@kafka-server docker-kafka]$ docker-compose up -d kafka3
```

- Tras iniciar los contenedores, deben estar disponibles en el sistema. Lo comprobamos con el siguiente comando.

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
PORTS	NAMES			
093e9d1c649e	bitnami/kafka:latest	"./opt/bitnami/script..."	4 seconds ago	Up 3 seconds
docker-kafka_kafka1_1				
551540ef4ce4	bitnami/kafka:latest	"./opt/bitnami/script..."	3 seconds ago	Up 2 seconds
docker-kafka_kafka2_1				
fbe3387824b3	bitnami/kafka:latest	"./opt/bitnami/script..."	3 seconds ago	Up 2 seconds
docker-kafka_kafka3_1				
40c2020d1486	bitnami/zookeeper:latest	"./opt/bitnami/script..."	2 minutes ago	Up 2 mintues
docker-kafka_zookeeper_1				

7.4.3. Atajos

- Vamos a crear una serie de atajos que nos permitirán realizar operaciones muy básicas contra kafka y zookeeper
- Para ello usaremos alias.
- Primero editamos el fichero .bash_profile que se encuentra oculto en el directorio home del usuario
- Agregamos los dos alias al final del documento

Contenido completo del fichero .bash_profile

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi

# User specific environment and startup programs
alias zkCli.sh="docker exec -it docker-kafka_zookeeper_1 zkCli.sh"
alias kafka-topics.sh="docker exec -it docker-kafka_kafka1_1 kafka-topics.sh"
```

- Por último, cargamos el fichero para tener las funcionalidades listas, y que no tengamos que iniciar una nueva shell.

```
[kafka@kafka-server ~]$ source .bash_profile
```

7.4.4. Comprobación de la nueva Instalación

- Vamos a comprobar que kafka está correctamente instalado.
- Para ello, vamos a comprobar que los brokers están correctamente registrados

```
[kafka@kafka-server ~]$ zkCli.sh
[zk: localhost:2181(CONNECTED) 0] ls /brokers/ids
[1, 2, 3]
```

- El array de tres numeros corresponden a los tres ids de cada uno de los brokers kafka.
- Podemos comprobar de forma sencilla los datos de cada broker

```
[zk: localhost:2181(CONNECTED) 1] get /brokers/ids/1
{"listener_security_protocol_map":{"PLAINTEXT":{"PLAINTEXT"}, "endpoints":["PLAINTEXT://127.0.0.1:9091"], "jmx_port":-1, "host":"127.0.0.1", "timestamp":1602832313571, "port":9091, "version":4}
[zk: localhost:2181(CONNECTED) 2] get /brokers/ids/2
{"listener_security_protocol_map":{"PLAINTEXT":{"PLAINTEXT"}, "endpoints":["PLAINTEXT://127.0.0.1:9092"], "jmx_port":-1, "host":"127.0.0.1", "timestamp":1602832313698, "port":9092, "version":4}
[zk: localhost:2181(CONNECTED) 3] get /brokers/ids/3
{"listener_security_protocol_map":{"PLAINTEXT":{"PLAINTEXT"}, "endpoints":["PLAINTEXT://127.0.0.1:9093"], "jmx_port":-1, "host":"127.0.0.1", "timestamp":1602832313783, "port":9093, "version":4}
```

- Por último nos salimos del CLI del zookeeper

```
[zk: localhost:2181(CONNECTED) 4] quit
WATCHER:::

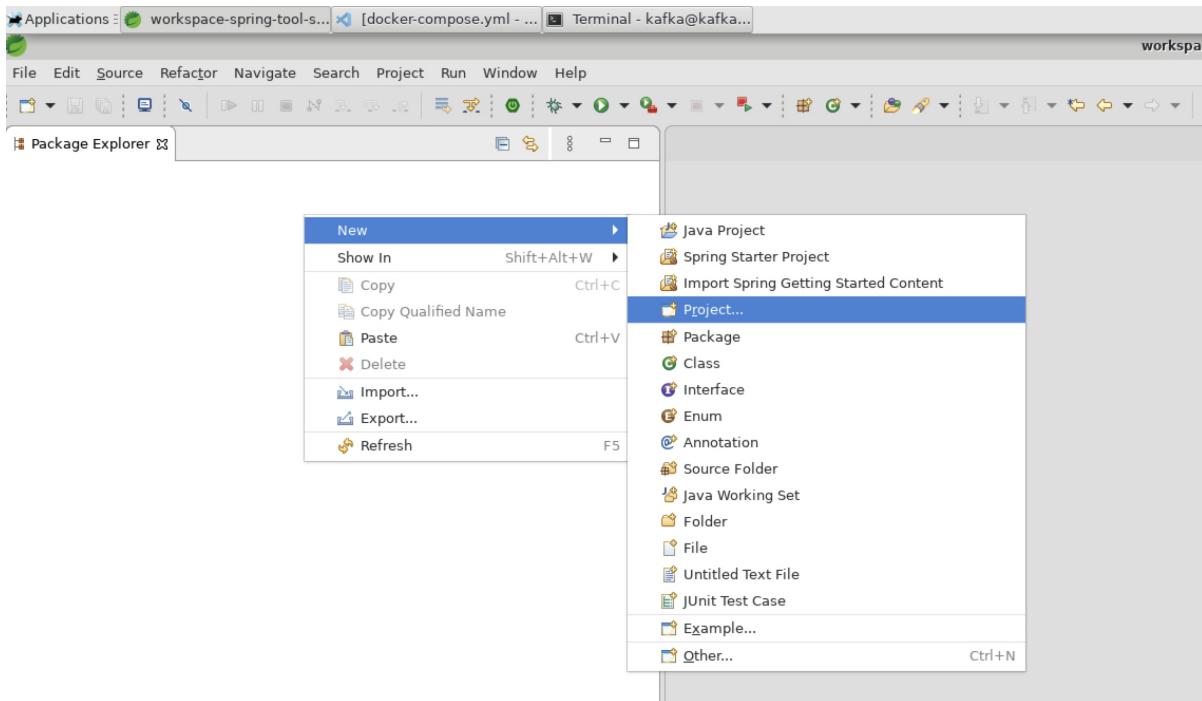
WatchedEvent state:Closed type:None path:null
2020-10-18 18:01:35,377 [myid:] - INFO  [main:ZooKeeper@1619] - Session: 0x10000fa6e4c0009 closed
2020-10-18 18:01:35,378 [myid:] - INFO  [main-EventThread:ClientCnxn$EventThread@577] - EventThread shut down for session: 0x10000fa6e4c0009
2020-10-18 18:01:35,379 [myid:] - ERROR [main:ServiceUtils@42] - Exiting JVM with code 0
```

- También comprobamos que tenemos conexión con kafka, y de paso creamos un nuevo topic.

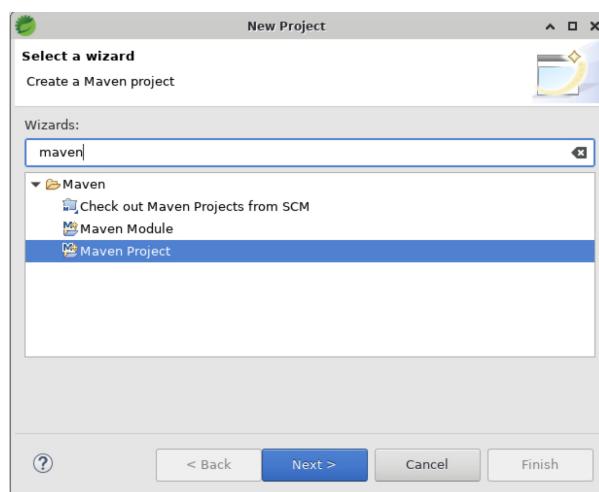
```
[kafka@kafka-server ~]$ kafka-topics.sh --bootstrap-server localhost:9092 --create --topic base-topic
Created topic base-topic.
```

7.4.5. Preparación de proyecto plantilla

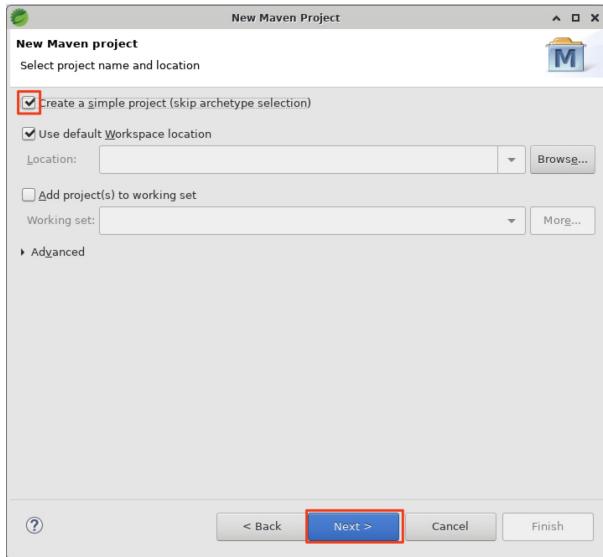
- Vamos a crear un proyecto plantilla el cual utilizaremos como base para los demás proyectos.
- Para ello abrimos nuestro IDE favorito y creamos un nuevo proyecto:



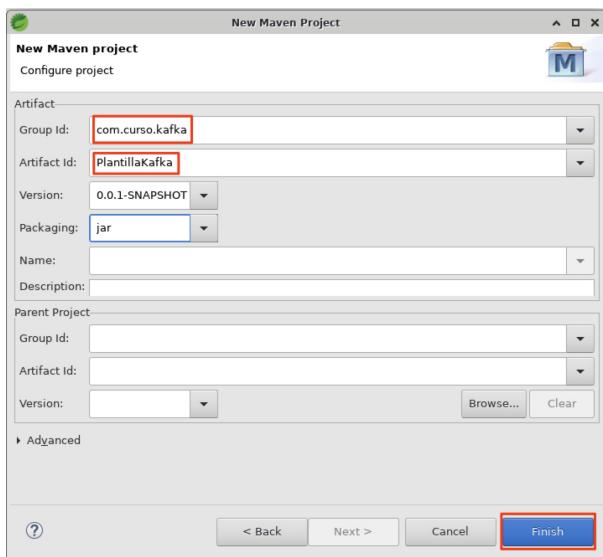
- Elegimos el **Maven Project**



- Seleccionamos la opción de crear un proyecto simple



- Rellenamos los siguientes campos
 - **Group id:** com.curso.kafka
 - **Artifact id:** PlantillaKafka
 - **Version:** 0.0.1-SNAPSHOT
 - **Packaging:** jar



- Pulsamos **finish**
- Ahora editamos el fichero pom.xml agregando las siguientes propiedades que permiten que se compile en Java 11 de forma automática:

```
<properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
</properties>
```

- Agregamos las dependencias mínimas para trabajar con kafka. Las Kafka Clients

```
<!-- Librerias Kafka -->
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.6.0</version>
</dependency>
```

- Debajo agregamos la librería helper de slf4j, ya que no viene ninguna implementación en las librerías de kafka, y sino no podría generar el sistema de log, y fallaría el arranque
- Y también jackson core para tratamiento de xml



- Cuando agreguemos kafka-streams, ya no será necesario, ya que viene incluida como dependencia.

```
<!-- Helpers -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.5</version>
</dependency>
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.11.3</version>
</dependency>
```

- El resultado final del fichero pom.xml será el siguiente:

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.curso.kafka</groupId>
  <artifactId>PlantillaKafka</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <!-- Java 11 OpenJDK como compilacion de proyecto -->
  <properties>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
  </properties>
  <dependencies>
    <!-- Librerias Kafka -->
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>2.6.0</version>
    </dependency>
    <!-- Helpers -->
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.5</version>
    </dependency>
  </dependencies>
</project>

```

- Para completar el sistema de log, vamos a agregar también el fichero log4j.properties en el directorio src/main/resources

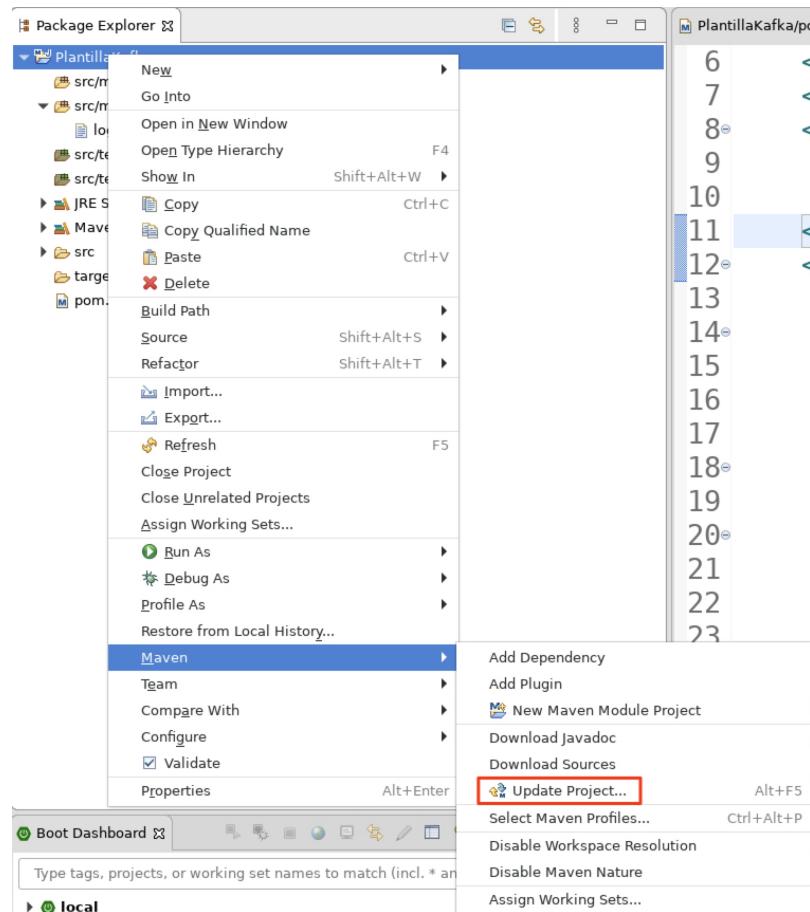
Contenido de src/main/resources/log4j.properties

```

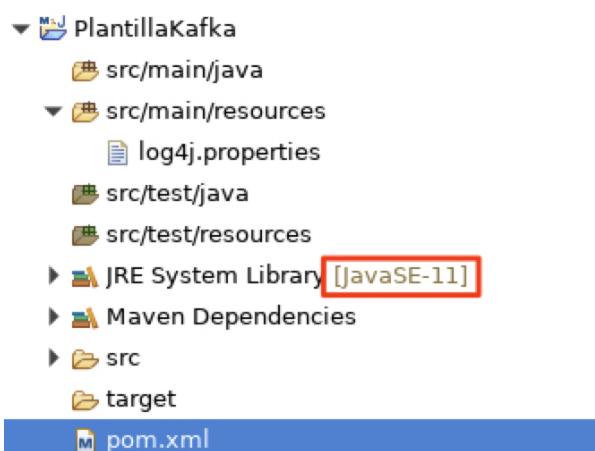
log4j.rootLogger=INFO, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d [%t] %-5p %c - %m%n

```

- Una vez terminado, debemos de sincronizar el proyecto maven con el proyecto del IDE.
- Para ello pulsamos botón derecho en la raiz del proyecto y seleccionamos **Maven → Update Project**



- Una vez que se sincronice y que guarde todas las dependencias, debemos cerciorarnos de que el IDE detecta el proyecto como Java 11



- Fin del laboratorio.

Capítulo 8. Configuración de Producers

- Las configuraciones que vamos a ver para los **Productores** son:
 - **bootstrap.servers** → Lista de Brokers
 - **key.serializer** → Clase para serializar la clave
 - **value.serializer** → Clase para serializar el valor
 - **retries** → Reintentos de envío
 - **ack** → Número de respuestas ack que esperamos
 - **compression.type** → Tipo de compresión
 - **batch.size** → Mensajes que se acumulan antes del envío
 - **linger.ms** → Tiempo máximo de espera para envío
 - **client.id** → Identificador del cliente
 - **partitioner.class** → Clase para el particionado
- **bootstrap.servers**: La lista de **Brokers**, separados por coma, indicando los puertos en los que están escuchando. Nuestro **productor** se pondrá en contacto con un **broker** de esta lista para crear el mapa de particionados.
- **key.serializer**: La clase que se debe utilizar por este **productor** para serializar la clave del mensaje. En nuestros ejemplos serializábamos los mensajes directamente como **Strings**. Kafka posee varios serializadores ya incluídos:
 - String → org.apache.kafka.common.serialization.StringSerializer
 - Long → org.apache.kafka.common.serialization.LongSerializer
 - Integer → org.apache.kafka.common.serialization.IntegerSerializer
 - Double → org.apache.kafka.common.serialization.DoubleSerializer
 - Bytes → org.apache.kafka.common.serialization.BytesSerializer
 - ByteArray → org.apache.kafka.common.serialization.ByteArraySerializer
 - ByteBuffer → org.apache.kafka.common.serialization.ByteBufferSerializer
- Podemos crear nuestro propio serializador, implementando el interfaz **Serializer<T>** (org.apache.kafka.common.serialization.Serializer), si queremos usar nuestros propios tipos en los mensajes.
- **value.serializer**: La clase que se usa para serializar el valor de los mensajes. Al igual que en el caso anterior, podemos hacer uso de una de las clases predefinidas o implementar nuestra propia serialización
- **retries**: Número de intentos que se van a realizar de enviar un mensaje. Cuidado, podemos obtener un fallo y eso no quiere decir que nuestro mensaje no se haya enviado (sencillamente que no se nos ha confirmado), con lo que se podría dar la situación de que enviáramos el mismo mensaje (podríamos duplicar mensajes)
- **ack**: Podemos configurar nuestros **productores** para esperar un número concreto de respuestas antes de dar por buena la escritura. Algunos valores son:

- 0 → "fire and forget", no esperamos confirmación. Es el que mayor rendimiento ofrece, pero el más susceptible a errores.
 - 1 → Sólo esperamos la respuesta de la partición líder
 - all → Esperamos la respuesta de todas las réplicas
- **compression.type** → Podemos realizar compresión a la hora de enviar nuestros mensajes (aunque por defecto no se realiza). Por cierto, los topics también se pueden configurar para que guarden comprimidos los datos
 - none → No se realiza compresión
 - gzip → Compresión gzip
 - snappy → Compresión snappy
 - lz4 → Compresión lz4
 - **batch.size**: Número de mensajes que van a agruparse antes de realizar el envío. Un valor de 1 envía todos los mensajes automáticamente según se generen.
 - **linger.ms**: Tiempo máximo de espera antes del envío. Si este tiempo se cumple, los mensajes se envían aunque no hayan alcanzado el número definido por **batch.size**
 - **client.id**: Es el identificador de cliente. Aunque existan varios **productores**, si todos comparten el mismo **client.id**, **Kafka** los considera el mismo cliente. Es de especial utilidad para definir cuotas.
 - En el siguiente ejemplo vemos cómo se define una cuota de tasa de producción para el cliente "clienteAlgo" de 1024 bytes por segundo (si supera esta tasa, **kafka** relentizará el cliente para no superarla):

```
[kafka@kafka-server ~]$ kafka-configs.sh --zookeeper localhost:2181 --alter --add -config 'producer_byte_rate=1024' --entity-name clienteAlgo --entity-type clients
```

- **partitioner.class**: Por último, indicar que a la hora de enviar mensajes, estos van a acabar en una partición. Tenemos dos opciones:
 - Indicar explícitamente para cada mensaje en qué partición queremos que acabe.
 - Usar un sistema de particionado.
- **Kafka** posee un particionador por defecto (basado en la clave hash obtenida mediante murmur3), pero podemos implementar nuestro propio particionador implementando el interfaz **Partitioner** (org.apache.kafka.clients.producer.Partitioner), y por supuesto especificando su uso en **partitioner.class**.

Capítulo 9. Configuración de Consumers

- Las configuraciones que vamos a ver para los **Consumidores** son:
 - **bootstrap.servers** → Lista de Brokers (análogo a la del productor)
 - **key.serializer** → Clase para serializar la clave (análogo a la del productor)
 - **value.serializer** → Clase para serializar el valor (análogo a la del productor)
 - **group.id** → Identificador del grupo de consumidores
 - **enable.auto.commit** → Control automático del offset
 - **auto.commit.interval.ms** → Cada cuantos milisegundos se actualiza el control del offset
 - **auto.offset.reset** → Para indicar dónde empieza un cliente nuevo
 - **client.id** → Identificador del cliente
- **group.id**: Servía para identificar un grupo de consumidores. Las instancias del consumidor que posean un mismo **group.id**, se reparten las particiones para procesar los mensajes de forma escalable. Los consumidores con distintos **group.id**, leerán los mismos mensajes.
- **enable.auto.commit**: Los clientes llevan un control de los mensajes leídos, para ello sencillamente tienen que recordar el último **offset** procesado para continuar (en caso de caída o parada, cuando vuelven a ejecutarse) donde lo habían dejado. Este control puede ser manual (hacemos desde la aplicación el control y guardamos el **offset** en momentos concretos), pero normalmente se hace de forma automática. Activando esta opción, hacemos que cada cierto tiempo, se guarde información del último **offset** leído.
- **auto.commit.interval.ms**: Cada cuánto tiempo se actualiza la información de los offsets
- **auto.offset.reset**: Cuando un cliente nuevo empieza a leer mensajes por primera vez, no hay información del último **offset** procesado. Tenemos dos opciones, leer desde el principio de los tiempos, o leer sólo los nuevos mensajes
 - **smallest** → Leemos desde el primer mensaje (como cuando usábamos la opción "--from-beginning" en "kafka-console-consumer.sh")
 - **largest** → Sólo leemos los mensajes nuevos (desde que mi cliente se levantó).
- **client.id**: Es el identificador de cliente. Aunque existan varios **consumidores**, si todos comparten el mismo **client.id**, **Kafka** los considera el mismo cliente. Es de especial utilidad para definir cuotas.
- En el siguiente ejemplo vemos cómo se define una cuota de tasa de consumo para el cliente "clienteAlgo" de 1024 bytes por segundo (si supera esta tasa, **kafka** relentizará el cliente para no superarla):

```
[kafka@kafka-server ~]$ kafka-configs.sh --zookeeper localhost:2181 --alter --add -config 'consumer_byte_rate=1024' --entity-name clienteAlgo --entity-type clients
```

- Como se puede ver, es análogo al ejemplo en el que poníamos una cuota al productor, pero en este caso al cliente le añadimos la opción "consumer_byte_rate"

Capítulo 10. Operaciones con Kafka

10.1. Utilidades y herramientas

- En este apartado vamos a ver 4 herramientas que vienen proporcionadas por **Kafka**, que son bastante útiles en distintos escenarios:
 - **kafka-preferred-replica-election.sh** → Para balancear las particiones líderes
 - **kafka-mirror-maker.sh** → Para copiar datos de un clúster a otro
 - **kafka-replay-log-producer.sh** → Para reproducir los mensajes de un Topic en otro
 - **kafka-replica-validation.sh** → Para verificar la validez de las réplicas

10.1.1. kafka-preferred-replica-election.sh (deprecada)

- Como ya vimos anteriormente, **Kafka** sólo balancea las particiones líderes en el momento de la creacion.
- Este balanceo, puede dejar de ser el adecuado al añadir nuevos brokers o al retirar o apagarlos.
- Para solucionar esto, tenemos la utilidad **kafka-preferred-replica-election.sh**, que balanceará las particiones líderes entre los distintos **brokers**.

```
[kafka@kafka-server ~]$ kafka-preferred-replica-election.sh --zookeeper localhost:2181
```

- Hay que tener una cosa en cuenta, balancea las particiones líderes entre los distintos **brokers** siguiendo una definición de "réplica favorita" propia de **kafka**.
- También tenemos que considerar que, si no hay réplicas en otros **brokers**, NO se va a usar otro **broker** como partición **Líder**, ya que no tiene réplica de dicha partición.
- Si queremos que este procedimiento se haga automáticamente (no bajo demanda), debemos a en el fichero de configuración
 - **server.properties** la opción **auto.leader.rebalance.enable**:

```
auto.leader.rebalance.enable=true
```

10.1.2. kafka-leader-election.sh

- Es la herramienta que sustituye a **kafka-preferred-replica-election**.
- Posee distintas opciones en las que destacan:
 - **--topic**: Permite definir que topic va a gestionar para la elección de réplicas líderes.
 - **--partition**: Indica que partición va a modificar
 - **--all-topic-partitions**: Para elegir todas las particiones basado en el tipo de elección.
 - **--election-type**: Indica el tipo de elección que se va a realizar:

- preferred: La elección solo se realiza si el preferido no es el líder
- unclean: solo si no hay líder en la partición del topic
- --path-to-json-file: Indica un fichero json con la lista de particiones y topics a gestionar

Ejemplo de fichero JSON para modificar del topic 1 la particion 1 y del topic2 la particion 3

```
{"partitions": [
  [
    {"topic": "topic1", "partition": 1},
    {"topic": "topic2", "partition": 3}
  ]
}]
```

10.1.3. kafka-mirror-maker.sh

- Utilidad que permite copiar datos desde otro clúster **Kafka**.
- Puede ser de interés si necesitáis realizar un backup, o tener una copia de producción para hacer pruebas.
- Se invoca con los parámetros:
 - **consumer.config** → Se le pasa un fichero con la configuración del clúster destino
 - **producer.config** → Se le pasa un fichero con la configuración del clúster origen
 - **whitelist** → los topics que queremos coger (admite expresiones regulares)
 - **num.streams** → Número de hilos para los consumidores

```
[kafka@kafka-server ~]$ kafka-mirror-maker.sh --consumer.config consumer.properties
--producer.config producer.properties --whitelist testTopic
```

10.1.4. kafka-replay-log-producer.sh

- Otra herramienta útil para realizar pruebas.
- Permite copiar los mensajes de un **topic** a otro, permitiéndonos por ejemplo realizar pruebas de nuestras aplicaciones contra datos reales extraídos de otro **topic**, y reproducir el comportamiento que hubo con los datos de ayer, por ejemplo.

```
[kafka@kafka-server ~]$ kafka-replay-log-producer.sh --broker-list localhost:9092
--inputtopic input --outputtopic output --zookeeper localhost:2181 --threads 1
```

- Como se puede ver, en **inputtopic** se especifica el origen, en **outputtopic** el destino, y podemos configurar el número de hilos que se van a dedicar en dicha tarea con **threads**

10.1.5. kafka-replica-verification.sh

- Nos va a permitir verificar las réplicas existentes de un conjunto de topics, para confirmar que

todas tienen los mismos datos.

- En su versión más simple, lo hace para todos los **topics**, pero el parámetro **--topic-white-list** permite especificar un conjunto de **topics** mediante una expresión regular.
- Otra opción interesante es **--time**, que permite especificar a partir de qué timestamp realizar la verificación.

```
[kafka@kafka-server ~]$ kafka-replica-validation.sh --broker-list localhost:9092  
--topic-white-list "^to.*a$"
```

- En el ejemplo vamos a verificar todos los topic cuyo nombre empiece por 'to', y acabe por la letra 'a'.

Capítulo 11. Kafka Java API

11.1. Dependencias

- Necesitamos añadir a un nuevo proyecto las dependencias a las librerías **Kafka**.
- Si usamos **Maven**, con añadir dependencias a **kafka-clients** debería ser suficiente.
 1. Dependencia mínima a utilizar

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.6.0</version>
</dependency>
```

En nuestro caso, vamos a crear una librería de usuario con las librerías que hay en **\$KAFKA_HOME/lib**

NOTA: Al crear la librería incluimos jackson-databind, esto tendría que ser añadido también a las dependencias de **maven**

11.2. API para Producer

- Para poder hacer nuestro propio **productor**, necesitaremos conocer las siguientes clases:
 - **KafkaProducer** → Las instancias de esta clase poseen un método **send()** que será usado para realizar los envíos al **Topic**
 - **ProducerRecord** → Las instancias de esta clase son los mensajes que queremos enviar a nuestros **Topics**. Poseen **Clave** y **Valor**, pero también debemos especificar el **Topic** destino, y podemos especificar la **partición** destino (o dejarlo a cargo de un particionador)
 - **ProducerConfig** → Clase que contiene las constantes con las distintas configuraciones aplicables. Es recomendable hacer uso de ellas al crear el fichero **properties**, en lugar de escribirlas manualmente.
- Vamos a empezar con la clase **KafkaProducer**, que está en el paquete **org.apache.kafka.clients.producer**.
- Esta clase posee un método **send()**, que nos permite enviar los mensajes a los **Topics** destino de manera asíncrona.
 - **send(ProducerRecord<K,V> record, Callback callback)** → Ejecutamos el envío del mensaje. Al ser procesado, se ejecutará el método proporcionado en **callback**. Podemos usar **send(record)** directamente, con lo que no invocaríamos un método al recibir el registro (o tener una excepción). Devuelve un objeto **Future<RecordMetadata>**.
 - **partitionsFor(String topic)** → Devuelve un **List<PartitionInfo>**, para poder consultar la información de las particiones de un **topic**.
 - **close()** → Cierra el productor.

- **flush()** → Fuerza el envío de los mensajes pendientes (aunque no lleguemos al linger.ms o batch.size) y bloquea hasta recibir la respuesta de estos.
- **metrics()** → Devuelve un **Map<MetricName,? extends Metric>**, con las métricas internas mantenidas por el **productor**.
- **initTransactions()** → Si vamos a usar transacciones debemos invocar este método antes de cualquier uso de las mismas (y **transactional.id** está puesto en la configuración)
- **beginTransaction()** → Para empezar una nueva transacción
- **commitTransaction()** → Valida la transacción actual
- **abortTransaction()** → Invalida la transacción actual



- Más información en: <https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/producer/KafkaProducer.html>

- Ya hemos dicho antes, que para enviar mensajes, necesitamos envolverlos en una instancia de **ProducerRecord**, que está en **org.apache.kafka.clients.producer**.
- Las instancias de **ProducerRecord** contienen el mensaje que queremos enviar a nuestros **topics**. La parte más importante de esta clase es su constructor, ya que es donde especificamos tanto el contenido, como el destino.
 - **ProducerRecord(String topic, V value)** → Enviamos a un topic un valor sin clave
 - **ProducerRecord(String topic, K key, V value)** → Enviamos a un topic un mensaje con clave y valor
 - **ProducerRecord(String topic, Integer partition, K key, V value)** → Enviamos a una partición concreta de un topic un mensaje con clave y valor
- También posee algunos métodos útiles para consultar el contenido:
 - **key()** → Devuelve la clave del mensaje
 - **value()** → Devuelve el valor del mensaje
 - **partition()** → Devuelve la partición a la que queremos enviar el mensaje (si lo hemos especificado)
 - **timestamp()** → Devuelve el timestamp del momento de creación del mensaje
 - **topic()** → Devuelve el topic que hemos especificado como destino
- Por último, vamos a ver las opciones de configuración.
 - Para poder instanciar un **KafkaProducer**, necesitamos facilitarle un objeto **Properties**.
 - Sobre él, tenemos que definir una serie de configuraciones (vamos a ver ahora las más importantes).
 - Para facilitarnos esto, **Kafka** nos ofrece la clase **ProducerConfig**, del paquete **org.apache.kafka.clients.producer**, que contiene las constantes con los nombres de las propiedades que podemos usar.
 - **ProducerConfig.CLIENT_ID_CONFIG** → Para especificar el **client.id** del productor.
 - **ProducerConfig.COMPRESSION_TYPE_CONFIG** → Para especificar el **compression.type** (tipo de compresión).

- **ProducerConfig.ACKS_CONFIG** → Especifica cuántas respuestas espera, propiedad **acks**.
- **ProducerConfig.RETRIES_CONFIG** → Especifica los reintentos, propiedad **retries**.
- **ProducerConfig.BOOTSTRAP_SERVERS_CONFIG** → Lista de los nodos a los que conectarnos para obtener el esquema, propiedad **bootstrap.servers**.
- **ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG** → Clase para serializar la clave, propiedad **key.serializer**.
- **ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG** → Clase para serializar el valor, propiedad **value.serializer**.
- **ProducerConfig.BATCH_SIZE_CONFIG** → Tamaño de mensajes antes del envío, propiedad **batch.size**.
- **ProducerConfig.LINGER_MS_CONFIG** → Tiempo máximo de almacenamiento de mensajes antes de envío, propiedad **linger.ms**.
- **ProducerConfig.PARTITIONER_CLASS_CONFIG** → Clase usada para decidir a qué partición enviar los mensajes, propiedad **partitioner.class**.
- Con todo esto en juego, vamos a crear nuestro primer ejemplo, que enviará a **topicEjemplo01**

```
package com.kafka.api.evolucion;

import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;

//kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic topicEjemplo01 --property print.key=true --fromBeginning
public class ProducerEjemplos01 {
    public static void main(String[] args) throws InterruptedException {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "host.broker1:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringSerializer");
    }

    Producer<String, String> producer = new KafkaProducer<>(props);
    for (int id = 0; id < 5000; id++) {
        producer.send(new ProducerRecord<>("topicEjemplo01", "Mensaje "+id));
        Thread.sleep(1000);
    }
    producer.flush();
    producer.close();
}
}
```

- Podemos añadir un **Callback** a nuestro **send()**, creamos la clase **CallbackSimple** implementando la interfaz **org.apache.kafka.clients.producer.Callback**

```

package com.kafka.api.evolucion;

import java.util.Properties;
import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

//kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic topicEjemplo01 --property print.key=true --from
-bEGINNING
public class ProducerEjemplos02 {
    public static void main(String[] args) throws InterruptedException {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "host.broker1:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringSerializer");
    }

    Producer<String, String> producer = new KafkaProducer<>(props);
    for (int id = 0; id < 5000; id++) {
        producer.send(new ProducerRecord<>("topicEjemplo01", "Mensaje "+id),new Callback(){
            @Override
            public void onCompletion(RecordMetadata meta, Exception arg1) {
                System.out.println("Mensaje escrito en "+meta.topic()+" particion "+meta.partition()+" con offset "
+meta.offset());
            }
        });
        Thread.sleep(1000);
    }
    producer.flush();
    producer.close();
}
}

```

- Lo habitual es que mandemos los mensajes con una clave, así que vamos a modificar un poco nuestro ejemplo:

```

package com.kafka.api.evolucion;

import java.util.Properties;
import org.apache.kafka.clients.producer.Callback;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
public class ProducerEjemplos03 {
    public static void main(String[] args) throws InterruptedException {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "host.broker1:9092");
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringSerializer");
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringSerializer"
);
        Producer<String, String> producer = new KafkaProducer<>(props);
        for (int id = 0; id < 5000; id++) {
            producer.send(new ProducerRecord<>("topicEjemplo01", "ID_" + id, "Mensaje " + id), new Callback(){
                @Override
                public void onCompletion(RecordMetadata meta, Exception arg1) {
                    System.out.println("Mensaje escrito en " + meta.topic() + " particion " + meta.partition() + " con offset "
+meta.offset());
                }
            });
            Thread.sleep(1000);
        }
        producer.flush();
        producer.close();
    }
}

```

- Con respecto al **particionador**, si queremos especificar el nuestro propio, debemos implementar la clase **Partitioner** del paquete **org.apache.kafka.clients.producer**.
- Quizás lo más interesante sea obtener del objeto **Cluster** información, concretamente **partitionCountForTopic()** que nos dice cuántas particiones existen para un **Topic** en concreto.

```

package com.kafka.api.producer;

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import java.util.Map;

public class SimplePartitioner implements Partitioner {

    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object value,
    byte[] valueBytes, Cluster cluster) {
        return Math.abs(key.hashCode() % cluster.partitionCountForTopic(topic));
    }

    @Override
    public void close() {}

    @Override
    public void configure(Map<String, ?> conf) {}
}

```

- No nos vamos a meter tampoco a estudiar los serializadores y de-serializadores, pero son interfaces que hemos de implementar para serializar o de-serializar nuestras clases en caso de que no nos valga con los que **Kafka** proporciona.
- La interfaz `org.apache.kafka.common.serialization.Serializer` obliga a implementar la función `serialize(topic, map)` debe convertir un `Map<String, Object>` en un array de bytes `byte[]`
- La interfaz `org.apache.kafka.common.serialization.Deserializer` obliga a implementar la función `deserialize(topic, data)` debe convertir un array de bytes `byte[]` en un `Map<String, Object>`

11.3. API para Consumer

- Para poder hacer nuestro propio **Consumer**, necesitaremos conocer las siguientes clases:
 - **KafkaConsumer** → Las instancias de esta clase poseen un método `subscribe()`, que permite suscribirse a unos **Topics**, y hacer `poll()` de sus mensajes.
 - **ConsumerRecord** → Las instancias de esta clase son los mensajes que consumimos de nuestros **Topics**. Poseen **Clave** y **Valor**, pero también tienen metainformación como **offset**, **partición**, **topic** o **timestamp**
 - **ConsumerConfig** → Clase que contiene las constantes con las distintas configuraciones aplicables. Es recomendable hacer uso de ellas al crear el fichero **properties**, en lugar de escribirlas manualmente.
- La clase **KafkaConsumer**, que está en el paquete `org.apache.kafka.clients.consumer`, tiene varios métodos que debemos conocer:
 - `assign(Collection<TopicPartition> c)` → Permite asignar una lista de particiones al

consumidor

- **commitAsync() / commitSync()** → Guardado manual del offset leído
- **metrics()** → Métricas internas del consumidor
- **poll(t)** → Trae los registros de los **topics** suscritos (si no hay, espera t milisegundos)
- **seek(TopicPartition t, long offset)** → Para especificar a partir de qué offset quiero traer datos de un topic y una partición concretas
- **subscribe(Collection<String> topics)** → Se suscribe a los **topics** especificados (Admite Regex en lugar de colección de String)
- **subscription()** → Nos da la lista de topics a los que está suscrito
- **unsubscribe()** → se retira de los topics (todos) a los que está suscrito



- Puedes obtener más información en: <https://kafka.apache.org/10/javadoc/org/apache/kafka/clients/consumer/KafkaConsumer.html>

- El **KafkaConsumer** se suscribe a una lista de **Topics**, y hace **poll()** sobre ella. Cuando realiza esta acción, nos devuelva una "colección" de mensajes leídos. Esta colección es un **ConsumerRecords<K,V>**, objeto sobre el que podemos iterar para tratar cada **ConsumerRecord<K,V>** individualmente.
- Este objeto proporciona unos métodos para poder extraer la clave y el valor, y cualquier otro dato que pueda ser de utilidad:
 - **key()** → Devuelve la clave del mensaje
 - **value()** → Devuelve el valor del mensaje
 - **partition()** → Devuelve la partición de la que hemos leído
 - **timestamp()** → Devuelve el timestamp del momento de creación del mensaje
 - **topic()** → Devuelve el topic del que hemos leído
- Por último, vamos a ver las opciones de configuración. Para poder instanciar un **KafkaConsumer**, necesitamos facilitarle un objeto **Properties**. Sobre él, tenemos que definir una serie de configuraciones (vamos a ver ahora las más importantes).
- Para facilitarnos esto, **Kafka** nos ofrece la clase **ConsumerConfig**, del paquete **org.apache.kafka.clients.consumer**, que contiene las constantes con los nombres de las propiedades que podemos usar.
 - **ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG** → Lista de los nodos a los que conectarnos para obtener el esquema, propiedad **bootstrap.servers**.
 - **ConsumerConfig.GROUP_ID_CONFIG** → Para especificar el **group.id** del consumidor. Todos los consumidores con este grupo actúan como un único consumidor.
 - **ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG** → Clase para serializar la clave, propiedad **key.deserializer**.
 - **ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG** → Clase para serializar el valor, propiedad **value.deserializer**.
 - **ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG** → Para que haga un commit manual

del último offset leído, propiedad "enable.auto.commit".

- **ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG** → Especifica el tiempo entre commits, propiedad **auto.commit.interval.ms**.
- **ConsumerConfig.SESSION_TIMEOUT_MS_CONFIG** → Tiempo máximo de la sesión, propiedad **session.timeout.ms**.
- **ConsumerConfig.AUTO_OFFSET_RESET_CONFIG** → Qué hacer si este grupo de clientes no tiene offset inicial, propiedad **auto.offset.reset**. Puede tomar:
 - **latest**: El offset se fija en el último registro existente, es decir, sólo mensajes nuevos (por defecto)
 - **earliest**: Si no hay offset, se empieza desde el primer registro
 - **none**: Si no hay offset, salta una excepción
- Vamos a hacer nuestro primer consumidor (Fijaros, que sólo procesa a partir de los mensajes nuevos, si queremos que esto cambi hay que usar **ConsumerConfig.AUTO_OFFSET_RESET_CONFIG**):

```
package com.kafka.api.evolucion;

import java.util.Collections;
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

public class ConsumerEjemplo01 {
    public static void main(String[] args) {
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "host.broker1:9092");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "consumer_base");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringDeserializer");
        String s=ConsumerConfig.AUTO_OFFSET_RESET_CONFIG;
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Collections.singletonList("topicEjemplo01"));
        int leidos=0;
        while (true) {
            ConsumerRecords<String, String> records = consumer.poll(1000);
            for (ConsumerRecord<String, String> record : records)
                System.out.printf("partition = %2d offset = %5d key = %7s timestamp = %8s value = %12s\n",
                record.partition(), record.offset(), record.key(), String.valueOf(record.timestamp()), record.value());
            if (++leidos>10) { break;}
        }
        consumer.close();
    }
}
```

- Esta configuración no es la que buscamos.
- Normalmente no vamos a parar al leer una serie de mensajes, estaremos hasta que alguien pare

la JVM.

- Para evitar esto, vamos a usar una variable **AtomicBoolean**, que nos permita parar el proceso cuando se reciba una señal de parar en la JVM

```
package com.kafka.api.evolucion;

import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.atomic.AtomicBoolean;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;

public class ConsumerEjemplo02 {
    private static final AtomicBoolean closed = new AtomicBoolean(false); //para cerrar al matar el proceso

    public static void main(String[] args) {
        Runtime.getRuntime().addShutdownHook(new Thread(){
            @Override
            public void run() {
                System.out.println("Shutting down");
                closed.set(true);
            }
        });

        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "host.broker1:9092");
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "consumer_base");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringDeserializer");
        String s=ConsumerConfig.AUTO_OFFSET_RESET_CONFIG;
        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Collections.singletonList("topicEjemplo01"));
        int leidos=0;
        while (!closed.get()) {
            ConsumerRecords<String, String> records = consumer.poll(1000);
            for (ConsumerRecord<String, String> record : records)
                System.out.printf("partition = %2d offset = %5d key = %7s timestamp = %8s value = %12s\n",
                    record.partition(), record.offset(), record.key(), String.valueOf(record.timestamp()), record.value());
            if (++leidos>10) { break;}
        }
        consumer.close();
    }
}
```

11.4. Lab: Invocando productores y consumidores

- Para realizar este laboratorio necesitamos tener iniciado tanto zookeeper como los brokers.
- Si se ha usado Docker, debemos comprobar si están o no levantados.
- Si aparece la lista vacía los iniciamos en orden

```
[kafka@kafka-server ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAME
```

- Nos colocamos en la carpeta del dockerfile e iniciamos zookeeper y los brokers

```
[kafka@kafka-server ~]$ cd Documents/docker-kafka
[kafka@kafka-server docker-kafka]$ docker-compose up -d zookeeper
[kafka@kafka-server docker-kafka]$ docker-compose up -d kafka1
[kafka@kafka-server docker-kafka]$ docker-compose up -d kafka2
[kafka@kafka-server docker-kafka]$ docker-compose up -d kafka3
```

- Comprobamos que están iniciados correctamente

```
[kafka@kafka-server docker-kafka]$ docker ps
CONTAINER ID        IMAGE               COMMAND
STATUS             PORTS              NAMES
093e9d1c649e      bitnami/kafka:latest   "/opt/bitnami/script...
Up 3 seconds          "                  docker-kafka_kafka1_1
551540ef4ce4      bitnami/kafka:latest   "/opt/bitnami/script...
Up 2 seconds          "                  docker-kafka_kafka2_1
fbe3387824b3      bitnami/kafka:latest   "/opt/bitnami/script...
Up 2 seconds          "                  docker-kafka_kafka3_1
40c2020d1486      bitnami/zookeeper:latest "/opt/bitnami/script...
Up 2 mintues         "                  docker-kafka_zookeeper_1
```

- Comprobamos que nuestros brokers están correctamente registrados:

```
[kafka@kafka-server ~]$ zkCli.sh
Connecting to localhost:2181
2019-01-23 19:34:34,042 [myid:] - INFO  [main:Environment@100] - Client
environment:zookeeper.version=3.4.14-e5259e437540f349646870ea94dc2658c4e44b3b, built on 03/27/2018
03:55 GMT
...
[zk: localhost:2181(CONNECTED) 0] ls /brokers/ids
[0, 1, 2]
```

- Vamos a crear un **Topic** para poder probar nuestro programa:

```
[kafka@kafka-server ~]$ kafka-topics.sh --bootstrap-server localhost:9092 --create  
--topic topicSimple --partitions 3
```

- Comprobamos que el topicSimple está creado correctamente

```
[kafka@kafka-server docker-kafka]$ kafka-topics.sh --bootstrap-server localhost:9092  
--describe --topic topicSimple  
Topic: topicSimple PartitionCount: 3 ReplicationFactor: 1 Configs:  
segment.bytes=1073741824  
Topic: topicSimple Partition: 0 Leader: 2 Replicas: 2 Isr: 2  
Topic: topicSimple Partition: 1 Leader: 3 Replicas: 3 Isr: 3  
Topic: topicSimple Partition: 2 Leader: 1 Replicas: 1 Isr: 1
```

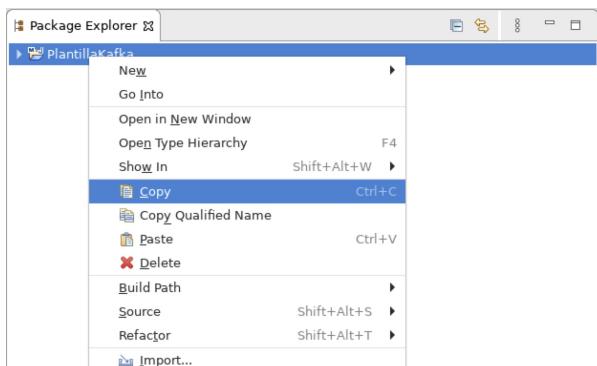
11.4.1. Creación de productor

- Para ello vamos a copiar nuestra plantilla y llamamos al proyecto kafka-java-api

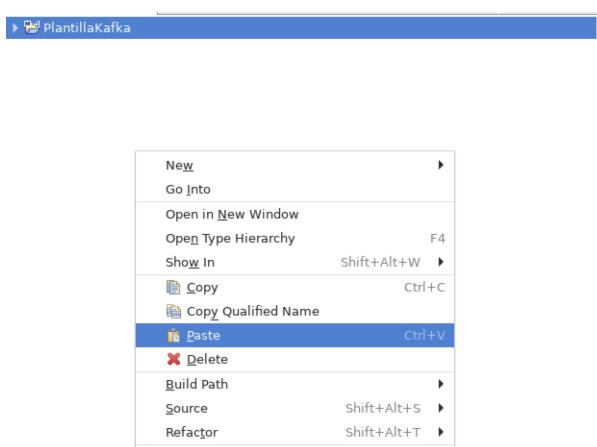


- Sería mucho más interesante generar un arquetipo en Maven para tener directamente el esquema del proyecto, pero por agilidad, copiaremos el proyecto

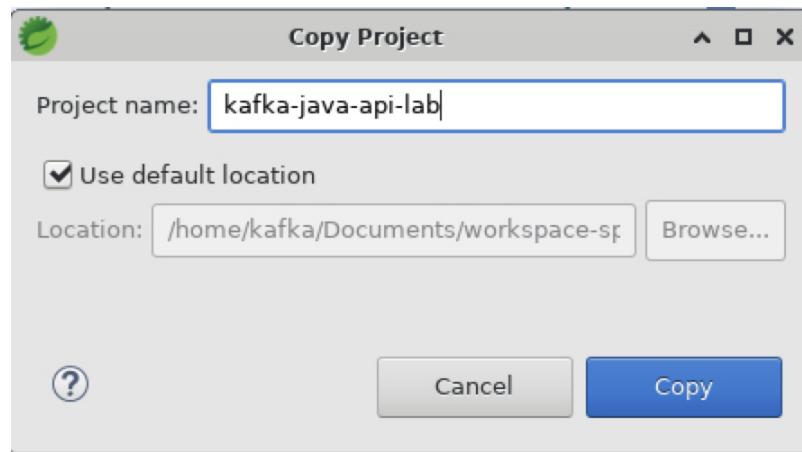
- Primero seleccionamos el proyecto y lo copiamos



- Luego pegamos en el **Package Explorer** el proyecto



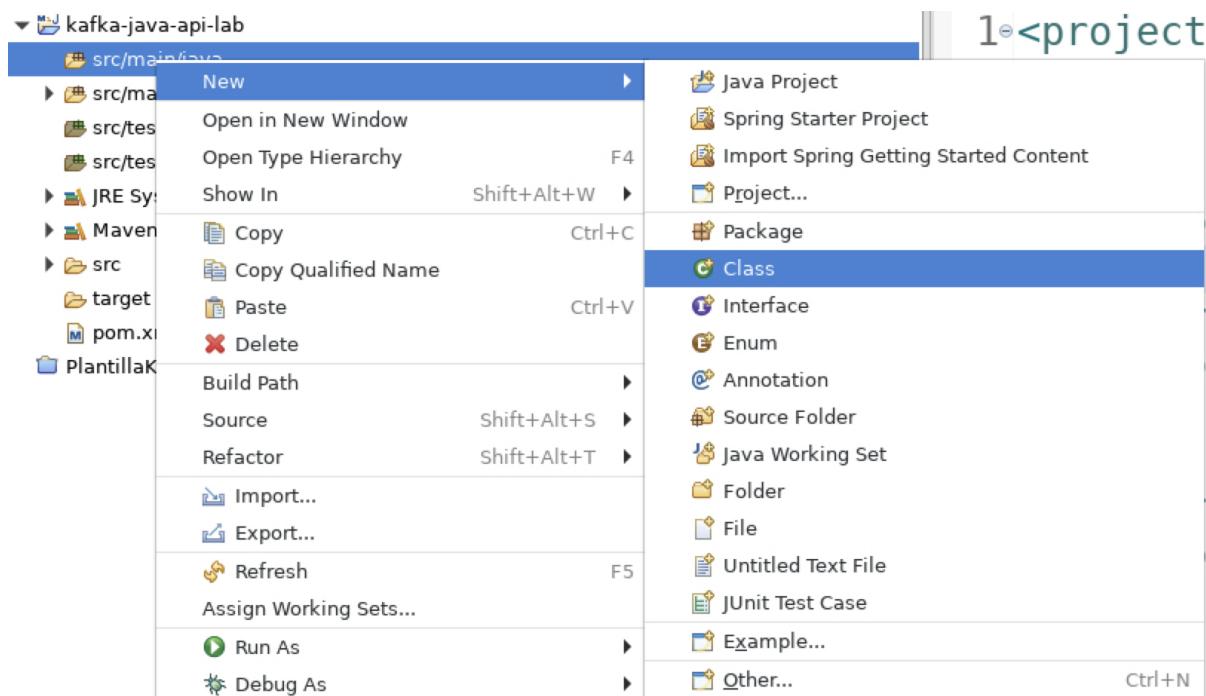
- Llamamos al proyecto **kafka-java-api-lab**



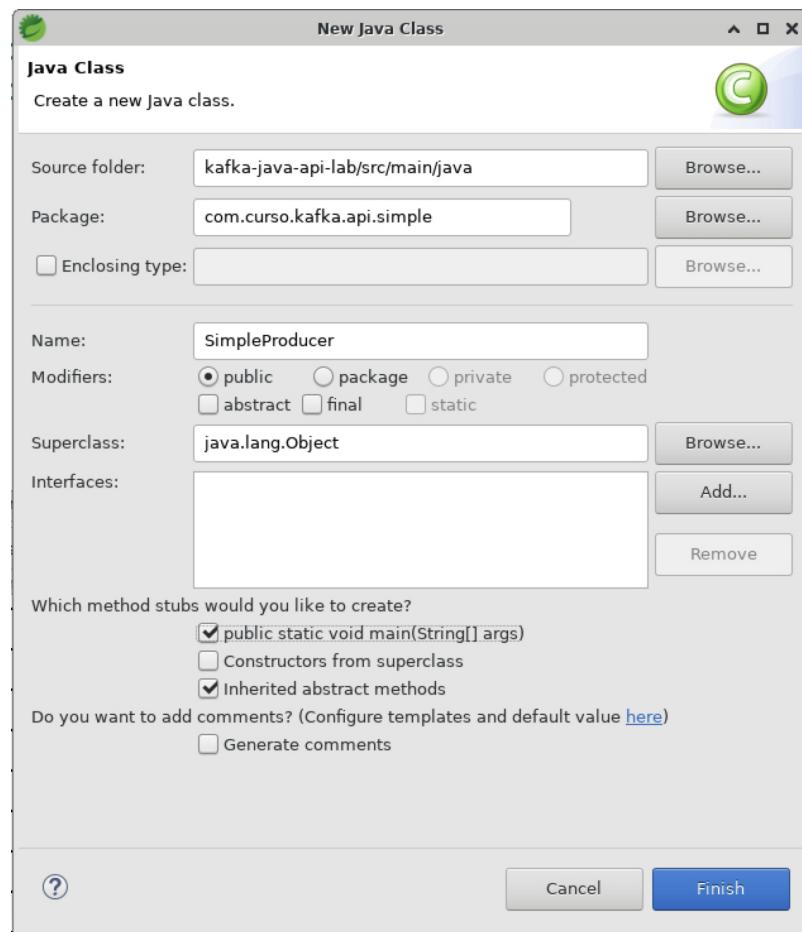
- Modificamos el pom del proyecto para que se llame como el proyecto que hemos creado.
- Solo modificamos el <artifactId>

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.curso.kafka</groupId>
  <artifactId>kafka-java-api-lab</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

- Vamos a producir mensajes, para ello creamos un productor:
- Creamos una nueva clase:



- La nueva clase se llamará:
 - Package: com.curso.java.api.simple
 - Name: SimpleProducer
 - Public static void main: true



- El contenido será el siguiente:

```

package com.curso.kafka.api.simple;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

public class SimpleProducer {

    public static final String BROKER_LIST = "localhost:9091,localhost:9092,localhost:9093";
    public static final String TOPIC = "topicSimple";

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BROKER_LIST);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        Producer<String, String> producer = new KafkaProducer<>(props);

        for (int id = 0; id < 5000; id++) {
            String key = String.format("key[%d]", id);
            String message = String.format("message[%d]", id);
            System.out.println("Sending message with: " + key);
            producer.send(new ProducerRecord<>(TOPIC, key, message));
            Thread.sleep(1000);
        }

        producer.flush();
        producer.close();
    }
}

```

- Si analizamos el contenido, al principio indicamos la configuración del productor

```

props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, brokerList);
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

```

- Estamos indicando la lista inicial de los brokers de kafka
- Indicamos los serializadores por defecto para almacenar en el topic

```
Producer<String, String> producer = new KafkaProducer<>(props);
```

- Aquí vemos como se crea el productor indicando la clave y el valor, y asignando las propiedades por defecto.

```

for (int id = 0; id < 5000; id++) {
    String key = String.format("key[%d]", id);
    String message = String.format("message[%d]", id);
    System.out.println("Sending message with: " + key);
    producer.send(new ProducerRecord<>(topic, key, message));
    Thread.sleep(1000);
}

```

- En el bucle, mandamos mensajes de tipo clave/valor al topic elegido
- Esperamos un segundo por petición



- El hecho de enviar no significa explicitamente que envie.
- Debe cumplir una condición de número de mensajes encolados o un tiempo de terminado, y manda toda la información en paquetes.

```

producer.flush();
producer.close();

```

- Forzamos a enviar los documentos que estén encolados con flush y cerramos el canal con close.
- Ejecutamos la clase para conectarse a nuestro topic y enviar los datos.

```

Sending message with: key[0]
Sending message with: key[1]
Sending message with: key[2]
...

```

- Nuestro **Productor** está funcionando!

11.4.2. Simple Consumer

- Veamos ahora con nuestro consumidor. Creamos la clase **com.curso.kafka.api.consumer.simple.SimpleConsumer**

```

package com.curso.kafka.api.simple;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.atomic.AtomicBoolean;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

public class SimpleConsumer {

    public static String KAFKA_HOST = "localhost:9092";
    private static final AtomicBoolean closed = new AtomicBoolean(false); //para cerrar al matar el proceso

    public static void main(String[] args) {
        Runtime.getRuntime().addShutdownHook(new Thread(){
            @Override
            public void run() {
                System.out.println("Shutting down");
                closed.set(true);
            }
        });

        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, KAFKA_HOST);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "simple-consumer");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Collections.singletonList(SimpleProducer.TOPIC));

        while (!closed.get()) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
            for (ConsumerRecord<String, String> record : records)
                System.out.printf("partition = %2d offset = %5d key = %7s timestamp = %8s value = %12s\n",
                    record.partition(), record.offset(), record.key(), String.valueOf(record.timestamp()), record.value());
        }

        consumer.close();
    }
}

```

- Primero definimos el shutdown hook para cerrar el consumidor correctamente.

```

Runtime.getRuntime().addShutdownHook(new Thread(){
    @Override
    public void run() {
        System.out.println("Shutting down");
        closed.set(true);
    }
});

```

- Configuramos el consumidor:

```
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, KAFKA_HOST);
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
props.put(ConsumerConfig.GROUP_ID_CONFIG, "simple-consumer");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
```

- Indicamos al menos un servidor para que sea alcanzable y pueda preguntar por el mapa de brokers que necesite para el topic.
- Activamos el autocommit para que se acuerde cada 100 ms de donde estaba el consumidor
- Indicamos cuales son los deserializadores para clave y valor.
- Ahora creamos el consumidor y nos suscribimos al topic. Podríamos suscribirnos a una lista de topics.

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(Collections.singletonList(SimpleProducer.TOPIC));
```

- Por último, indicamos que mientras no esté cerrada la vm, vamos a consumir registros:

```
while (!closed.get()) {
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(1000));
    for (ConsumerRecord<String, String> record : records)
        System.out.printf("partition = %2d offset = %5d key = %7s timestamp = %8s value = %12s\n",
                          record.partition(), record.offset(), record.key(), String.valueOf(record.timestamp()), record.value());
}
```

- Obtenemos el record y vemos que información relevante posee publicandola directamente en el log

11.4.3. Prueba de ejecución

- Ejecutamos el consumidor y confirmamos que consume los mensajes nuevos.

```
...
partition = 1 offset =      3 key = key[13] timestamp = 1603106530616 value = message[13]
partition = 1 offset =      4 key = key[14] timestamp = 1603106531617 value = message[14]
partition = 0 offset =      6 key = key[15] timestamp = 1603106532619 value = message[15]
partition = 2 offset =      4 key = key[16] timestamp = 1603106533621 value = message[16]
partition = 1 offset =      5 key = key[17] timestamp = 1603106534622 value = message[17]
```

- Paramos el consumidor y nos fijamos en que offset estaba trabajando:

```

partition = 1 offset = 15 key = key[37] timestamp = 1603106554646 value = message[37]
partition = 2 offset = 10 key = key[38] timestamp = 1603106555647 value = message[38]
partition = 0 offset = 12 key = key[39] timestamp = 1603106556648 value = message[39]

```

- Volvemos a arrancarlo al cabo de unos segundos.

```

partition = 0 offset = 13 key = key[41] timestamp = 1603106558650 value = message[41]
partition = 0 offset = 14 key = key[42] timestamp = 1603106559651 value = message[42]
partition = 0 offset = 15 key = key[48] timestamp = 1603106565658 value = message[48]
partition = 0 offset = 16 key = key[52] timestamp = 1603106569665 value = message[52]
partition = 0 offset = 17 key = key[53] timestamp = 1603106570667 value = message[53]
partition = 0 offset = 18 key = key[54] timestamp = 1603106571668 value = message[54]
partition = 0 offset = 19 key = key[61] timestamp = 1603106578678 value = message[61]
partition = 0 offset = 20 key = key[62] timestamp = 1603106579679 value = message[62]
partition = 0 offset = 21 key = key[63] timestamp = 1603106580681 value = message[63]
partition = 0 offset = 22 key = key[65] timestamp = 1603106582688 value = message[65]
partition = 0 offset = 23 key = key[67] timestamp = 1603106584696 value = message[67]
partition = 0 offset = 24 key = key[68] timestamp = 1603106585698 value = message[68]
partition = 2 offset = 11 key = key[40] timestamp = 1603106557649 value = message[40]
partition = 2 offset = 12 key = key[43] timestamp = 1603106560653 value = message[43]

```

- Comprobamos como sigue trabajando directamente desde donde lo dejó.
- Hay que darse cuenta que el orden no está garantizado entre particiones.

11.4.4. PartitionProducer

- Si nos damos cuenta, a pesar de tener un topic con tres particiones, toda la información va a la primera partición.
- Vamos a crear un nuevo particionador y un productor basado en el anterior para comprobar que podemos decidir a donde van los documentos por medio del mismo.
- Primero creamos el particionador en com.curso.kafka.api.simple.partitioner.SimplePartitioner

```

package com.curso.kafka.api.simple.partitioner;

import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import java.util.Map;

public class SimplePartitioner implements Partitioner {

    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {
        return Math.abs(key.hashCode() % cluster.partitionCountForTopic(topic));
    }

    @Override
    public void close() {}

    @Override
    public void configure(Map<String, ?> conf) {}
}

```

- En este caso, usamos el código hash y calculamos el módulo con la información de particiones que nos ofrece el cluster.
- Ahora vamos a crear un nuevo productor que use el nuevo particionador.
- Copiamos el productor y lo renombramos a SimpleProducerWithPartitioner.
- Agregamos la información del partitioner en la configuración:

```
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, SimplePartitioner.class.getName());
```

- La clase resultante es la siguiente.

```

package com.curso.kafka.api.simple;

import java.util.Properties;
import java.util.concurrent.ExecutionException;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import com.curso.kafka.api.simple.partitioner.SimplePartitioner;

public class SimpleProducerWithPartitioner {

    public static final String BROKER_LIST =
"localhost:9091,localhost:9092,localhost:9093";
    public static final String TOPIC = "topicSimple";

    public static void main(String[] args) throws InterruptedException,
ExecutionException {
        Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BROKER_LIST);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class
.getName());
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.
class.getName());
        props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, SimplePartitioner.class
.getName());

        Producer<String, String> producer = new KafkaProducer<>(props);

        for (int id = 0; id < 5000; id++) {
            String key = String.format("key[%d]", id);
            String message = String.format("message[%d]", id);
            System.out.println("Sending message with: " + key);
            producer.send(new ProducerRecord<>(TOPIC, key, message));
            Thread.sleep(1000);
        }

        producer.flush();
        producer.close();
    }
}

```

- Iniciamos el productor para que mande los mensajes y luego el consumidor si no lo hemos dejado levantado con anterioridad.
- Comprobaremos como estamos usando nuestro propio particionador.

```
partition = 1 offset = 53 key = key[0] timestamp = 1603106685658 value = message[0]
partition = 0 offset = 57 key = key[1] timestamp = 1603106686672 value = message[1]
partition = 2 offset = 45 key = key[2] timestamp = 1603106687674 value = message[2]
partition = 1 offset = 54 key = key[3] timestamp = 1603106688677 value = message[3]
partition = 0 offset = 58 key = key[4] timestamp = 1603106689677 value = message[4]
partition = 2 offset = 46 key = key[5] timestamp = 1603106690681 value = message[5]
partition = 1 offset = 55 key = key[6] timestamp = 1603106691683 value = message[6]
partition = 0 offset = 59 key = key[7] timestamp = 1603106692686 value = message[7]
```

11.4.5. Groups

- Con el productor levantado, vamos a crear un nuevo consumidor
- Para ello creamos la siguiente clase en com.curso.kafka.api.groups

```

package com.curso.kafka.api.groups;

import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

public class GroupConsumer {

    public static String KAFKA_HOST = "localhost:9092";
    public static String TOPIC = "topicSimple";
    public static Integer THREADS = 2;
    public static List<KafkaConsumerRunner> consumers = new ArrayList<>();

    public static void main(String[] args) {
        Runtime.getRuntime().addShutdownHook(new Thread(){
            @Override
            public void run() {
                System.out.println("Shutting down");
                for(KafkaConsumerRunner consumerRunner : consumers) consumerRunner.shutdown();
            }
        });

        Properties props = new Properties();
        props.put("bootstrap.servers", KAFKA_HOST);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "consumer-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        ExecutorService executor = Executors.newFixedThreadPool(THREADS);

        for (Integer threads = 0; threads < THREADS; threads++) {
            KafkaConsumerRunner consumerRunner = new KafkaConsumerRunner(threads, consumer, TOPIC);
            consumers.add(consumerRunner);
            executor.submit(consumerRunner);
        }
    }
}

```

- Básicamente, estamos creando un consumidor con un grupo.
- En cuanto lo ejecutemos, veremos como se apodera de todas las particiones y consume todos los registros.

```

Adding : [topicSimple-0, topicSimple-1, topicSimple-2]
....
THREAD[0]: partition = 2 offset = 176 key = key[394] value = message[394]
THREAD[0]: partition = 0 offset = 188 key = key[395] value = message[395]
THREAD[0]: partition = 1 offset = 185 key = key[396] value = message[396]
THREAD[0]: partition = 2 offset = 177 key = key[397] value = message[397]
THREAD[0]: partition = 0 offset = 189 key = key[398] value = message[398]
THREAD[0]: partition = 1 offset = 186 key = key[399] value = message[399]

```

- Ahora ejecutamos otra instancia de la misma clase con run as → Java Application.
- Comprobaremos como en la instancia inicial aparece el siguiente mensaje

```

Removing : [topicSimple-0, topicSimple-1, topicSimple-2]
Adding : [topicSimple-0, topicSimple-1]
THREAD[0]: partition = 0 offset = 190 key = key[401] value = message[401]
THREAD[0]: partition = 1 offset = 187 key = key[402] value = message[402]
THREAD[0]: partition = 0 offset = 191 key = key[404] value = message[404]
THREAD[0]: partition = 1 offset = 188 key = key[405] value = message[405]
THREAD[0]: partition = 0 offset = 192 key = key[407] value = message[407]

```

- Y en la nueva instancia aparece la partición que no está asignada, y a partir de ahí, los mensajes solo se leen desde las particiones que posee
- En cuanto a la segunda instancia:

```

Adding : [topicSimple-2]
THREAD[1]: partition = 2 offset = 179 key = key[403] value = message[403]
THREAD[1]: partition = 2 offset = 180 key = key[406] value = message[406]
THREAD[1]: partition = 2 offset = 181 key = key[409] value = message[409]
THREAD[1]: partition = 2 offset = 182 key = key[412] value = message[412]
THREAD[1]: partition = 2 offset = 183 key = key[415] value = message[415]
THREAD[1]: partition = 2 offset = 184 key = key[418] value = message[418]

```

- Comprobamos como se agrupan y obtienen los mensajes de la partición adicional.
- Arrancamos una tercera instancia y veremos como cada uno posee el acceso a una sola partición.
- Así garantizamos que las instancias lean mensajes distintos.
- Pero, y si agregamos un cuarto,
- Veremos como una de las instancias se queda sin recibir mensajes.

```

THREAD[0]: partition = 2 offset = 322 key = key[832] value = message[832]
Removing : [topicSimple-2]
Adding : []

```

- Si paramos cualquier otra instancia, el sistema lo entiende y reasigna las particiones

```
Removing : []
Adding   : [topicSimple-2]
THREAD[0]: partition = 2    offset = 345    key = key[901]    value = message[901]
```

- Paramos todos los consumidores

11.4.6. Autocommit

- En este caso vamos a gestionar el commit de forma manual.
- Para ello creamos una clase llamada ManualConsumer.

```

package com.curso.kafka.api.autocommit;

import org.apache.kafka.clients.consumer.*;

import com.curso.kafka.api.simple.SimpleProducer;

import java.time.Duration;
import java.util.Collections;
import java.util.Properties;
import java.util.concurrent.atomic.AtomicBoolean;

public class ManualConsumer {
    public static String KAFKA_HOST = "localhost:9092";
    private static final AtomicBoolean closed = new AtomicBoolean(false);

    public static void main(String[] args) throws InterruptedException {
        Runtime.getRuntime().addShutdownHook(new Thread(){
            @Override
            public void run() {
                System.out.println("Shutting down");
                closed.set(true);
            }
        });

        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, KAFKA_HOST);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "manual-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, "org.apache.kafka.common.serialization.StringDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        "org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Collections.singletonList(SimpleProducer.TOPIC));

        while (!closed.get()) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("partition = %2d offset = %5d key = %7s value = %12s\n",
                    record.partition(), record.offset(), record.key(), record.value());
                Thread.sleep(5000);
            }

            consumer.commitSync();
        }

        consumer.close();
    }
}

```

- Ejecutamos el nuevo productor

```

partition = 0 offset = 470 key = key[1242] value = message[1242]
partition = 0 offset = 471 key = key[1245] value = message[1245]
partition = 0 offset = 472 key = key[1248] value = message[1248]
partition = 0 offset = 473 key = key[1251] value = message[1251]

```

- Ejecutamos este consumidor, posteriormente el productor. Vemos como va consumiendo. Hacemos una parada después de unos 10 segundos y volvemos a levantarla.

- Paramos entonces el consumidor y lo dejamos parado 10 segundos.

```
partition = 0    offset = 470    key = key[1242]    value = message[1242]
partition = 0    offset = 471    key = key[1245]    value = message[1245]
```

- Hay registros que ha vuelto a procesar. Como no llegó a guardar el último offset, no registró los últimos mensajes procesados.

11.4.7. PartitionConsumer

- Vamos a crear un consumidor que se conecte a una sola partición.
- Para ello creamos la siguiente clase:

```

package com.curso.kafka.api.partitions;

import java.util.Collections;
import java.util.HashSet;
import java.util.Properties;
import java.util.Set;
import java.util.concurrent.atomic.AtomicBoolean;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.serialization.StringDeserializer;

import com.curso.kafka.api.simple.SimpleProducer;

public class PartitionsConsumer {
    public static String KAFKA_HOST = "localhost:9092";
    private static final AtomicBoolean closed = new AtomicBoolean(false);

    public static void main(String[] args) {
        Runtime.getRuntime().addShutdownHook(new Thread(){
            @Override
            public void run() {
                System.out.println("Shutting down");
                closed.set(true);
            }
        });
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, KAFKA_HOST);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "500");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "partition-group");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        Set<TopicPartition> partitions = new HashSet<>();

        partitions.add(new TopicPartition(SimpleProducer.TOPIC, 0));
        partitions.add(new TopicPartition(SimpleProducer.TOPIC, 1));
        consumer.assign(partitions);

        while (!closed.get()) {
            ConsumerRecords<String, String> records = consumer.poll(100);
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("topic = %2s partition = %2d offset = %5d key = %7s value = %12s\n",
                    record.topic(), record.partition(), record.offset(), record.key(), record.value());
            }
        }

        consumer.close();
    }
}

```

- Al ejecutarlo, vemos como solo usa las particiones que hemos decidido.

```

topic = topicSimple partition = 1 offset = 764 key = key[2132] value = message[2132]
topic = topicSimple partition = 0 offset = 767 key = key[2133] value = message[2133]
topic = topicSimple partition = 1 offset = 765 key = key[2135] value = message[2135]
topic = topicSimple partition = 0 offset = 768 key = key[2136] value = message[2136]

```

11.4.8. SeekConsumer

- Usando el truco de las particiones, vamos a buscar en este caso un offset concreto.
- Para ello vamos a crear la siguiente clase

```

package com.curso.kafka.api.seek;

import java.time.Duration;
import java.util.Collections;
import java.util.HashSet;
import java.util.Properties;
import java.util.Set;
import java.util.concurrent.atomic.AtomicBoolean;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.TopicPartition;
import org.apache.kafka.common.serialization.StringDeserializer;

import com.curso.kafka.api.simple.SimpleProducer;

public class SeekConsumer {
    public static String KAFKA_HOST = "localhost:9092";
    private static final AtomicBoolean closed = new AtomicBoolean(false);

    public static void main(String[] args) {
        Runtime.getRuntime().addShutdownHook(new Thread(){
            @Override
            public void run() {
                System.out.println("Shutting down");
                closed.set(true);
            }
        });
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, KAFKA_HOST);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "500");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "seek-group3");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        Set<TopicPartition> partitions = new HashSet<>();
        partitions.add(new TopicPartition(SimpleProducer.TOPIC, 0));
        partitions.add(new TopicPartition(SimpleProducer.TOPIC, 1));
        partitions.add(new TopicPartition(SimpleProducer.TOPIC, 2));
        consumer.assign(partitions);
    }
}

```

```

        consumer.seekToBeginning(Collections.singleton(new TopicPartition(SimpleProducer.TOPIC, 0)));
        consumer.seekToEnd(Collections.singleton(new TopicPartition(SimpleProducer.TOPIC, 1)));

        consumer.seek(new TopicPartition(SimpleProducer.TOPIC, 2), 600);

        while (!closed.get()) {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
            for (ConsumerRecord<String, String> record : records) {
                System.out.printf("topic = %s partition = %d offset = %d key = %s value = %s\n",
                    record.topic(), record.partition(), record.offset(), record.key(), record.value());
            }
        }

        consumer.close();
    }
}

```

- En este caso, vamos a ejecutarlo con las siguientes condiciones:
 - Leemos de la partición 0 desde el principio.
 - Leemos de la partición 1 a partir del último registro
 - Leemos de la partición 2 a partir del offset 600 (Si no tenemos offset 600 no dará error y leerá desde el final).

```

topic = topicSimple partition = 2 offset = 600 key = key[1666] value = message[1666]
topic = topicSimple partition = 2 offset = 601 key = key[1669] value = message[1669]
topic = topicSimple partition = 2 offset = 602 key = key[1672] value = message[1672]
topic = topicSimple partition = 2 offset = 603 key = key[1675] value = message[1675]
topic = topicSimple partition = 2 offset = 604 key = key[1678] value = message[1678]
...
topic = topicSimple partition = 0 offset = 0 key = key[0] value = message[0]
topic = topicSimple partition = 0 offset = 1 key = key[5] value = message[5]
topic = topicSimple partition = 0 offset = 2 key = key[7] value = message[7]
topic = topicSimple partition = 0 offset = 3 key = key[8] value = message[8]
...
topic = topicSimple partition = 0 offset = 709 key = key[1959] value = message[1959]
topic = topicSimple partition = 2 offset = 698 key = key[1960] value = message[1960]
topic = topicSimple partition = 1 offset = 707 key = key[1961] value = message[1961]

```

11.4.9. Consumer Info

- Los consumidores poseen información relevante de los topics donde están conectados.
- Para comprobarlo, creamos la siguiente clase:

```

package com.curso.kafka.api.info;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.Node;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.TopicPartition;

```

```

import org.apache.kafka.common.serialization.StringDeserializer;

import com.curso.kafka.api.simple.SimpleProducer;

import java.util.*;
import java.util.concurrent.atomic.AtomicBoolean;

public class InfoConsumer {
    public static String KAFKA_HOST = "localhost:9092";
    private static final AtomicBoolean closed = new AtomicBoolean(false);

    public static void main(String[] args) {
        Runtime.getRuntime().addShutdownHook(new Thread(){
            @Override
            public void run() {
                System.out.println("Shutting down");
                closed.set(true);
            }
        });

        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, KAFKA_HOST);
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
        props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "100");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "info-simple");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

        for(Map.Entry<String, List<PartitionInfo>> entry : consumer.listTopics().entrySet()){
            System.out.println("Topic: " + entry.getKey());
            for(PartitionInfo partition : entry.getValue()) {
                Set<Integer> replicas = new HashSet<>();
                Set<Integer> inSync = new HashSet<>();

                for(Node node : partition.replicas()) replicas.add(node.id());
                for(Node node : partition.inSyncReplicas()) inSync.add(node.id());

                System.out.println(String.format(" P: %2s Leader: %2s Replicas: %4s InSync: %4s",
                    partition.partition(), partition.leader().id(), replicas, inSync));
            }
        }

        System.out.println("-----");
        System.out.println("-----");
        TopicPartition topic = new TopicPartition(SimpleProducer.TOPIC, 0);
        Set<TopicPartition> topics = new HashSet<TopicPartition>();
        topics.add(topic);

        System.out.println(String.format("Last offsets for %s : %s", topic, consumer.committed(topics)));

        consumer.close();
    }
}

```

- Si ejecutamos, el resultado será el siguiente

```
Topic: topicSimple
P: 0 Leader: 2 Replicas: [2] InSync: [2]
P: 2 Leader: 1 Replicas: [1] InSync: [1]
P: 1 Leader: 3 Replicas: [3] InSync: [3]
```

```
Topic: __consumer_offsets
P: 0 Leader: 1 Replicas: [1] InSync: [1]
P: 10 Leader: 2 Replicas: [2] InSync: [2]
P: 20 Leader: 3 Replicas: [3] InSync: [3]
```

....

```
Last offsets for topicSimple-0 : {topicSimple-0=null}
```

```
Shutting down
```



- Fin del laboratorio

Capítulo 12. Esquemas en Kafka

- La serialización es una forma de representar los datos en memoria como conjuntos de bytes para transferencia o almacenamiento en disco
- La deserialización permite realizar la operación inversa, convirtiendo los bytes en objetos
- Kafka posee su propia clase de serialización, como hemos visto anteriormente
 - org.apache.kafka.common.serialization
- Sin embargo, el tratamiento de datos de forma simple no es suficiente. La serialización en formatos de tipo texto es poco eficiente:
 - Almacenamiento no eficiente
 - Los datos de tipo no-texto deben pasarse a cadenas
 - Es ineficiente transformar datos de texto a binarios y viceversa

12.1. Tipos de serialización en kafka

- Existen cuatro tipos de formatos de serialización comunes en Apache kafka:
 - Avro
 - Es un formato binario, lo que beneficia en tamaño almacenado la mayoría de las veces
 - No es fácilmente entendible, ya que se almacena en binario
 - Posee un esquema conocido.
 - Soporta schema registry y está muy integrado con kafka
 - JSON
 - No almacena en formato binario
 - Es fácil de interpretar
 - No posee un esquema, aunque en los nuevos estándares esto se solucionará
 - Sencillo, sin curva de aprendizaje y muy común
 - No está indicado para alto rendimiento.
 - Protobuf
 - Almacena en formato binario
 - No es fácil de interpretar al ser binario
 - Posee esquema
 - Google Protobuf es altamente eficiente.
 - Uso de gRPC
 - Thrift
 - Almacena en formato binario
 - No es fácil de interpretar al ser binario

- Posee esquema
- Usado por twitter y en las primeras versiones de Apache Cassandra
- Todos ellos son agnósticos a la plataforma y el lenguaje.
- Sin embargo, aquellos que usan datos binarios permiten compactar los datos, usan esquemas y permiten un alto rendimiento.

12.1.1. Ejemplos Esquemas-IDL

Ejemplo JSON - *schema.json*

```
{
  "type": "string"
}
```

Ejemplo Proto - *schema.proto*

```
syntax = "proto3"
message UserCommand {
    string command = 1
}
```

Ejemplo avro - *schema.avsc*

```
{
  "type": "string",
  "name": "command"
}
```

Ejemplo thrift schema.thrift

```
struct UserCommand {
    1: string command
}
```

- Estos esquemas permiten ceñirnos a una estructura concreta de datos.
- Definimos un grano fino de estructuras de datos.

12.2. Avro

- Se trata de un sistema que permite la serialización de datos, no es solo un serializador/deserializador
- Creado por *Doug Cutting*, el creador de Hadoop
- Permite:
 - Posee estructuras de datos complejas y ricas

- Uso de datos binarios compactados, ocupando menos espacio en disco y uso menor de red
- Permite el uso de contenedores, es decir, el fichero contenedor puede almacenar el esquema en la cabecera y el objeto en el resto del mensaje.
- Permite comunicación RPC, puede definir su propio protocolo.
- Serialización de datos
- Los datos se definen con un esquema
- Soportado por diversos lenguajes de programación
- Soporta generadores de código para los tipos de datos
- La comprobación de tipos se realiza en tiempo de escritura.

12.3. Esquemas de Avro

- Los esquemas definen la estructura de datos.
- Se representan en formato JSON
- Posee tres formas de creación de records
 - **Generic** : Mapeo de cada campo al campo del objeto
 - **Reflection** : Generación de esquema a partir de una clase java
 - **Específica** : Generación de una clase java para nuestro esquema

12.4. Tipos de datos

- Avro soporta dos tipos de datos, primitivos o complejos:

Name	Description	Java equivalent
boolean	True or false	boolean
int	32-bit signed integer	int
long	64-bit signed integer	long
float	Single-precision floating-point number	float
double	Double-precision floating-point number	double
string	Sequence of Unicode characters	java.lang.CharSequence
bytes	Sequence of bytes	java.nio.ByteBuffer
null	The absence of a value	null

Figure 2. Tipos de datos simples

Name	Description
record	A user-defined field comprising one or more simple or complex data types, including nested records
enum	A specified set of values
union	Exactly one value from a specified set of types
array	Zero or more values, each of the same type
map	Set of key/value pairs; key is always a string , value is the specified type
fixed	A fixed number of bytes

Figure 3. Tipos de datos complejos

- Podemos destacar el tipo de datos fixed que permite almacenar una cadena fija de caracteres, lo que viene muy bien para almacenar hashes en avro.
- Más información de los tipos de datos en:
 - <https://avro.apache.org/docs/1.10.0/spec.html>

12.4.1. Compactación

- Avro utiliza un sistema de compactación para cada tipo de datos primitivo.
- Para ello usa los siguientes criterios:
 - null: No almacena datos
 - boolean: bit 0 o 1
 - int/long: Uso de variable-length zig-zag encoding. Lo que le permite no distinguir entre int y long y no reservar ese espacio concreto para cada uno.
 - float/double: Almacenan 32bit IEEE 754 para float y 64bit IEEE 754 para double
 - Binary: En la cabecera almacena el tamaño como long con zig-zag encoding, y luego los bytes.
 - String: En la cabecera almacena el tamaño como long con zig-zag encoding, y luego almacnea el texto en una cadena codificada en UTF-8

12.4.2. Integración

- Este es un ejemplo de un esquema en Avro.

Ejemplo de esquema en Avro:

```
{  
  "namespace": "model",  
  "type": "record",  
  "name": "SimpleCard",  
  "fields": [  
    {  
      "name": "suit",  
      "type": "string",  
      "doc": "The suit of the card"  
    }, {  
      "name": "card",  
      "type": "string",  
      "doc": "The card number"  
    }  
  ]  
}
```

- Por defecto, los esquemas se almacenan con extensión **.avsc** en el directorio `src/main/avro`
- El namespace es el paquete java
- Docs permite indicar comentarios del campo

Ejemplo con un array y un map

```
{  
  "name": "cards_list",  
  "type": {  
    "type": "array",  
    "items": "string"  
  },  
  "doc": "The cards played"  
}, {  
  "name": "cards_map",  
  "type": {  
    "type": "map",  
    "values": "string"  
  },  
  "doc": "The cards played"  
}, {  
  "name": "suit_type",  
  "type": {  
    "type": "enum",  
    "name": "Suit",  
    "symbols": ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]  
  },  
  "doc": "The suit of the card"  
}
```

- Los esquemas de Avro se pueden actualizar.
- Permite también la compatibilidad mediante esquemas
 - Retrocompatibilidad:
 - El código con una nueva versión de esquema puede leer la versión vieja
 - Los campos no existentes los deja con valores por defecto
 - Compatibilidad a futuro
 - Si el código recibe nuevos esquemas, los campos nuevos son ignorados

12.5. Avro y Java

- Uno de los lenguajes mas comunes es Java.
- Avro se comunica de forma sencilla con java y permite generar recursos de forma sencilla.
- Existen dos metodologías para generar las clases Avro para su serialización/deserialización.

12.5.1. Avro Tools

- La primera es por medio del jar avro-tools-1.10.0.jar
- Con las avro tools podemos generar los objetos de forma sencilla por medio del siguiente comando:

```
$ java -jar avro-tools-1.10.0.jar compile schema <fichero_esquema_avsc> <ruta_destino>
```

- La variable fichero_esquema_avsc es la ruta al esquema en formato avro que hemos generado anteriormente
- La variable ruta_destino usará el namespace del esquema para generar las clases que permitirán serializar/deserializar los contenidos

12.5.2. Maven

- Con maven podemos automatizar la generación de los contenidos.
- Para ello, necesitamos primero declarar en el pom.xml la dependencia de avro:

```
<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.10.0</version>
</dependency>
```

- Luego necesitamos indicar una fase de construcción que permita generar la clase java.
- Para ello generamos un nuevo build:

```

<plugin>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro-maven-plugin</artifactId>
    <version>1.10.0</version>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>schema</goal>
            </goals>
            <configuration>
                <sourceDirectory>${project.basedir}/src/main/avro</sourceDirectory>
                <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
            </configuration>
        </execution>
    </executions>
</plugin>

```

- Una vez generado, solo necesitamos guardar en el directorio src/main/avro cuya raiz es el directorio con el fichero pom.xml todos los esquemas.
- Este elegirá todos los ficheros contenidos en el directorio src/main/avro que posean la extensión .avsc
- Usarán el directorio src/main/java para mostrar el resultado, usarán el namespace del esquema para generar los paquetes donde se aloje la clase, y el name como nombre de la clase

12.5.3. Lab: Generando las clases avro

- Vamos a crear un esquema y lo vamos a generar de dos formas.
- La primera será por medio de las avro-tools, y la segunda será por medio de maven.

12.5.3.1. Avro-tools

- Vamos a generar con las avro-tools las clases necesarias para un tipo record que se almacenará en kafka.
- Para ello, creamos primero un directorio llamado avro-labs en el directorio Documents.

```
[kafka@kafka-server ~]$ cd Documents/  
[kafka@kafka-server Documents]$ mkdir avro-labs  
[kafka@kafka-server Documents]$ cd avro-labs/
```

- Dentro vamos a crear nuestro nuevo esquema sencillo.
- Para ello creamos un nuevo fichero llamado UserCommand.avsc con el siguiente contenido:

```
{  
  "name": "UserCommand",  
  "namespace": "com.curso.kafka.avro.model",  
  "type": "record",  
  "fields":  
  [  
    {  
      "name": "command",  
      "type": "string"  
    }  
  ]  
}
```

- Creamos una carpeta llamada avro-tools y descargamos la librería de avro tools Dentro

```
[kafka@kafka-server avro-labs]$ mkdir avro-tools  
[kafka@kafka-server avro-labs]$ cd avro-tools/  
[kafka@kafka-server avro-tools]$ wget https://repo1.maven.org/maven2/org/apache/avro/avro-tools/1.10.0/avro-tools-1.10.0.jar  
--2020-10-14 16:05:22-- https://repo1.maven.org/maven2/org/apache/avro/avro-tools/1.10.0/avro-tools-1.10.0.jar  
Resolving repo1.maven.org (repo1.maven.org)... 199.232.192.209, 199.232.196.209  
Connecting to repo1.maven.org (repo1.maven.org)|199.232.192.209|:443... connected.  
HTTP request sent, awaiting response... 200 OK  
Length: 47426559 (45M) [application/java-archive]  
Saving to: `avro-tools-1.10.0.jar`  
  
avro-tools-1.10.0.jar 100%[=====] 45.23M 51.6MB/s in 0.9s  
  
2020-10-14 16:05:23 (51.6 MB/s) - `avro-tools-1.10.0.jar` saved [47426559/47426559]
```

- Dentro copiamos el esquema que generamos anteriormente

```
[kafka@kafka-server avro-tools]$ cp ./UserCommand.avsc .
[kafka@kafka-server avro-tools]$ ls
avro-tools-1.10.0.jar  UserCommand.avsc
```

- Por último, construimos la clase por medio del siguiente comando:

```
[kafka@kafka-server avro-tools]$ java -jar avro-tools-1.10.0.jar compile schema UserCommand.avsc .
Input files to compile:
  UserCommand.avsc
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

- Comprobamos el resultado:

```
[kafka@kafka-server avro-tools]$ tree .
.
├── avro-tools-1.10.0.jar
└── com
    └── curso
        └── kafka
            └── avro
                └── model
                    └── UserCommand.java
└── UserCommand.avsc
```

- Podemos observar también la nueva clase UserCommand.java

Parte del contenido de la clase UserCommand.java

```
[kafka@kafka-server avro-tools]$ cat com/curso/kafka/avro/model/UserCommand.java
/**
 * Autogenerated by Avro
 *
 * DO NOT EDIT DIRECTLY
 */
package com.curso.kafka.avro.model;

import org.apache.avro.generic.GenericArray;
import org.apache.avro.specific.SpecificData;
import org.apache.avro.util.Utf8;
import org.apache.avro.message.BinaryMessageEncoder;
import org.apache.avro.message.BinaryMessageDecoder;
import org.apache.avro.message.SchemaStore;

@org.apache.avro.specific.AvroGenerated
public class UserCommand extends org.apache.avro.specific.SpecificRecordBase implements org.apache.avro.specific.SpecificRecord {
    private static final long serialVersionUID = -8944913839296861894L;
    public static final org.apache.avro.Schema SCHEMA$ = new org.apache.avro.Schema.Parser().parse("{\"type\":\"record\", \"name\":\"UserCommand\", \"namespace\":\"com.curso.kafka.avro.model\", \"fields\":[{\"name\":\"command\", \"type\":\"string\"}]}");
    public static org.apache.avro.Schema getClassSchema() { return SCHEMA$; }

    private static SpecificData MODEL$ = new SpecificData();

    private static final BinaryMessageEncoder<UserCommand> ENCODER =
        new BinaryMessageEncoder<UserCommand>(MODEL$, SCHEMA$);

    private static final BinaryMessageDecoder<UserCommand> DECODER =
        new BinaryMessageDecoder<UserCommand>(MODEL$, SCHEMA$);

    /**
     * Return the BinaryMessageEncoder instance used by this class.
     * @return the message encoder used by this class
     */
    public static BinaryMessageEncoder<UserCommand> getEncoder() {
        return ENCODER;
    }
    ...
}
```

12.5.3.2. Avro maven

- Necesitamos tener instalado maven.
- Para comprobarlo, ejecutamos el siguiente comando:

```
[kafka@kafka-server avro-labs]$ mvn -v
Apache Maven 3.6.3 (cecedd343002696d0abb50b32b541b8a6ba2883f)
Maven home: /usr/local/apache-maven-3.6.3
Java version: 11.0.8, vendor: N/A, runtime: /usr/lib/jvm/java-11-openjdk-11.0.8.10-0.el8_2.x86_64
Default locale: en_US, platform encoding: UTF-8
OS name: "linux", version: "4.18.0-193.6.3.el8_2.x86_64", arch: "amd64", family: "unix"
```

- Para este caso, vamos a crear una jerarquía de carpetas desde la carpeta avro-labs:

Parte del contenido de la clase UserCommand.java

```
[kafka@kafka-server avro-labs]$ mkdir -p avro-maven/src/main/avro
```

- Dentro vamos a meter el esquema original que tenemos en el directorio.

Parte del contenido de la clase UserCommand.java

```
[kafka@kafka-server avro-labs]$ mv UserCommand.avsc avro-maven/src/main/avro/
```

- Nos colocamos en el directorio avro-maven:

```
[kafka@kafka-server avro-labs]$ cd avro-maven
```

- Ahora creamos el pom.xml que permitirá generar el artefacto
- Lo generamos con el siguiente contenido

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.kafka</groupId>
  <artifactId>avro-tools</artifactId>
  <version>1.0.0</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <source>11</source>
          <target>11</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.avro</groupId>
        <artifactId>avro-maven-plugin</artifactId>
        <version>1.10.0</version>
        <executions>
          <execution>
            <phase>generate-sources</phase>
            <goals>
              <goal>schema</goal>
            </goals>
            <configuration>
              <sourceDirectory>${project.basedir}/src/main/avro</sourceDirectory>
              <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.apache.avro</groupId>
      <artifactId>avro</artifactId>
      <version>1.10.0</version>
    </dependency>
  </dependencies>
</project>

```

- El resultado del directorio avro-maven debe ser el siguiente:

```
[kafka@kafka-server schemas]$ tree .
```

```
.  
└── pom.xml  
└── src  
    └── main  
        └── avro  
            └── UserCommand.avsc
```

- Una vez generada la estructura del proyecto, solo tenemos que introducir los esquemas en el directorio src/main/avro y se generará todo en un nuevo directorio llamado src/main/java.
- Para generar el artefacto, usamos el siguiente comando:

```
[kafka@kafka-server avro-maven]$ mvn generate-sources  
[INFO] Scanning for projects...  
[INFO]  
[INFO] -----< com.kafka:avro-tools >-----  
[INFO] Building avro-tools 1.0.0  
[INFO] -----[ jar ]-----  
[INFO]  
[INFO] --- avro-maven-plugin:1.10.0:schema (default) @ avro-tools ---  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 1.818 s  
[INFO] Finished at: 2020-10-14T16:25:57+02:00  
[INFO] -----
```

- Si volvemos a listar el contenido:

```
[kafka@kafka-server avro-maven]$ tree .
```

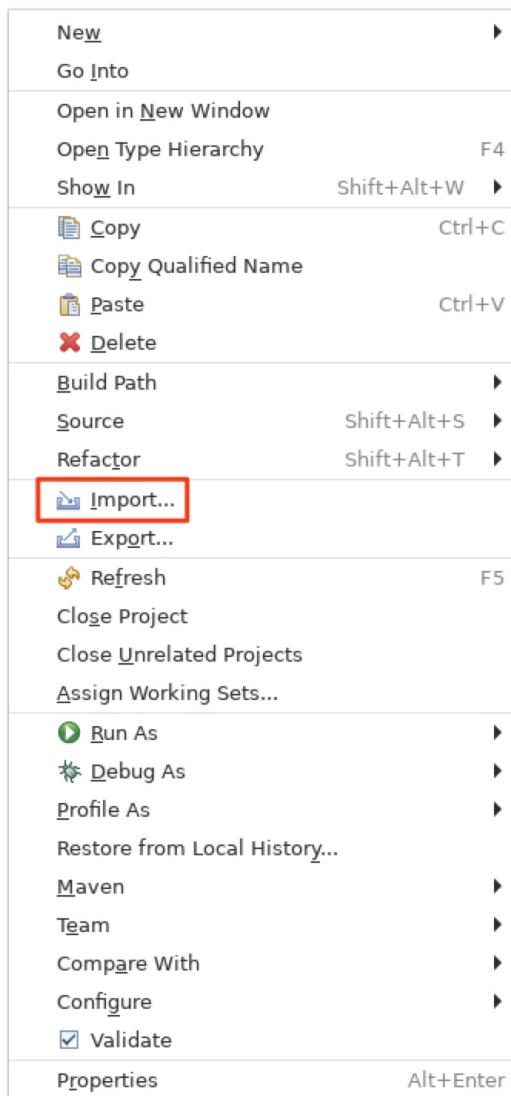
```
.  
└── pom.xml  
└── src  
    └── main  
        ├── avro  
        │   └── UserCommand.avsc  
        └── java  
            └── com  
                └── curso  
                    └── kafka  
                        └── avro  
                            └── model  
                                └── UserCommand.java
```

9 directories, 3 files

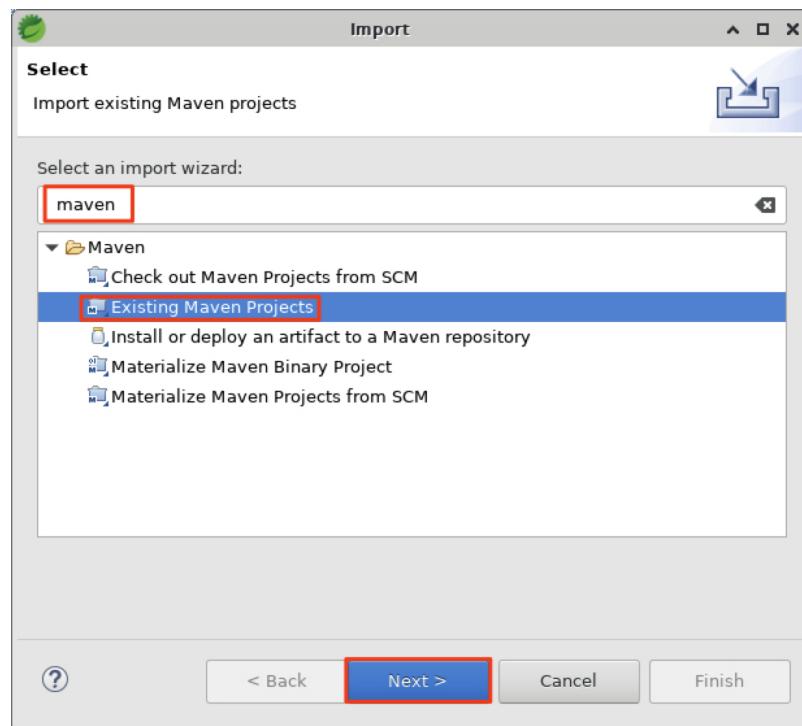
- Vemos como resultado la misma clase java que en el ejemplo anterior.

12.5.3.3. Uso de clase avro

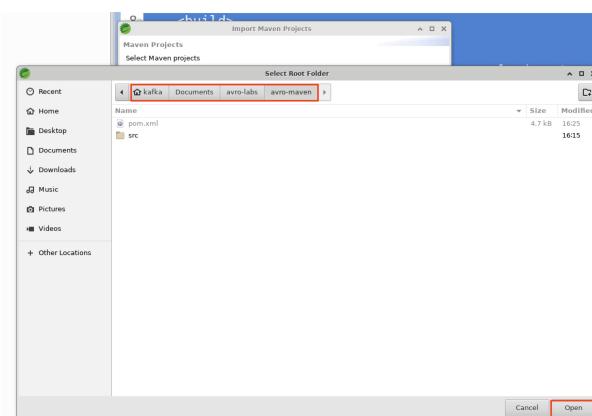
- Para poder usarla, vamos a importar el proyecto.
- Para ello, abrimos el eclipse/STS e importamos el proyecto como proyecto maven, buscando el directorio donde está el pom.xml
- Pulsamos botón derecho en el panel de Package Explorer o en el menú file buscando la opción import...



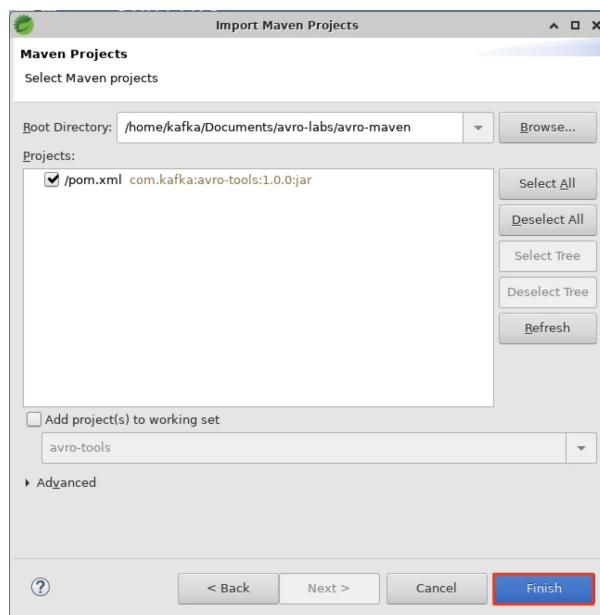
- En la caja superior de búsqueda pulsamos **maven**
- Seleccionamos la opción de **Existing Maven Projects**



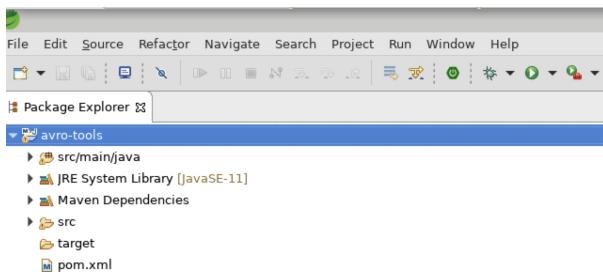
- Pulsamos en buscar, vamos a la carpeta Documents/avro-labs/avro-maven y pulsamos open



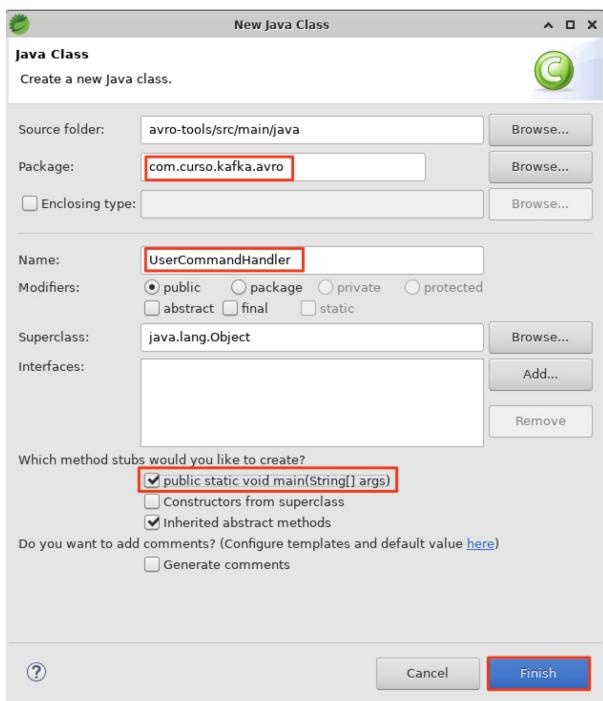
- En el siguiente diálogo aparecerá el pom.xml seleccionado.
- Pulsamos finish



- El proyecto aparecerá importado como avro-tools



- Ahora vamos a crear una nueva clase Main llamada UserCommandHandler.
- Para ello pulsamos en new class y rellenamos:
 - package: com.curso.kafka.avro
 - Name: UserCommandHandler
 - public static void main: true



```

package com.curso.kafka.avro;

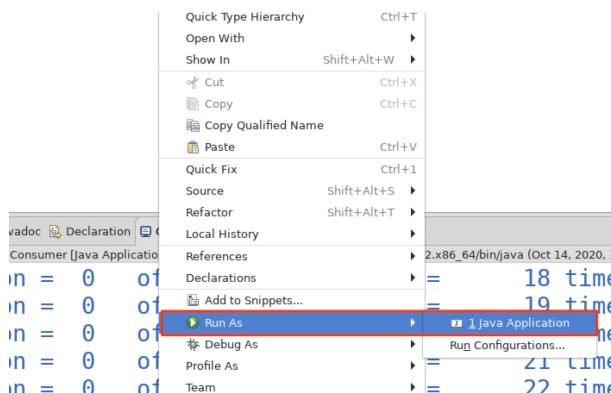
import com.curso.kafka.avro.model.UserCommand;

public class UserCommandHandler {

    public static void main(String[] args) {
        UserCommand command = UserCommand.newBuilder()
            .setCommand("Get new phone!")
            .build();
        System.out.println(command.toString());
    }
}

```

- Con newBuilder creamos un constructor de objetos, y le agregamos los atributos o campos que hayamos declarado.
- Por último, con el build construimos el objeto
- Al pasarlo a toString comprobamos el resultado de construir el objeto con avro y ver el resultado de la creación del objeto.
- Ejecutamos la clase pulsando encima botón derecho y run as → Java Application



```

SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
{"command": "Get new phone!"}

```

- Podemos observar como el resultado es el correcto.
- Ahora comentamos la línea de setCommand. Vamos a dejarlo sin rellenar para comprobar su comportamiento

```

package com.curso.kafka.avro;

import com.curso.kafka.avro.model.UserCommand;

public class UserCommandHandler {

    public static void main(String[] args) {
        UserCommand command = UserCommand.newBuilder()
            //.setCommand("Get new phone!")
            .build();
        System.out.println(command.toString());
    }

}

```

- Ejecutamos de nuevo para ver el resultado

```

SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Exception in thread "main" Path in schema: --> command
        at org.apache.avro.generic.GenericData.getDefaultValue(GenericData.java:1176)
        at org.apache.avro.data.RecordBuilderBase.defaultValue(RecordBuilderBase.java:138)
        at com.curso.kafka.avro.model.UserCommand$Builder.build(UserCommand.java:244)
        at com.curso.kafka.avro.UserCommandHandler.main(UserCommandHandler.java:10)

```

- En este caso, el command es un campo obligatorio en el esquema.

12.6. Esquemas en Avro

- Si lo que queremos es un tipo primitivo, el esquema que habría que generar es el siguiente:

```
{  
  "type": "string"  
}
```

- Como normalmente no es el caso, lo común es que queramos tipos complejos.
- Para ello generamos un esquema similar al siguiente

```
{  
  "name": "Esquema",  
  "namespace": "com.curso.kafkaapp.model.avro",  
  "type": "record",  
  "fields": [  
    {  
      "name": "EsquemaComplejo",  
      "type": "record",  
      "fields": [ ... ]  
    },  
    ...  
  ]  
}
```

- La representación del **name** será el nombre de la clase
- El **namespace** representa el paquete donde se creará
- El **type** indica el tipo de dato a almacenar, en este caso record, que es un tipo complejo.
- Los **fields** indican los campos, que pueden ser simples o complejos como en el ejemplo.
- Algunos tipos de datos permiten solucionar problemas básicos como tipos de datos cerrados (Colores, días de la semana, meses del año, etc)

```
{
  "name": "Esquema",
  "namespace": "com.curso.kafkaapp.model.avro",
  "type": "record",
  "fields: [
    {
      "name": "diasDeLaSemana",
      "type": "enum",
      "symbols": ["LUN", "MAR", "MIE", "JUE", "VIE", "SAB", "DOM"]
    },
    ...
  ]
}
```

- Estos campos declarados son obligatorios, sin embargo, es necesario crear campos opcionales, para ello, declaramos el doble tipo:

```
{
  "name": "Esquema",
  "namespace": "com.curso.kafkaapp.model.avro",
  "type": "record",
  "fields: [
    {
      "name": "campoOpcional",
      "type": ["null", "string"]
    },
    ...
  ]
}
```

- En este caso, el campo puede ser o no declarados
- Si queremos por ejemplo, crear un campo con un tamaño concreto para un campo, como un hash md5, usamos fixed

```
{  
  "name": "Esquema",  
  "namespace": "com.curso.kafkaapp.model.avro",  
  "type": "record",  
  "fields": [  
    {  
      "name": "campoFixedParaMd5",  
      "type": "fixed",  
      "size": 16  
    },  
    ...  
  ]  
}
```

12.6.1. Lab: Esquemas en Avro

- Vamos a generar un esquema de clima, donde un productor y un consumidor van a enviar y recibir mensajes basados en el.

12.6.1.1. Generación del esquema

- Para ello, vamos a crear un esquema en nuestro proyecto java llamado Clima.avsc

Clima.avsc versión 1

```
{
  "name": "Clima",
  "namespace": "com.curso.kafka.avro.model",
  "type": "record",
  "fields": [
    {
      "name": "id",
      "type": "long"
    },
    {
      "name": "nombre",
      "type": "string"
    },
    {
      "name": "datos",
      "type": "record",
      "fields": [
        {
          "name": "temp",
          "type": "float"
        },
        {
          "name": "presion",
          "type": "int"
        },
        {
          "name": "humedad",
          "type": "int"
        },
        {
          "name": "tempMin",
          "type": "float"
        },
        {
          "name": "tempMax",
          "type": "float"
        }
      ]
    }
  ]
}
```

- Los datos que tenemos en este esquema son un objeto complejo.
- Podemos poner tantos datos complejos o con la profundidad que queramos, sin embargo, su gestión se vuelve complicada.
- Para solucionarlo, vamos a crear un nuevo esquema llamado Datos.avsc con el contenido del campo datos.
- El namespace debe estar en el mismo sitio que el padre para que puedan encontrarse

Datos.avsc

```
{
  "name": "datos",
  "namespace": "com.curso.kafka.avro.model",
  "type": "record",
  "fields": [
    {
      "name": "temp",
      "type": "float"
    },
    {
      "name": "presion",
      "type": "int"
    },
    {
      "name": "humedad",
      "type": "int"
    },
    {
      "name": "tempMin",
      "type": "float"
    },
    {
      "name": "tempMax",
      "type": "float"
    }
  ]
}
```

- Ahora sustituimos el valor del tipo de datos en el esquema Clima.avsc

Clima.avsc versión 2

```
{  
    "name": "Clima",  
    "namespace": "com.curso.kafka.avro.model",  
    "type": "record",  
    "fields": [  
        {  
            "name": "id",  
            "type": "long"  
        },  
        {  
            "name": "nombre",  
            "type": "string"  
        },  
        {  
            "name": "datos",  
            "type": "Datos"  
        }  
    ]  
}
```

- Vamos a adelantarnos y creamos un nuevo esquema detalles que usaremos en clima

Detalles.avsc

```
{  
    "name": "Detalles",  
    "namespace": "com.curso.kafka.avro.model",  
    "type": "record",  
    "fields": [  
        {  
            "name": "id",  
            "type": "long"  
        },  
        {  
            "name": "principal",  
            "type": "string"  
        },  
        {  
            "name": "descripcion",  
            "type": "string"  
        },  
        {  
            "name": "icono",  
            "type": "string"  
        }  
    ]  
}
```

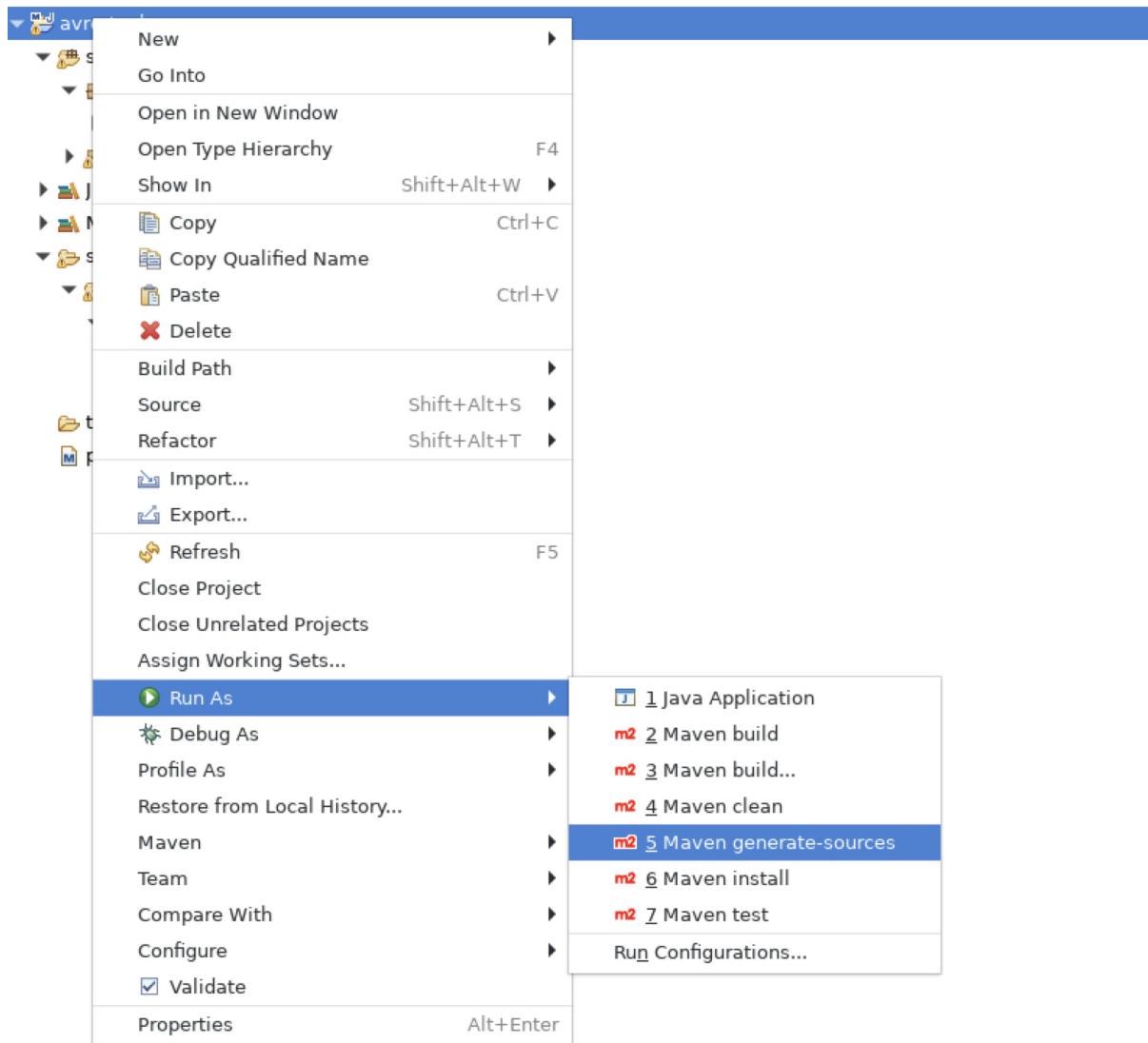
- Modificamos de nuevo el Clima.avsc para que genere el resultado final

```
{
  "name": "Clima",
  "namespace": "com.curso.kafka.avro.model",
  "type": "record",
  "fields": [
    {
      "name": "id",
      "type": "long"
    },
    {
      "name": "nombre",
      "type": "string"
    },
    {
      "name": "datos",
      "type": "Datos"
    },
    {
      "name": "detalles",
      "type": {
        "type": "array",
        "items": {
          "name": "detalles",
          "type": "Detalles"
        }
      }
    }
  ]
}
```

- Hemos agregado un array de un tipo complejo, en este caso de detalles.

12.6.1.2. Compilación de las clases

- Para ello, ejecutamos desde el directorio pom el comando mvn **generate-sources**



```
[INFO] Scanning for projects...
[INFO]
[INFO] [1m-----< 0;36mcom.kafka:avro-tools[0;1m >-----][m
[INFO] [1mBuilding avro-tools 1.0.0[m
[INFO] [1m-----[ jar ]-----[m
[INFO]
[INFO] [1m--- [0;32mavro-maven-plugin:1.10.0:schema[m [1m(default)[m @ [36mavro-tools[0;1m ---[m
[INFO] [1m-----[m
[INFO] [1;31mBUILD FAILURE[m
[INFO] [1m-----[m
[INFO] Total time: 1.662 s
[INFO] Finished at: 2020-10-14T17:43:42+02:00
[INFO] [1m-----[m
[ERROR] Failed to execute goal [32morg.apache.avro:avro-maven-plugin:1.10.0:schema[m [1m(default)[m on project
[36mavro-tools[m: [1;31mExecution default of goal org.apache.avro:avro-maven-plugin:1.10.0:schema failed: "Datos" is
not a defined name. The type of the "datos" field must be a defined name or a {"type": ...} expression.[m -> [1m[Help
1][m
[ERROR]
[ERROR] To see the full stack trace of the errors, re-run Maven with the [1m-e[m switch.
[ERROR] Re-run Maven using the [1m-X[m switch to enable full debug logging.
[ERROR]
[ERROR] For more information about the errors and possible solutions, please read the following articles:
[ERROR] [1m[Help 1][m http://cwiki.apache.org/confluence/display/MAVEN/PluginExecutionException
```

- Si comprobamos el error que nos sale, nos indica que Datos no es un tipo definido
- Para solucionarlo, debemos importar los esquemas para que sepa donde están.
- Para ello, modificamos el pom.xml y agregamos los ficheros que nos faltan.

```

<plugin>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro-maven-plugin</artifactId>
    <version>1.10.0</version>
    <executions>
        <execution>
            <phase>generate-sources</phase>
            <goals>
                <goal>schema</goal>
            </goals>
            <configuration>
                <sourceDirectory>${project.basedir}/src/main/avro</sourceDirectory>
                <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
                <imports>
                    <import>${project.basedir}/src/main/avro/Datos.avsc</import>
                    <import>${project.basedir}/src/main/avro/Detalles.avsc</import>
                </imports>
            </configuration>
        </execution>
    </executions>
</plugin>

```

- Nos fijamos en los imports del configuration, esta es la sección que hay que agregar

```

<imports>
    <import>${project.basedir}/src/main/avro/Datos.avsc</import>
    <import>${project.basedir}/src/main/avro/Detalles.avsc</import>
</imports>

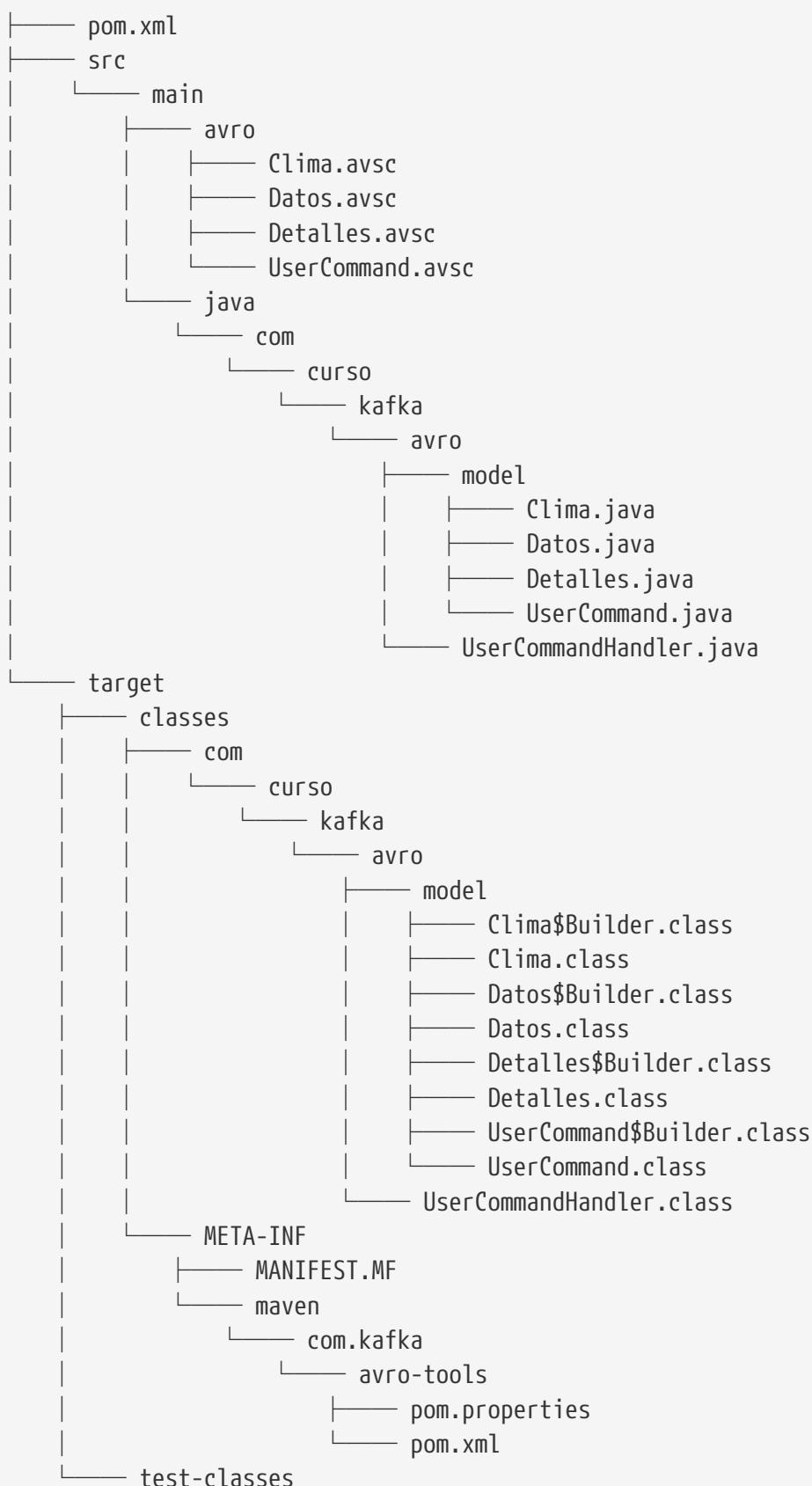
```

- Lo que estamos diciendo es que antes de comenzar a generar los esquemas, debe generar primero Datos.avsc y Detalles.avsc
- De esa forma tendrá disponible los recursos para generar Clima.avsc

```
[INFO] Scanning for projects...
[INFO]
[INFO] [1m-----< @36com.kafka:avro-tools@1m >-----[m
[INFO] [1mBuilding avro-tools 1.0.0[m
[INFO] [1m-----[ jar ]-----[m
[INFO]
[INFO] [1m-- [0;32mavro-maven-plugin:1.10.0:schema[m [1m(default)[m @ [36mavro-tools@1m --[m
[INFO] Importing File: /home/kafka/Documents/avro-labs/avro-maven/src/main/avro/Datos.avsc
[INFO] Importing File: /home/kafka/Documents/avro-labs/avro-maven/src/main/avro/Detalles.avsc
[INFO] [1m-----[m
[INFO] [1;32mBUILD SUCCESS[m
[INFO] [1m-----[m
[INFO] Total time: 1.497 s
[INFO] Finished at: 2020-10-14T17:50:19+02:00
[INFO] [1m-----
```

- Ahora si que los ha generado correctamente
- Podemos comprobar como ahora se han generado las clases java.

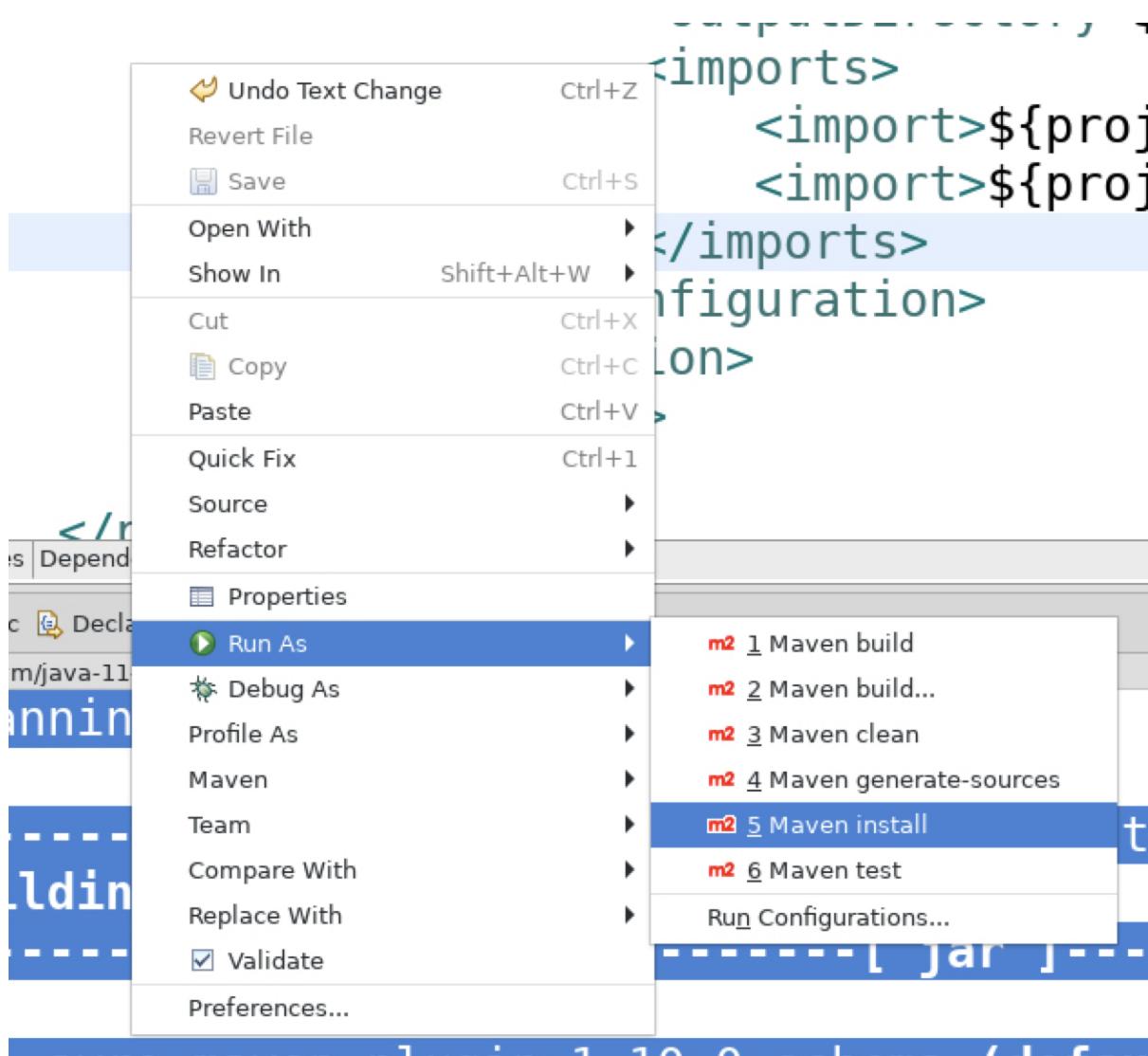
```
[kafka@kafka-server avro-maven]$ tree .
```



21 directories, 22 files

- Podemos ver el main java con las clases nuevas generadas.

- El target con las clases compiladas.
- Por último, vamos a generar el artefacto para que podamos usarla en otros proyectos
- Para ello lanzamos el comando mvn install

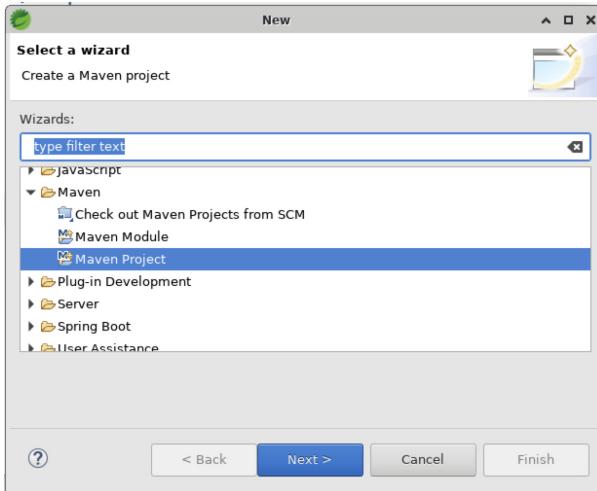


- Comprobamos que la salida es correcta.

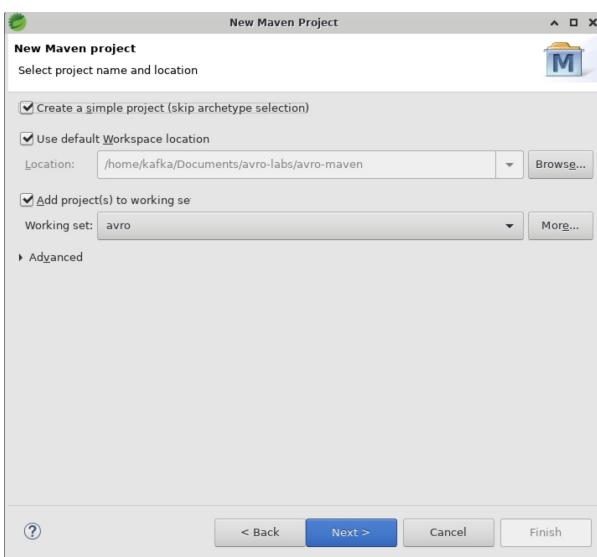
```
[INFO] Scanning for projects...
[INFO]
[INFO] [1m-----< 0;36mcom.kafka:avro-tools 0;1m >-----][m
[INFO] [1mBuilding avro-tools 1.0.0[m
[INFO] [1m-----[ jar ]-----][m
[INFO]
[INFO]
...
[INFO] [1m-----][m
[INFO] [1;32mBUILD SUCCESS[m
[INFO] [1m-----][m
[INFO] Total time: 6.744 s
[INFO] Finished at: 2020-10-14T17:53:31+02:00
[INFO] [1m-----]
```

12.6.1.3. Desarrollo de un productor

- Para poder aprovechar los esquemas vamos a generar un productor.
- Para ello tenemos que usar el sistema que ya conocemos.
- Creamos un nuevo proyecto llamado ProductorClima por medio de New → Project → Maven → Maven Project

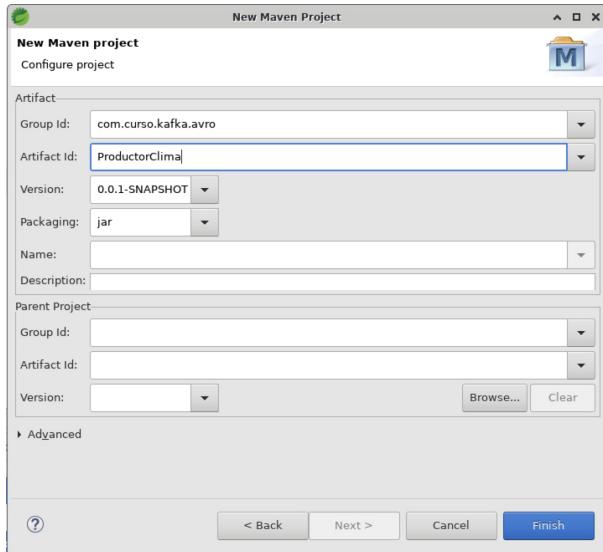


- Creamos un proyecto simple



- Indicamos los siguientes datos:

- Group ID: com.curso.kafka.avro
- Artifact ID: ProductorClima
- Version: 0.0.1-SNAPSHOT
- Packaging: jar



- Ahora abrimos el pom.xml y agregamos la dependencia a java 11

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.0</version>
      <configuration>
        <source>11</source>
        <target>11</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

- Primero agregamos las dependencias del proyecto del modelo llamado **avro-tools**

```
<dependency>
  <groupId>com.kafka</groupId>
  <artifactId>avro-tools</artifactId>
  <version>1.0.0</version>
</dependency>
```

- Agregamos las dependencias de kafka client para poder generar el productor

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.7.5</version>
</dependency>
```

- Vamos a agregar otra dependencia que nos servirá para obtener datos más adelante sobre información meteorológica que obtendremos para agregar al topic desde el productor.

```
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.1</version>
</dependency>
```

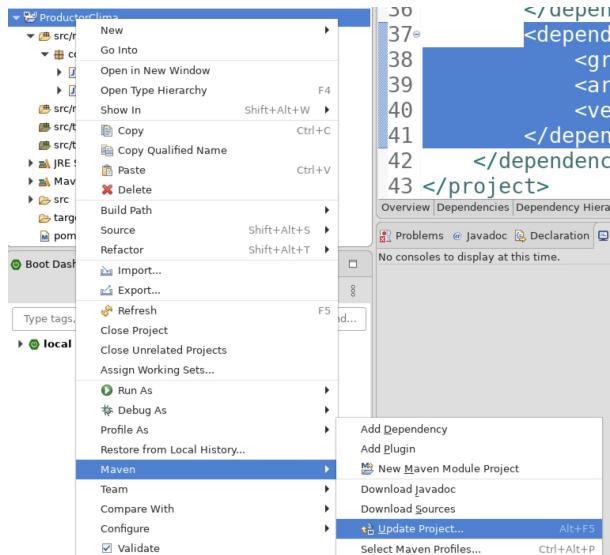
- Nos quedaría un pom.xml como el siguiente

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.curso.kafka.avro</groupId>
  <artifactId>ProductorClima</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <source>11</source>
          <target>11</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>com.kafka</groupId>
      <artifactId>avro-tools</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>2.6.0</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.5</version>
    </dependency>
    <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
      <version>2.8.1</version>
    </dependency>
  </dependencies>
</project>

```

- Una vez agregado, actualizamos el proyecto para que obtenga los cambios realizados en el pom.xml



- Vamos a crear una clase de apoyo llamada OpenWeatherMap que permite obtener la información del servicio climático.

```

package com.curso.kafka.productorclima;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.curso.kafka.avro.model.Clima;
import com.curso.kafka.avro.model.Datos;
import com.curso.kafka.avro.model.Detalles;
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;;

public class OpenWeatherMap {
    private static String API_KEY = "<API_KEY>";

    private static Map<String, Object> jsonToMap(String responseString) {
        return new Gson().fromJson(responseString, new TypeToken<HashMap<String, Object>>() {
            }.getType());
    }

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static Clima getWeatherFromOpenWeatherMap(String city) throws IOException {

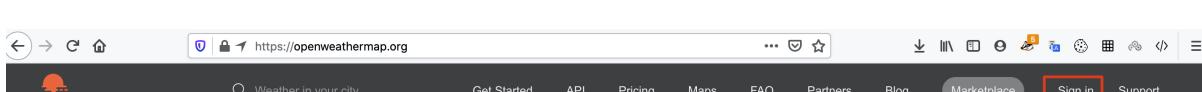
        String url = "http://api.openweathermap.org/data/2.5/weather?q=" + city + "&appid=" + API_KEY;
        StringBuilder result = new StringBuilder();
        URL urlRequest = new URL(url);
        URLConnection connection = urlRequest.openConnection();
        BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String line = null;
        while ((line = reader.readLine()) != null) {
            result.append(line);
        }
        reader.close();
        System.out.println(result);
        Map resultMap = jsonToMap(result.toString());

        Map mainMap = jsonToMap(resultMap.get("main").toString());
        List<Object> list = (List<Object>) resultMap.get("weather");
        List<Detalles> detalles = new ArrayList();
        for (Object detail : list) {
            Map<String, Object> detailMap = (Map) detail;
            detalles.add(Detalles.newBuilder().setId((long) Double.parseDouble(detailMap.get("id").toString()))
                .setPrincipal(detailMap.get("main").toString()).setIcono(detailMap.get("icon").toString())
                .setDescripcion(detailMap.get("description").toString()).build());
        }

        Datos datos = Datos.newBuilder().setPresion((int) Double.parseDouble(mainMap.get("pressure").toString()))
            .setHumedad((int) Double.parseDouble(mainMap.get("humidity").toString()))
            .setTemp((int) Double.parseDouble(mainMap.get("temp").toString()))
            .setTempMax((int) Double.parseDouble(mainMap.get("temp_max").toString()))
            .setTempMin((int) Double.parseDouble(mainMap.get("temp_min").toString())).build();
        Clima clima = Clima.newBuilder().setId((long) Double.parseDouble(resultMap.get("id").toString()))
            .setNombre(resultMap.get("name").toString()).setDatos(datos).setDetalles(detalles).build();
        return clima;
    }
}

```

- Como podemos observar, es necesario un API KEY para poder realizar las peticiones.
- El API KEY es gratuito y se permiten hasta 60 peticiones por minuto como máximo. Mas allá de esas peticiones, el servicio impide su consulta.
- Para llenar el API KEY nos dirigimos a la página web:
 - <https://openweathermap.org/> y pulsamos en **Sign in**



OpenWeather global services

Weather forecasts, nowcasts and history in fast and elegant way

2 Billion Forecasts Per Day
2,500 new subscribers a day

2,600,000 customers
20+ weather APIs



- Pulsamos en **Create an Account**

Sign In To Your Account

Enter email

Password

Remember me

Submit

Not registered? [Create an Account.](#)

Lost your password? [Click here to recover.](#)

- Indicamos los datos personales y un correo electrónico para recibir la confirmación de alta en la web

Create New Account

alumno

@gmail.com

.....

.....

We will use information you provided for management and administration purposes, and for keeping you informed by mail, telephone, email and SMS of other products and services from us and our partners. You can proactively manage your preferences or opt-out of communications with us at any time using Privacy Centre. You have the right to access your data held by us or to request your data to be deleted. For full details please see the OpenWeather [Privacy Policy](#).

- I am 16 years old and over
- I agree with [Privacy Policy](#), [Terms and conditions of sale](#) and [Websites terms and conditions of use](#)

I consent to receive communications from OpenWeather Group of Companies and their partners:

- Rellenamos el reCAPTCHA y pulsamos en **Create Account**

I consent to receive communications from OpenWeather Group of Companies and their partners:

- System news (API usage alert, system update, temporary system shutdown, etc)
- Product news (change to price, new product features, etc)
- Corporate news (our life, the launch of a new service, etc)

No soy un robot


reCAPTCHA
Privacidad - Términos

Create Account

- Nos preguntan por el propósito e indicamos lo que corresponda. En el ejemplo **Education/Science**
- Pulsamos save

How and where will you use our API?

X

Hi! We are doing some housekeeping around thousands of our customers. Your impact will be much appreciated. All you need to do is to choose in which exact area you use our services.

Company

* Purpose

Education/Science

Cancel

Save

- Veremos una confirmación de envío al correo que utilizamos para dar de alta la cuenta

The screenshot shows the OpenWeatherMap account settings interface. At the top, there's a navigation bar with links like 'Get Started', 'Pricing', 'Maps', 'FAQ', 'Partners', 'Blog', 'Marketplace', and 'alumn...'. Below the navigation, a green banner displays the message: 'We have sent the confirmation link to [rubengomez78@gmail.com](#). Please check your email.' A red box highlights this message. The main content area has a sub-navigation bar with links: 'New Products' (which is underlined), 'Services', 'API keys', 'Billing plans', 'Payments', 'Block logs', 'My orders', and 'My profile'. To the left, there's a decorative image of a sunset or sunrise. To the right, the text 'Historical weather for any location' is displayed in orange, followed by a subtext: 'Our new technology, Time Machine, has allowed us to enhance the data in the [Historical Weather Collection](#).'. Below this, there's a section titled 'What is the historical weather for ANY location?' with a 'Read more' link.

- Nos dirigimos al correo electrónico y pulsamos en **Verify your email**



Dear Customer!

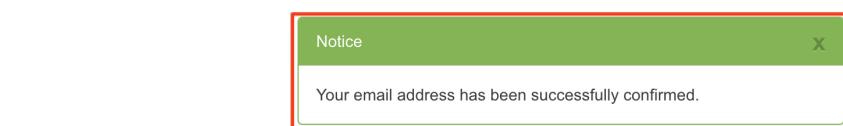
Thank you for choosing [OpenWeatherMap](#)!

Please confirm your email address to help us ensure your account is always protected.

[Verify your email](#)

For further technical questions and support, please contact us at info@openweathermap.org

- Se abre el navegador indicando que el email ha sido validado.



New Products Services API keys Billing plans Payments Block logs My orders My profile



Historical weather for any location

Our new technology, Time Machine, has allowed us to enhance the data in the [Historical Weather Collection](#).

- Historical weather data available for **ANY** coordinate
- The depth of historical data have been extended to **40 YEARS**

You can download data from [Personal account](#) or [contact us](#) to order it. The price is highly competitive - only **10\$** per location!

[Learn more](#)

[Go to purchase](#)

- En el menú del usuario, desplegamos y pulsamos en **My API Keys**

This is the detail guide about our services and how to start working with them.

How to migrate from Dark Sky API to OpenWeather One Call API
With our One Call API you can easily migrate from the [Dark Sky weather API](#) and get [free access](#) to main weather parameters such as current weather, forecast and historical weather with only one call to API.

[Read the Guide](#)

- Veremos una clave llamada **Default**.
- Copiamos la clave y modificamos la clase java OpenWeatherMap usando nuestra API Key.

New Products Services API keys Billing plans Payments Block logs My orders My profile

You can generate as many API keys as needed for your subscription. We accumulate the total load from all of them.

Key	Name	Create key
<input type="text" value="Default"/>	Default <input type="button" value="Edit"/>	<input type="button" value="Generate"/>
<input type="text" value="API key name"/>		

```
...
public class OpenWeatherMap {
    private static String API_KEY = "AQUI PONEMOS NUESTRA API_KEY DE LA PAGINA WEB";

    private static Map<String, Object> jsonToMap(String responseString) {
        return new Gson().fromJson(responseString, new TypeToken<HashMap<String, Object>>() {
            .getType());
    }
...
}
```

- Ahora creamos la entrada del productor. Para ello, creamos una nueva clase con la siguiente

información

```
package com.curso.kafka.productorclima;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.ByteArraySerializer;
import org.apache.kafka.common.serialization.StringSerializer;

import com.curso.kafka.avro.model.Clima;

public class Productor {

    public static final String CITY = "madrid";
    public static final String TOPIC = "avro-clima";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, ByteArraySerializer.class.getName());

        KafkaProducer<String,byte[]> producer = new KafkaProducer<>(properties);

        Thread shutdownHook = new Thread(producer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);

        while(true) {
            Clima clima = OpenWeatherMap.getWeatherFromOpenWeatherMap(CITY);
            byte[] value = serializeClima(clima);
            ProducerRecord<String, byte[]> record = new ProducerRecord<>(TOPIC, CITY, value);
            producer.send(record);
            Thread.sleep(1500);
        }
    }

    private static byte[] serializeClima(Clima clima) throws IOException {
        return clima.toByteBuffer().array();
    }
}
```

- Si observamos detenidamente, comprobamos como indicamos a Kafka, que queremos guardar una cadena como clave, que es la ciudad, y un array de bits como valor

```
KafkaProducer<String,byte[]> producer = new KafkaProducer<>(properties);
```

- Como propiedades, indicamos que se va a serializar una cadena y un ByteArray

```
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, ByteArraySerializer.class.getName());
```

- El objeto que manejamos es el que avro ha generado

```
import com.curso.kafka.avro.model.Clima;

public class Productor {
...
    Clima clima = OpenWeatherMap.getWeatherFromOpenWeatherMap(CITY);
...
}
```

- Luego serializamos manualmente con avro el objeto antes de enviarlo

```
...
public class Productor {

    public static final String CITY = "madrid";
    public static final String TOPIC = "avro-clima";

    public static void main(String[] args) throws InterruptedException, IOException {
...
        while(true) {
...
            byte[] value = serializeClima(clima);
            ProducerRecord<String, byte[]> record = new ProducerRecord<>(TOPIC, CITY, value);
...
        }
    }

    private static byte[] serializeClima(Clima clima) throws IOException {
        return clima.toByteBuffer().array();
    }

}
```

- Usamos el método clima.toByteBuffer().array() para obtener el array de bits que vamos a enviar.
- También ponemos un sleep para evitar no superar las 60 peticiones por minuto de máximo que tenemos

```
Thread.sleep(1500);
```

- De esa forma ya tenemos el productor funcional.

12.6.1.4. Generación de consumidor

- Para generar el consumidor creamos un nuevo proyecto igual que el anterior pero con los datos:
 - Group ID: com.curso.kafka.avro

- Artifact ID: ConsumidorClima
- Version: 0.0.1-SNAPSHOT
- Packaging: jar
- El pom.xml será el mismo que el anterior, con la librería de avro-tools que contiene nuestro modelo y las librerías clientes.
- Esta vez ya no nos hace falta el GSon, ya que tratamos solo con el objeto avro.
- Nos quedará un pom como el que se muestra

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.curso.kafka.avro</groupId>
  <artifactId>ConsumidorClima</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <source>11</source>
          <target>11</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>com.kafka</groupId>
      <artifactId>avro-tools</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>2.6.0</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.5</version>
    </dependency>
  </dependencies>
</project>

```

- Ahora creamos el cliente:

```

package com.curso.kafka.consumidorclima;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.ByteArrayDeserializer;
import org.apache.kafka.common.serialization.StringDeserializer;

import com.curso.kafka.avro.model.Clima;

public class Consumidor {

    public static final String TOPIC = "avro-clima";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, ByteArrayDeserializer.class.getName());
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "consumidorClima");

        KafkaConsumer<String, byte[]> consumer = new KafkaConsumer<>(properties);

        Thread shutdownHook = new Thread(consumer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);

        consumer.subscribe(Collections.singletonList(TOPIC));
        while(true) {
            ConsumerRecords<String, byte[]> records = consumer.poll(Duration.ofMillis(100));
            for(ConsumerRecord<String, byte[]> record : records) {
                Clima clima = deserializeClima(record.value());
                System.out.println("ID: " + record.key() + " - value: " + clima.toString());
            }
        }
    }

    private static Clima deserializeClima(byte[] array) throws IOException {
        return Clima.fromByteBuffer(ByteBuffer.wrap(array));
    }
}

```

- A la hora de generar el consumidor, vamos a suscribirlo al mismo topic.
- En este caso, vamos a deserializar en la configuración el string y el array de bits

```

...
public class Consumidor {

    public static final String TOPIC = "avro-clima";

    public static void main(String[] args) throws InterruptedException, IOException {
    ...
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, ByteArrayDeserializer.class.getName());
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "consumidorClima");

        KafkaConsumer<String, byte[]> consumer = new KafkaConsumer<>(properties);
    ...
}

```

- A la hora de deserializarlo, usamos de nuevo el mismo sistema por medio del método de deserialización de avro que posee

```

package com.curso.kafka.consumidorclima;

...
import com.curso.kafka.avro.model.Clima;

public class Consumidor {

    public static final String TOPIC = "avro-clima";

    ...
    private static Clima deserializeClima(byte[] array) throws IOException {
        return Clima.fromByteBuffer(ByteBuffer.wrap(array));
    }
}

```

- Por último, mostramos los resultados para comprobar que la deserialización se hace correctamente

```

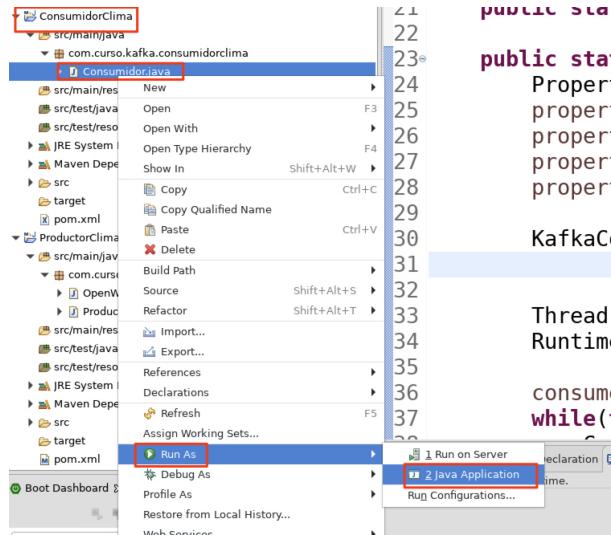
...
    consumer.subscribe(Collections.singletonList(TOPIC));
    while(true) {
        ConsumerRecords<String, byte[]> records = consumer.poll(Duration.ofMillis(100));
        for(ConsumerRecord<String, byte[]> record : records) {
            Clima clima = deserializeClima(record.value());
            System.out.println("ID: " + record.key() + " - value: " + clima.toString());
        }
    }
...

```

- Recorremos los records, y por cada array de bits sacamos un objeto Clima con el resultado

12.6.1.5. Prueba de ejecución.

- Para probarlo, primero conectamos el consumidor.
- Pulsamos en botón derecho en la clase Consumidor.java y pulsamos en Run as → Java Application



- Comprobamos como el sistema no está devolviendo información.

```
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.consumer.ConsumerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
```

- Ahora arrancamos el productor por el mismo procedimiento y observamos la consola del consumidor
- Comprobamos como el productor obtiene la información

```
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.producer.ProducerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
{"coord":{"lon":-3.7,"lat":40.42}, "weather":[{"id":801,"main":"Clouds","description":"few clouds","icon":"02d"}], "base":"stations", "main":{"temp":287.17,"feels_like":283.67,"temp_min":286.48,"temp_max":288.15,"pressure":1018,"humidity":44}, "visibility":10000, "wind":{"speed":2.6,"deg":220}, "clouds":{"all":20}, "dt":1602761250, "sys": {"type":1,"id":6421,"country":"ES","sunrise":1602743147,"sunset":1602783292}, "timezone":7200, "id":3117735, "name":"Madrid", "cod":200}
{"coord":{"lon":-3.7,"lat":40.42}, "weather":[{"id":801,"main":"Clouds","description":"few clouds","icon":"02d"}], "base":"stations", "main":{"temp":287.17,"feels_like":283.67,"temp_min":286.48,"temp_max":288.15,"pressure":1018,"humidity":44}, "visibility":10000, "wind":{"speed":2.6,"deg":220}, "clouds":{"all":20}, "dt":1602761250, "sys": {"type":1,"id":6421,"country":"ES","sunrise":1602743147,"sunset":1602783292}, "timezone":7200, "id":3117735, "name":"Madrid", "cod":200}
{"coord":{"lon":-3.7,"lat":40.42}, "weather":[{"id":801,"main":"Clouds","description":"few clouds","icon":"02d"}], "base":"stations", "main":{"temp":287.17,"feels_like":283.67,"temp_min":286.48,"temp_max":288.15,"pressure":1018,"humidity":44}, "visibility":10000, "wind":{"speed":2.6,"deg":220}, "clouds":{"all":20}, "dt":1602761250, "sys": {"type":1,"id":6421,"country":"ES","sunrise":1602743147,"sunset":1602783292}, "timezone":7200, "id":3117735, "name":"Madrid", "cod":200}
{"coord":{"lon":-3.7,"lat":40.42}, "weather":[{"id":801,"main":"Clouds","description":"few clouds","icon":"02d"}], "base":"stations", "main":{"temp":287.17,"feels_like":283.67,"temp_min":286.48,"temp_max":288.15,"pressure":1018,"humidity":44}, "visibility":10000, "wind":{"speed":2.6,"deg":220}, "clouds":{"all":20}, "dt":1602761250, "sys": {"type":1,"id":6421,"country":"ES","sunrise":1602743147,"sunset":1602783292}, "timezone":7200, "id":3117735, "name":"Madrid", "cod":200}
```

- Ahora nos dirigimos al consumidor

```
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.consumer.ConsumerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
ID: madrid - value: {"id": 3117735, "nombre": "Madrid", "datos": {"temp": 287.0, "presion": 1018, "humedad": 44, "tempMin": 286.0, "tempMax": 288.0}, "detalles": [{"id": 801, "principal": "Clouds", "descripcion": "few clouds", "icono": "02d"}]}
ID: madrid - value: {"id": 3117735, "nombre": "Madrid", "datos": {"temp": 287.0, "presion": 1018, "humedad": 44, "tempMin": 286.0, "tempMax": 288.0}, "detalles": [{"id": 801, "principal": "Clouds", "descripcion": "few clouds", "icono": "02d"}]}
ID: madrid - value: {"id": 3117735, "nombre": "Madrid", "datos": {"temp": 287.0, "presion": 1018, "humedad": 44, "tempMin": 286.0, "tempMax": 288.0}, "detalles": [{"id": 801, "principal": "Clouds", "descripcion": "few clouds", "icono": "02d"}]}
ID: madrid - value: {"id": 3117735, "nombre": "Madrid", "datos": {"temp": 287.0, "presion": 1018, "humedad": 44, "tempMin": 286.0, "tempMax": 288.0}, "detalles": [{"id": 801, "principal": "Clouds", "descripcion": "few clouds", "icono": "02d"}]}
```

- Comprobamos que los datos consumidos son correctos y coinciden con la información que enviamos anteriormente.
 - Por último paramos los dos servicios.

Capítulo 13. Schema Registry

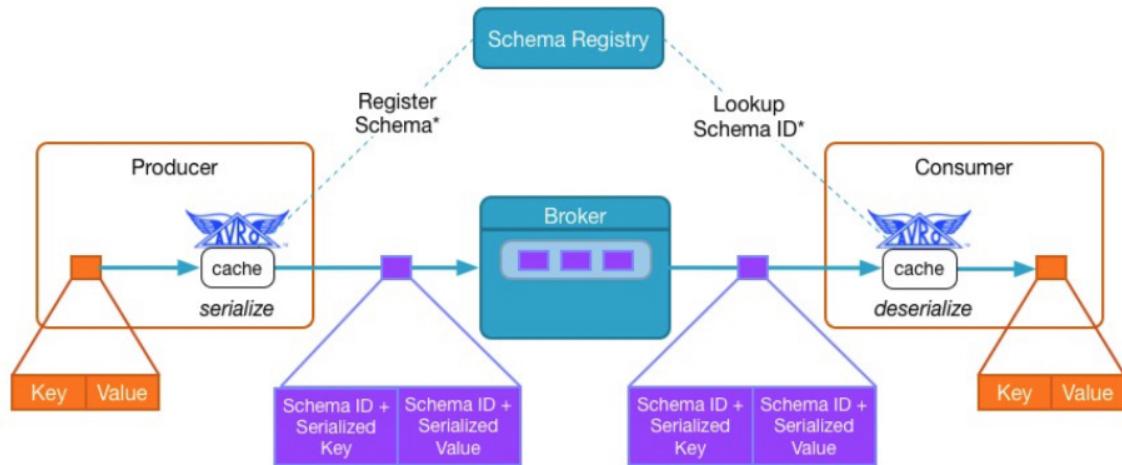
- Si analizamos los anteriores laboratorios, podemos ver como tenemos un acoplamiento entre el productor y el consumidor.
- Este acomplamiento en ecosistemas más complejos no es adecuado, lo que impide un correcto desacoplamiento.
- Para solucionar ese problema tenemos el **Schema Registry**
- Este registro permite:
 - Reforzar los contratos
 - Permite gestionar la evolución de los contratos.
- Este producto no es totalmente público ya que pertenece a Confluent.
- La licencia de uso indica que se puede acceder al código fuente, modificar y distribuir mientras no se haga competencia de los servicios SaaS que ofrece Confluent.
- Todo esto pasa de forma transparente. Lo único que hay que hacer es publicar el serializador y deserializador usando:
 - KakaAvroSerializer
 - KafkaAvroDeserializer
- Por otro lado, el registro necesita almacenar la información en un topic como persistencia.
- El Schema Registry tiene un productor que envia los esquemas y un consumidor para obtener los que tenía previamente.
- Se llama __schemas

13.1. Integración de datos

- Para integrar los datos, avro usa un esquema.
- En Kafka debemos pensar que cientos de productores se conectan a kafka con un esquema propio y cientos de consumidores lo usan para deserializar los datos.
- En resumen, al menos un esquema es usado por un productor y un consumidor.
- Para solucionar el problema, usamos un repositorio de esquemas centralizado.
- Propietario de Confluent
- Provee de un sistema centralizado de gestión de esquemas
- Posee una interfaz RESTful para almacenamiento y obtención de esquemas Avro
- Comprobación de esquemas y lanzamiento de excepciones si los datos no cumplen el esquema
- Permite la mejora de los esquemas según su configuración de compatibilidad.
- Permite evitar mandar el esquema en cada mensaje
- El registro almacena la información en un topic de Kafka
- Es posible acceder al registro via API Rest o API Java.

- Posee herramientas de línea de comandos

13.2. Flujo de trabajo



- Los mensajes clave y valor se serializan de forma independiente
- Los productores serializan los datos y usan preferentemente el ID de esquema
- Los consumidores usan el id de esquema para deserializar los datos
- Los esquemas son cacheados en productores y consumidores y solo mandan el id del esquema

13.3. Ejemplos

- Soporte de los clientes para el *Schema Registry*

```
Properties newProperties = new Properties();
newProperties.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://schemaregistry1:8081");
```

Productor Avro

```
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
// Configuración de clases de serialización
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    io.confluent.kafka.serializers.KafkaAvroSerializer.class);
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    io.confluent.kafka.serializers.KafkaAvroSerializer.class);
// Ruta al Schema Registry
props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://schemaregistry1:8081");
// Creando productor
KafkaProducer<Object, Object> avroProducer = new KafkaProducer<Object, Object>(props);
// Creando objetos Avro
CardSuit suit = new CardSuit("spades");
SimpleCard card = new SimpleCard("spades", "ace");
// Creando el ProducerRecord con objetos Avro
ProducerRecord<Object, Object> record = new ProducerRecord<Object, Object>("my_avro_topic", suit, card);
avroProducer.send(record);
```

Consumidor Avro

```
public class CardConsumer {
    public static void main(String[] args){
        Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "testgroup");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, "io.confluent.kafka.serializers.KafkaAvroDeserializer");
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, "io.confluent.kafka.serializers.KafkaAvroDeserializer");
        props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://schemaregistry1:8081");
        props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, "true");
        KafkaConsumer<CardSuit, SimpleCard> consumer = new KafkaConsumer<>(props);
        consumer.subscribe(Arrays.asList("my_avro_topic"));
        while(true){
            ConsumerRecords<CardSuit, SimpleCard> records = consumer.poll(100);
            for (ConsumerRecord<CardSuit, SimpleCard> record : records) {
                System.out.printf("offset = %d, key = %s, value = %s\n", record.offset(), record.key().getSuit(), record.value().getCard());
            }
        }
    }
}
```

13.4. Lab: Schema Registry

- El objetivo es clonar los proyectos de productor y consumidor modificando las clases para que usen el Schema Registry

13.4.1. Inicio de los servicios

- Primero comprobamos que poseemos todos los servicios levantados.

- En este caso (Iniciados con Docker), sería de la siguiente manera:

```
[kafka@kafka-server avro-maven]$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              NAMES
3f5f09c38891        confluentinc/cp-schema-registry:6.0.0   "/etc/confluent/dock..."   27 hours ago      Up 27 hours       docker-kafka_schema_registry_1
093e9d1c649e        bitnami/kafka:latest          "/opt/bitnami/script..."  2 days ago       Up 27 hours       docker-kafka_kafka1_1
551540ef4ce4        bitnami/kafka:latest          "/opt/bitnami/script..."  2 days ago       Up 27 hours       docker-kafka_kafka2_1
fbe3387824b3        bitnami/kafka:latest          "/opt/bitnami/script..."  2 days ago       Up 27 hours       docker-kafka_kafka3_1
40c2020d1486        bitnami/zookeeper:latest      "/opt/bitnami/script..."  2 days ago       Up 27 hours       docker-kafka_zookeeper_1
```



- Si no está iniciado el schema registry, usamos el comando:

```
[kafka@kafka-server avro-maven]$ docker-compose up -d schema_registry
```

- En el caso de poseer los servicios comprobamos que todos están iniciados correctamente

13.4.2. Adaptación del productor

- Vamos a adaptar el ejercicio anterior para que use el Schema Registry.
- Primero copiamos el proyecto productor y lo pegamos con un nuevo nombre llamado:
 - ProductorClimaSchemaRegistry
- También vamos a modificar el pom.xml para que responda a un nuevo paquete:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.curso.kafka.avro</groupId>
  <artifactId>ProductorClimaSchemaRegistry</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

- Si queremos que Avro se encargue de serializar y deserializar a partir del Schema Registry,

tenemos que agregar el serializador en el productor.

- Para ello agregamos la siguiente entrada

```
<dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>6.0.0</version>
</dependency>
```

- Sin embargo, en cuanto guardamos, el recurso no está disponible, ya que está en otro repositorio.
- Vamos a agregar un nuevo repositorio en el pom

```
<repositories>
    <repository>
        <id>confluent</id>
        <url>http://packages.confluent.io/maven/</url>
    </repository>
</repositories>
```

- Con este nuevo repositorio ya está disponible el paquete de Confluent que nos fallaba.
- El pom.xml quedará de la siguiente manera

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.curso.kafka.avro</groupId>
  <artifactId>ProductorClimaSchemaRegistry</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <source>11</source>
          <target>11</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <repositories>
    <repository>
      <id>confluent</id>
      <url>http://packages.confluent.io/maven/</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency>
      <groupId>com.kafka</groupId>
      <artifactId>avro-tools</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>2.6.0</version>
    </dependency>
    <dependency>
      <groupId>io.confluent</groupId>
      <artifactId>kafka-avro-serializer</artifactId>
      <version>6.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-log4j12</artifactId>
      <version>1.7.5</version>
    </dependency>
    <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
      <version>2.8.1</version>
    </dependency>
  </dependencies>
</project>

```

- En cuanto a la clase Productor, vamos a modificar el paquete y el nombre de la clase por:

- paquete: com.curso.kafka.productorclimaschemaregistry;
- nombre de clase: ProductorSchemaRegistry
- En cuanto a los cambios, vamos a modificar el productor para usar el Schema Registry
- Primero modificamos el topic para apuntar a uno nuevo

```
public static final String TOPIC = "avro-clima-schema-registry";
```

- Por otro lado, vamos a usar un nuevo serializador:

```
import io.confluent.kafka.serializers.KafkaAvroSerializer;
import io.confluent.kafka.serializers.KafkaAvroSerializerConfig;

public class ProductorSchemaRegistry {
...
    public static void main(String[] args) throws InterruptedException, IOException {
...
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());
```

- También debemos agregar una propiedad que usará el serializador, que es la del Schema Registry

```
properties.put(KafkaAvroSerializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");
```

- Apuntamos al servicio que tenemos levantado en localhost:8081
- Ahora modificamos el productor para que envie directamente nuestro objeto Clima

```
KafkaProducer<String, Clima> producer = new KafkaProducer<>(properties);
```

- Por último, enviamos directamente el clima al servicio sin necesidad de serializarlo.

```
ProducerRecord<String, Clima> record = new ProducerRecord<>(TOPIC, CITY, clima);
```

- La clase quedaría de la siguiente manera:

```

package com.curso.kafka.productorclimaschemaregistry;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import com.curso.kafka.avro.model.Clima;

import io.confluent.kafka.serializers.KafkaAvroSerializer;
import io.confluent.kafka.serializers.KafkaAvroSerializerConfig;

public class ProductorSchemaRegistry {

    public static final String CITY = "madrid";
    public static final String TOPIC = "avro-clima-schema-registry";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());
        properties.put(KafkaAvroSerializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");

        KafkaProducer<String,Clima> producer = new KafkaProducer<>(properties);

        Thread shutdownHook = new Thread(producer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);

        while(true) {
            Clima clima = OpenWeatherMap.getWeatherFromOpenWeatherMap(CITY);
            ProducerRecord<String, Clima> record = new ProducerRecord<>(TOPIC, CITY, clima);
            producer.send(record);
            Thread.sleep(1500);
        }
    }
}

```

- Hemos eliminado los bytes y el método serializador.

13.4.3. Generación del consumidor

- Realizamos la misma operación al consumidor.
- Para ello, copiamos el proyecto y lo renombramos a:
 - consumidorClimaSchemaRegistry
- Cambiamos el pom.xml para modificar los identificadores:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.curso.kafka.avro</groupId>
<artifactId>ConsumidorClimaSchemaRegistry</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

- Agregamos como en el ejemplo anterior el serializador y el repositorio de confluent

```
<repositories>
  <repository>
    <id>confluent</id>
    <url>http://packages.confluent.io/maven/</url>
  </repository>
</repositories>
<dependency>
  <groupId>io.confluent</groupId>
  <artifactId>kafka-avro-serializer</artifactId>
  <version>6.0.0</version>
</dependency>
```

- El fichero pom quedará de la siguiente manera

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.curso.kafka.avro</groupId>
    <artifactId>ConsumidorClimaSchemaRegistry</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.0</version>
                <configuration>
                    <source>11</source>
                    <target>11</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <repositories>
        <repository>
            <id>confluent</id>
            <url>http://packages.confluent.io/maven/</url>
        </repository>
    </repositories>
    <dependencies>
        <dependency>
            <groupId>com.kafka</groupId>
            <artifactId>avro-tools</artifactId>
            <version>1.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>2.6.0</version>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
            <version>1.7.5</version>
        </dependency>
        <dependency>
            <groupId>io.confluent</groupId>
            <artifactId>kafka-avro-serializer</artifactId>
            <version>6.0.0</version>
        </dependency>
    </dependencies>
</project>

```

- En cuanto al consumidor, hay que modificar el nombre de la clase y el paquete

```
package com.curso.kafka.consumidorclimaschemaregistry;
```

```
...
```

```
public class ConsumidorSchemaRegistry {
```

- Cambiamos el topic para que coincida con el del productor

```
public static final String TOPIC = "avro-clima-schema-registry";
```

- Cambiamos el deserializador para que use el de Avro

```
properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.class.getName());
```

- Configuramos el Schema Registry

```
properties.put(KafkaAvroDeserializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");
```

- Cambiamos el client-id para que sea distinto y el consumidor pertenezca a un grupo distinto

```
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "consumidorClimaSchemaRegistry");
```

- Creamos el consumidor con tipo Clima

```
KafkaConsumer<String, Clima> consumer = new KafkaConsumer<>(properties);
```

- Configuramos los ConsumerRecords para que sean de tipo <String, Clima>

```
ConsumerRecords<String, Clima> records = consumer.poll(Duration.ofMillis(100));
```

- Por último, recogemos cada record y consideramos que el value ya está deserializado

```
for(ConsumerRecord<String, Clima> record : records) {  
    System.out.println("ID: " + record.key() + " - value: " + record.value().toString());  
}
```

- La clase quedaría como sigue

```

package com.curso.kafka.consumidorclimaschemaregistry;

import java.io.IOException;
import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import com.curso.kafka.avro.model.Clima;

import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializerConfig;

public class ConsumidorSchemaRegistry {

    public static final String TOPIC = "avro-clima-schema-registry";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.class.getName());
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "consumidorClimaSchemaRegistry");
        properties.put(KafkaAvroDeserializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");

        KafkaConsumer<String, Clima> consumer = new KafkaConsumer<>(properties);

        Thread shutdownHook = new Thread(consumer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);

        consumer.subscribe(Collections.singletonList(TOPIC));
        while(true) {
            ConsumerRecords<String, Clima> records = consumer.poll(Duration.ofMillis(100));
            for(ConsumerRecord<String, Clima> record : records) {
                System.out.println("ID: " + record.key() + " - value: " + record.value().toString());
            }
        }
    }
}

```

13.4.4. Prueba de ejecución

- Antes de ejecutar, vamos a comprobar que no existen **Subjects** dentro del Schema Registry.
- Cada subject indica el esquema almacenado dentro de un topic (generalmente)

```
[kafka@kafka-server avro-maven]$ curl http://localhost:8081/subjects
[]
```

- El array vacío indica que no existen esquemas almacenados.

- Ahora, lo primero ejecutamos el productor para comenzar a enviar mensajes ejecutando el productor.
- En cuanto al consumidor, ejecutamos la aplicación java como en laboratorios anteriores, sin embargo, el consumidor no ha sido capaz de deserializar el objeto

```
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.consumer.ConsumerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
Exception in thread "main" java.lang.ClassCastException: class org.apache.avro.generic.GenericData$Record cannot be cast
to class com.curso.kafka.avro.model.Clima (org.apache.avro.generic.GenericData$Record and
com.curso.kafka.avro.model.Clima are in unnamed module of loader 'app')
    at com.curso.kafka.consumidorclimaschemaregistry.ConsumidorSchemaRegistry.main(ConsumidorSchemaRegistry.java:44)
```

- En este caso, podríamos usar lo que se llama un Generic Record, que es una interfaz simple para obtener valores.
- Sería de la siguiente manera

```
KafkaConsumer<String, GenericRecord> consumer = new KafkaConsumer<>(properties);
```

- Pero si lo que queremos es que lo lea con un lector avro específico, que es el Clima, hay que indicarlo en la configuración del consumidor

```
properties.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, true);
```

- En el momento en que lo ejecutamos ya funciona correctamente.

13.4.5. Comprobación del Schema Registry

- En el Schema Registry se ha almacenado el esquema de serialización.
- Podemos consultar su existencia con:

```
[kafka@kafka-server avro-maven]$ curl http://localhost:8081/subjects
[{"avro-clima-schema-value"}]
```

13.5. Estrategias de nombres.

- Cuando un productor almacena un nuevo esquema, por defecto utiliza la estrategia de indicar en que campo disponible del topic se puede almacenar.
- Entre ellos está:
 - key
 - value
- En el caso de que se almacene en el value, el nombre sería:
 - <topic>-value
- Existen tres tipos:
 - **Topic Name Strategy**
 - <topic_name>-key
 - <topic_name>-value
 - **Record Name Strategy**
 - Se utiliza el nombre completo del esquema (Fully Qualified Schema Name), namespace+name
 - **Topic Record Name Strategy**
 - Es una mezcla de los dos, donde se usa el <topic_name> seguido del nombre completo del esquema
- Por defecto se usa el Topic Name Strategy
- Sin embargo, para casos de un solo topic, podemos usar también el **Record Name Strategy**
- En el caso de que tengamos distintos tipos de records, podemos usar Record Name Strategy o Topic Record Name Stategy, la cual es más compleja de usar que las demás.
- En este último caso, no se podría usar el Topic Name Strategy

13.6. Lab: Estrategias de Schema Registry

- En este caso, vamos a copiar de nuevo los ejemplos para generar la versión V2, el esquema de tipo topic-strategy, donde la clave tiene un esquema y el valor tiene otro esquema

13.6.1. Topic strategy

- Vamos a completarlo creando un nuevo objeto Ciudad que se guardará en la clave.

13.6.1.1. Generación de nuevo esquema

- Para ello, en nuestro proyecto avro-tools agregamos un nuevo esquema llamado Ciudad.avsc

Contenido de /src/main/avro/Ciudad.avsc

```
{  
    "name": "Ciudad",  
    "namespace": "com.curso.kafka.avro.model",  
    "type": "record",  
    "fields": [  
        {  
            "name": "ciudad",  
            "type": "string"  
        }  
    ]  
}
```

- Una vez que lo tenemos, vamos a generar el modelo, construyendo por medio de run as → maven → generate-sources
- Una vez hecho ya estará disponible para todos los demás proyectos.

13.6.1.2. Generación del productor

- Vamos a adaptar el ejercicio anterior para que use el Schema Registry en la clave también
- Primero copiamos el proyecto productor anterior y lo pegamos con un nuevo nombre llamado:
 - ProductorClimaSchemaRegistryV2
- También vamos a modificar el pom.xml para que responda a un nuevo paquete:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"  
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
    <modelVersion>4.0.0</modelVersion>  
    <groupId>com.curso.kafka.avro</groupId>  
    <artifactId>ProductorClimaSchemaRegistryV2</artifactId>  
    <version>0.0.1-SNAPSHOT</version>
```

- Cambiamos el topic para que apunte a uno nuevo

```
public static final String TOPIC = "avro-clima-schema-registry-v2";
```

- Indicamos que la clave ahora ya no es un StringSerializer sino avro

```
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());
```

- Cambiamos la clave del productor a Clima

```
KafkaProducer<Ciudad,Clima> producer = new KafkaProducer<>(properties);
```

- Construimos el objeto Ciudad para que lo podamos introducir

```
Ciudad ciudad = Ciudad.newBuilder()
    .setCiudad(CITY)
    .build();
```

- Cambiamos el record para que agreguemos la ciudad

```
ProducerRecord<Ciudad, Clima> record = new ProducerRecord<>(TOPIC, ciudad, clima);
```

13.6.1.3. Generación de consumidor

- Realizamos la misma operación al consumidor.
- Para ello, copiamos el proyecto y lo renombramos a:
 - consumidorClimaSchemaRegistryV2
- Cambiamos el pom.xml para modificar los identificadores:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.curso.kafka.avro</groupId>
  <artifactId>ConsumidorClimaSchemaRegistryV2</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

- Renombramos el paquete y el nombre de la clase

```
package com.curso.kafka.consumidorclimaschemaregistryv2;
...
public class ConsumidorSchemaRegistryV2 {
```

- Cambiamos el topic para que apunte al nuevo Topic

```
public static final String TOPIC = "avro-clima-schema-registry-v2";
```

- Cambiamos el deserializador de la clave

```
properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.class.getName());
```

- Cambiamos el groupId

```
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "consumidorClimaSchemaRegistryV2");
```

- Cambiamos la declaración del consumidor para que devuelva una Ciudad como clave, en vez de un String

```
KafkaConsumer<Ciudad, Clima> consumer = new KafkaConsumer<>(properties);
```

- En el poll, devolvemos ya la clave de tipo ciudad

```
ConsumerRecords<Ciudad, Clima> records = consumer.poll(Duration.ofMillis(100));
```

- Por último, como el record devuelve como key una Ciudad, indicamos el toString para que lo devuelva la clase Avro.

```
for(ConsumerRecord<Ciudad, Clima> record : records) {  
    System.out.println("ID: " + record.key().toString() + " - value: " + record.value().toString());  
}
```

13.6.1.4. Prueba de ejecución

- Para ello, ejecutamos el productor y el consumidor con Run as → Java Application
- Comprobamos como sigue funcionando correctamente la serialización/deserialización
- Si ahora nos fijamos en el Schema Registry

```
[kafka@kafka-server avro-maven]$ curl http://localhost:8081/subjects  
"avro-clima-schema-registry-value","avro-clima-schema-registry-v2-value","avro-clima-schema-registry-v2-key"]
```

- Podemos observar que tenemos el key y el value de nuestro topic como dos subjects distintos.

13.6.2. Record Name Strategy

- En este caso vamos a almacenar en valor dos esquemas distintos, Datos y Clima, y el consumidor podrá obtener la información de los dos.

- Para ello, vamos a crear dos productores y un único consumidor

13.6.2.1. Generación de productores

- Para el segundo caso, vamos a tener dos productores.
- Copiamos el proyecto productor y lo llamamos
 - ProductorClimaSchemaRegistryRecordNameStrategy
- Luego, la clase productor la renombramos a
 - ProductorClimaSchemaRegistryRecordNameStrategyClima
- Copiamos la clase productor y la llamamos
 - ProductorClimaSchemaRegistryRecordNameStrategyDatos
- El resultado debe ser el siguiente:



- Cambiamos el pom.xml para renombrar el artefacto

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.curso.kafka.avro</groupId>
  <artifactId>ProductorClimaSchemaRegistryRecordNameStrategy</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

- En este caso vamos usar el productor Clima, y cambiamos el topic para que inserte los datos en un nuevo topic.

```
public static final String TOPIC = "avro-clima-schema-registry-record-name-strategy";
```

- Si ejecutamos el productor, vemos como inserta los datos sin problemas

```

log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.producer.ProducerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
{"coord":{"lon":-3.7,"lat":40.42},"weather":[{"id":802,"main":"Clouds","description":"scattered clouds","icon":"03d"}],"base":"stations","main":{"temp":288.58,"feels_like":284.95,"temp_min":287.59,"temp_max":289.26,"pressure":1017,"humidity":38},"visibility":10000,"wind":{"speed":2.6,"deg":350},"clouds":{"all":49},"dt":1602780001,"sys":{"type":1,"id":6443,"country":"ES","sunrise":1602743147,"sunset":1602783292},"timezone":7200,"id":3117735,"name":"Madrid","cod":200}
{"coord":{"lon":-3.7,"lat":40.42},"weather":[{"id":802,"main":"Clouds","description":"scattered clouds","icon":"03d"}],"base":"stations","main":{"temp":288.58,"feels_like":284.95,"temp_min":287.59,"temp_max":289.26,"pressure":1017,"humidity":38},"visibility":10000,"wind":{"speed":2.6,"deg":350},"clouds":{"all":49},"dt":1602780001,"sys":{"type":1,"id":6443,"country":"ES","sunrise":1602743147,"sunset":1602783292},"timezone":7200,"id":3117735,"name":"Madrid","cod":200}
{"coord":{"lon":-3.7,"lat":40.42},"weather":[{"id":802,"main":"Clouds","description":"scattered clouds","icon":"03d"}],"base":"stations","main":{"temp":288.58,"feels_like":284.95,"temp_min":287.59,"temp_max":289.26,"pressure":1017,"humidity":38},"visibility":10000,"wind":{"speed":2.6,"deg":350},"clouds":{"all":49},"dt":1602780001,"sys":{"type":1,"id":6443,"country":"ES","sunrise":1602743147,"sunset":1602783292},"timezone":7200,"id":3117735,"name":"Madrid","cod":200}

```

- Vamos al segundo productor, y cambiamos el tipo enviado

```
KafkaProducer<Ciudad,Datos> producer = new KafkaProducer<>(properties);
```

- Cambiamos el record a enviar para enviar los datos en vez del clima

```
ProducerRecord<Ciudad, Datos> record = new ProducerRecord<>(TOPIC, ciudad, clima.getDatos());
```

- Ahora ejecutamos el productor

```

Exception in thread "main" org.apache.kafka.common.errors.SerializationException: Error registering Avro schema:
{"type":"record","name":"Datos","namespace":"com.curso.kafka.avro.model","fields":[{"name":"temp","type":"float"}, {"name":"presion","type":"int"}, {"name": "humedad", "type": "int"}, {"name": "tempMin", "type": "float"}, {"name": "tempMax", "type": "float"}]}
Caused by: io.confluent.kafka.schemaregistry.client.rest.exceptions.RestClientException: Schema being registered is incompatible with an earlier schema for subject "avro-clima-schema-registry-record-name-strategy-value"; error code: 409
    at io.confluent.kafka.schemaregistry.client.rest.RestService.sendHttpRequest(RestService.java:292)
    at io.confluent.kafka.schemaregistry.client.rest.RestService.httpRequest(RestService.java:352)
    at io.confluent.kafka.schemaregistry.client.rest.RestService.registerSchema(RestService.java:495)
    at io.confluent.kafka.schemaregistry.client.rest.RestService.registerSchema(RestService.java:486)
    at io.confluent.kafka.schemaregistry.client.rest.RestService.registerSchema(RestService.java:459)
    at
io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient.registerAndGetId(CachedSchemaRegistryClient.java:206)
    at io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient.register(CachedSchemaRegistryClient.java:268)
    at io.confluent.kafka.schemaregistry.client.CachedSchemaRegistryClient.register(CachedSchemaRegistryClient.java:244)
    at io.confluent.kafka.serializers.AbstractKafkaAvroSerializer.serializeImpl(AbstractKafkaAvroSerializer.java:75)
    at io.confluent.kafka.serializers.KafkaAvroSerializer.serialize(KafkaAvroSerializer.java:59)
    at org.apache.kafka.common.serialization.Serializer.serialize(Serializer.java:62)
    at org.apache.kafka.clients.producer.KafkaProducer.doSend(KafkaProducer.java:910)
    at org.apache.kafka.clients.producer.KafkaProducer.send(KafkaProducer.java:870)
    at org.apache.kafka.clients.producer.KafkaProducer.send(KafkaProducer.java:758)
    at
com.curso.kafka.productorclimaschemaregistryrecordnamestrategy.ProductorSchemaRegistryRecordNameStrategy.main(ProductorSchemaRegistryRecordNameStrategy.java:41)

```

- Vaya! por lo que parece, el schema registry nos impide introducir objetos que no estén serializados en el schema registry en la entrada correspondiente!

- Comprobamos que entrada nos ha introducido dentro del Schema registry

- Para que los documentos se vean en formato pretty, debemos instalar el comando jq



```
[kafka@kafka-server avro-maven]$ sudo dnf -y install jq
```

```
[kafka@kafka-server avro-maven]$ curl http://localhost:8081/subjects | jq
% Total    % Received % Xferd  Average Speed   Time     Time      Current
                                         Dload  Upload Total Spent   Left  Speed
100  256  100  256    0     0  23272       0 --::-- --::-- --::-- 23272
[
  "avro-clima-schema-registry-value",
  "avro-clima-schema-registry-record-name-strategy-key",
  "avro-clima-schema-registry-record-name-strategy-value",
  "avro-clima-schema-registry-v2-value",
  "avro-clima-schema-registry-v2-key"
]
```

- Parece que sigue usando el criterio de key-value.
- Vamos a cambiarlo.
- Para ello, cambiamos en el productor la estrategia.

```
properties.put(KafkaAvroSerializerConfig.VALUE_SUBJECT_NAME_STRATEGY, RecordNameStrategy.class.getName());
```

- Y listo!
- Volvemos a ejecutar el productor de datos y ahora no da error!!
- Si volvemos a buscar los subjects vemos que aparece como el namespace+name del esquema.

```
[kafka@kafka-server avro-maven]$ curl http://localhost:8081/subjects | jq
% Total    % Received % Xferd  Average Speed   Time     Time      Current
                                         Dload  Upload Total Spent   Left  Speed
100  291  100  291    0     0  48500       0 --::-- --::-- --::-- 48500
[
  "com.curso.kafka.avro.model.Datos",
  "avro-clima-schema-registry-value",
  "avro-clima-schema-registry-record-name-strategy-key",
  "avro-clima-schema-registry-record-name-strategy-value",
  "avro-clima-schema-registry-v2-value",
  "avro-clima-schema-registry-v2-key"
]
```

- Vamos a configurar el productor original para que su esquema sea de la misma estrategia.

- Para ello, agregamos la estrategia

```
properties.put(KafkaAvroSerializerConfig.VALUE_SUBJECT_NAME_STRATEGY, RecordNameStrategy.class.getName());
```

- Si ejecutamos los dos, ahora ya pueden guardar los dos de forma independiente.
- Si consultamos los subjects de nuevo:

```
[kafka@kafka-server avro-maven]$ curl http://localhost:8081/subjects | jq
% Total    % Received % Xferd  Average Speed   Time     Time     Time  Current
          Dload  Upload Total Spent   Left Speed
100    326  100    326     0      0  32600       0 ---:---:---:---:---:--- 32600
[
  "com.curso.kafka.avro.model.Datos",
  "avro-clima-schema-registry-value",
  "avro-clima-schema-registry-record-name-strategy-key",
  "avro-clima-schema-registry-record-name-strategy-value",
  "avro-clima-schema-registry-v2-value",
  "avro-clima-schema-registry-v2-key",
  "com.curso.kafka.avro.model.Clima"
]
```

- Ya tenemos tanto datos como clima en el value.

13.6.2.2. Configuración del consumidor

- Está claro que ahora no podemos leer mas que algunos de los mensajes que nos llegan, ya que no puede transformar aquellos que no son de tipo Clima.
- Pero podemos generalizarlo para aceptar cualquiera.
- Para ello usamos el GenericRecord
- Primero copiamos el proyecto de consumidor anterior y lo llamamos
 - ConsumidorClimaSchemaRegistryRecordNameStrategy
- Modificamos el paquete y la clase para que se llamen:

```
package com.curso.kafka.consumidorclimaschemaregistryrecordnamestrategy;
...
public class ConsumidorSchemaRegistryRecordNameStrategy {
```

- Modificamos el pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.curso.kafka.avro</groupId>
<artifactId>ConsumidorClimaSchemaRegistryRecordNameStrategy</artifactId>
<version>0.0.1-SNAPSHOT</version>
```

- En la clase, apuntamos al topic

```
public static final String TOPIC = "avro-clima-schema-registry-record-name-strategy";
```

- Luego cambiamos el groupId

```
properties.put(ConsumerConfig.GROUP_ID_CONFIG, "consumidorClimaSchemaRegistryRecordNameStrategy");
```

- Luego cambiamos también la estrategia

```
properties.put(KafkaAvroDeserializerConfig.VALUE_SUBJECT_NAME_STRATEGY, RecordNameStrategy.class.getName());
```

- Cambiamos el consumer

```
KafkaConsumer<Ciudad, GenericRecord> consumer = new KafkaConsumer<>(properties);
```

- Por último, cambiamos los registros para que obtengan el valor de GenericRecord y devolvemos el toString() asociado para cada objeto que llegue

```
while(true) {
    ConsumerRecords<Ciudad, GenericRecord> records = consumer.poll(Duration.ofMillis(100));
    for(ConsumerRecord<Ciudad, GenericRecord> record : records) {
        System.out.println("ID: " + record.key().toString() + " - value: " + record.value().toString());
    }
}
```

- Ahora ejecutamos el cliente y vemos como puede consumir cualquier objeto avro.

```
log4j:WARN No appenders could be found for logger (org.apache.kafka.clients.consumer.ConsumerConfig).
log4j:WARN Please initialize the log4j system properly.
log4j:WARN See http://logging.apache.org/log4j/1.2/faq.html#noconfig for more info.
ID: {"ciudad": "madrid"} - value: {"id": 3117735, "nombre": "Madrid", "datos": {"temp": 288.0, "presion": 1017, "humedad": 37, "tempMin": 287.0, "tempMax": 290.0}, "detalles": [{"id": 802, "principal": "Clouds", "descripcion": "scattered clouds", "icono": "03d"}]}
ID: {"ciudad": "madrid"} - value: {"temp": 288.0, "presion": 1017, "humedad": 37, "tempMin": 287.0, "tempMax": 290.0}
ID: {"ciudad": "madrid"} - value: {"id": 3117735, "nombre": "Madrid", "datos": {"temp": 288.0, "presion": 1017, "humedad": 37, "tempMin": 287.0, "tempMax": 290.0}, "detalles": [{"id": 802, "principal": "Clouds", "descripcion": "scattered clouds", "icono": "03d"}]}
ID: {"ciudad": "madrid"} - value: {"temp": 288.0, "presion": 1017, "humedad": 37, "tempMin": 287.0, "tempMax": 290.0}
ID: {"ciudad": "madrid"} - value: {"id": 3117735, "nombre": "Madrid", "datos": {"temp": 288.0, "presion": 1017, "humedad": 37, "tempMin": 287.0, "tempMax": 290.0}, "detalles": [{"id": 802, "principal": "Clouds", "descripcion": "scattered clouds", "icono": "03d"}]}
ID: {"ciudad": "madrid"} - value: {"temp": 288.0, "presion": 1017, "humedad": 37, "tempMin": 287.0, "tempMax": 290.0}
ID: {"ciudad": "madrid"} - value: {"id": 3117735, "nombre": "Madrid", "datos": {"temp": 288.0, "presion": 1017, "humedad": 37, "tempMin": 287.0, "tempMax": 290.0}, "detalles": [{"id": 802, "principal": "Clouds", "descripcion": "scattered clouds", "icono": "03d"}]}
ID: {"ciudad": "madrid"} - value: {"temp": 288.0, "presion": 1017, "humedad": 37, "tempMin": 287.0, "tempMax": 290.0}
ID: {"ciudad": "madrid"} - value: {"id": 3117735, "nombre": "Madrid", "datos": {"temp": 288.0, "presion": 1017, "humedad": 37, "tempMin": 287.0, "tempMax": 290.0}, "detalles": [{"id": 802, "principal": "Clouds", "descripcion": "scattered clouds", "icono": "03d"}]}
```

- Ya tenemos un consumidor con RecordNameStrategy

Capítulo 14. Log Compaction

Log Compaction es una propiedad que asegura que se almacena el último valor para una clave dada.

Esto, es configurable a nivel de **Topic** (puedes tener **topics** que lo usen, y otros que no).

Es de especial utilidad cuando hay que recuperar un estado tras un fallo, o restaurar una caché que se ha ido creando durante la ejecución de un proceso.

En sistemas como **Kafka Stream** o **Apache Samza** es de bastante utilidad.

En la siguiente imagen vemos el aspecto que tiene un **topic**, que recibe mensajes (clave,valor), y les asigna un **offset**

offset	0	1	2	3	4	5	6
clave	k1	k2	k3	k3	k2	k4	k1
valor	2	3	5	7	9	7	9

El **Log Compaction** identifica los mensajes que tienen una misma clave

offset	0	1	2	3	4	5	6
clave	k1	k2	k3	k3	k2	k4	k1
valor	2	3	5	7	9	7	9

Después, se queda con un único mensaje por clave, siendo este el que mayor **offset** posee

offset	0	1	2	3	4	5	6
clave	k1	k2	k3	k3	k2	k4	k1
valor	2	3	5	7	9	7	9

Dejando el log con las últimas versiones de cada clave

offset	3	4	5	6
clave	k3	k2	k4	k1
valor	7	9	7	9

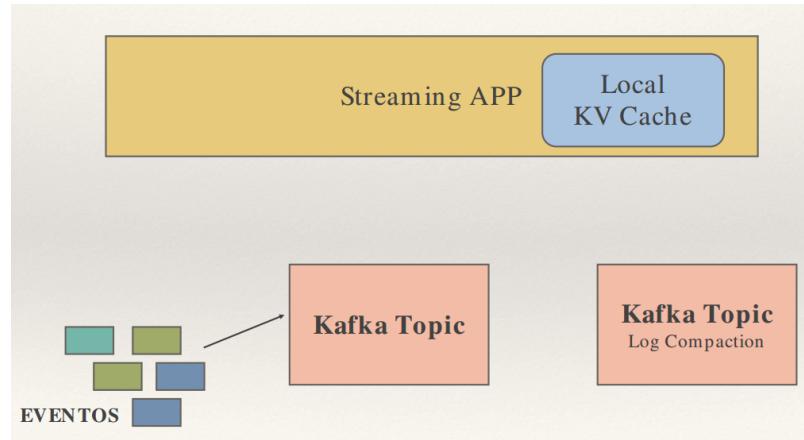
Un caso de uso de esto sería una aplicación de streaming que guarda una caché de los últimos valores recibidos.

Para ello, va leyendo de un **topic** la información, y actualizando su caché.

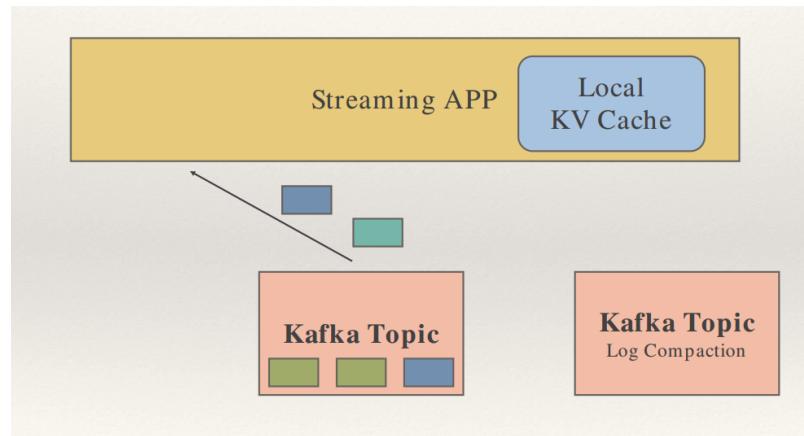
Al mismo tiempo, guarda en un **topic** propio con **log compaction** los datos que lee.

Si el sistema se cae, vuelve a levantarse leyendo todos los datos del **topic** en cuestión, regenerando así su caché de forma automática

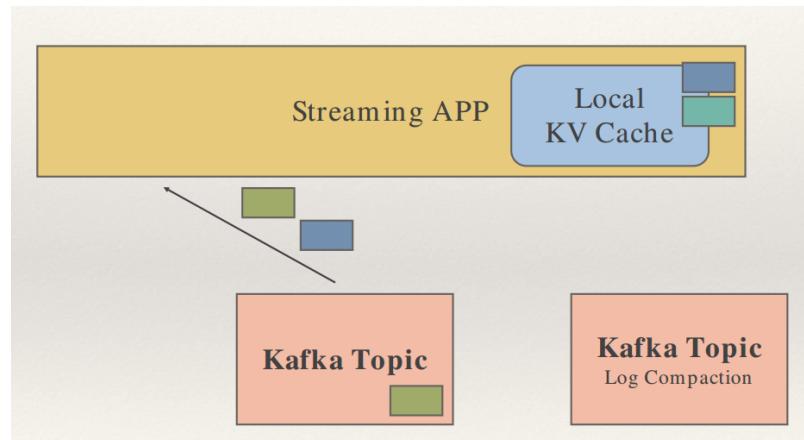
Los productores escriben en un topic



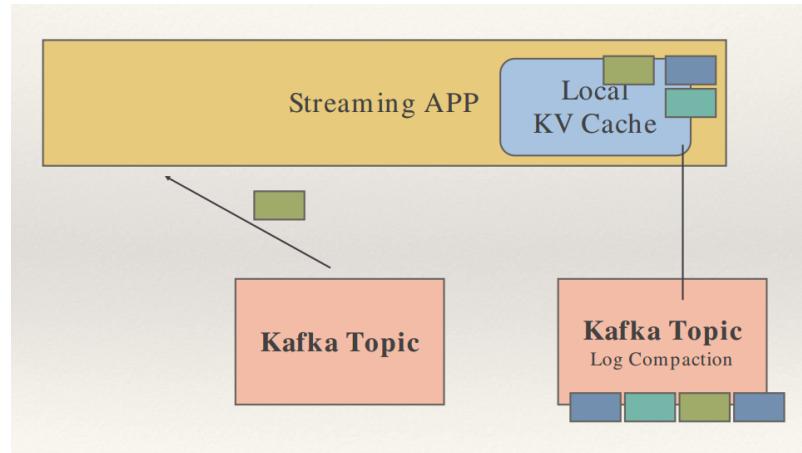
Nuestra aplicación, consume los mensajes de dicho topic



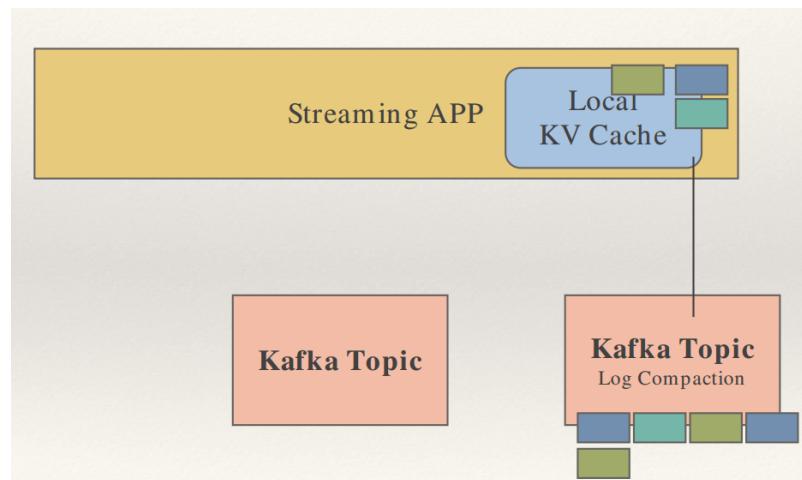
Al consumir los mensajes, va creando una caché (por clave)



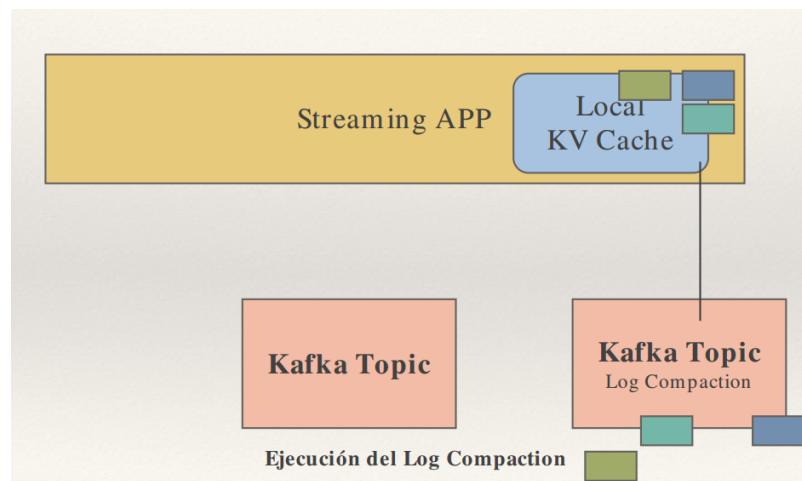
Todos los mensajes que lee, los guarda en un topic con log compaction. Internamente, actualiza los valores de su caché si una clave ya había sido leída



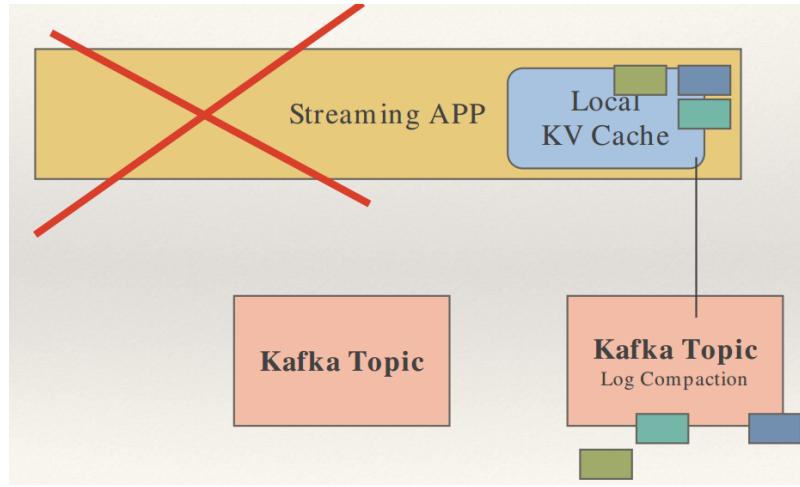
La caché tendrá un subconjunto de datos, pero el topic con Log Compaction habrá recibido el total de los mensajes leídos



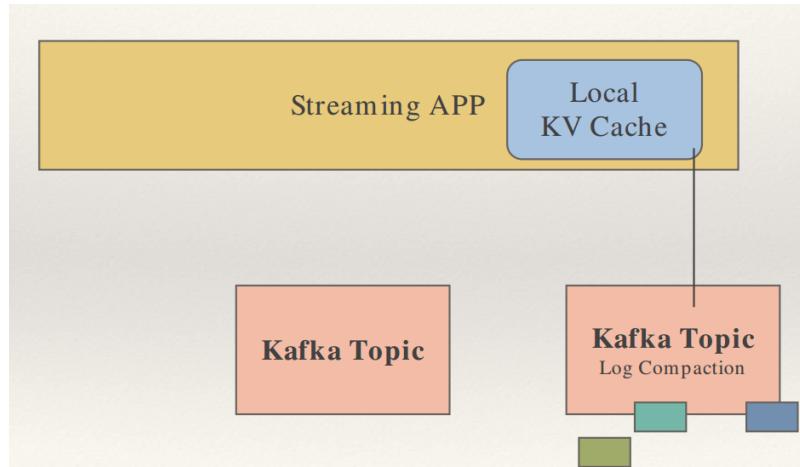
Al hacer un Log Compaction, el topic se queda con un único mensaje por cada clave



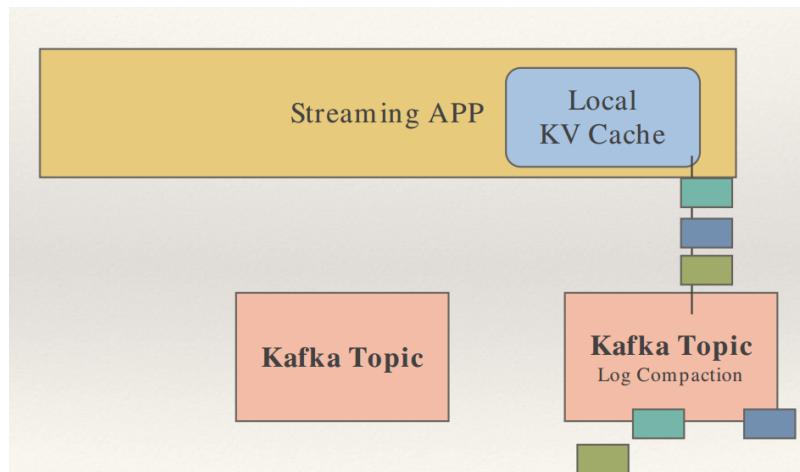
Nuestra aplicación se cae



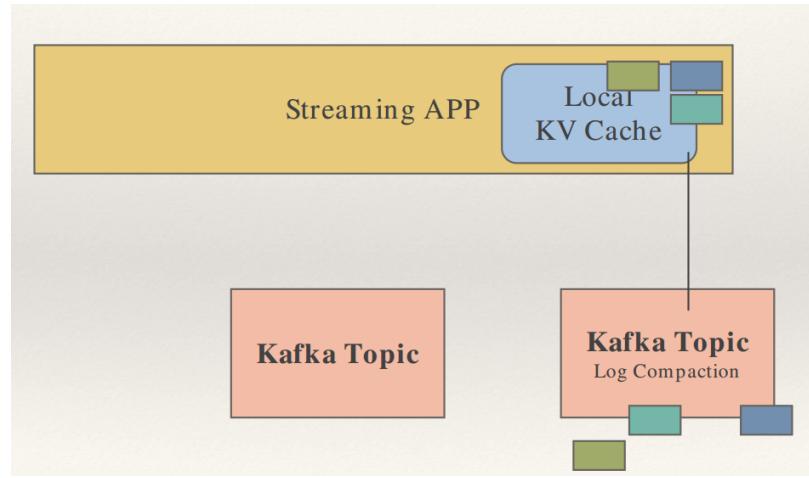
Al levantarse, se conecta a su topic como consumidor



El topic, le proporciona todos los mensajes, que contienen sólo la última versión de cada clave



Con estos mensajes, la caché se ha regenerado, y la aplicación puede volver a trabajar con normalidad.



Capítulo 15. Kafka Streams API

- Permite transformar y enriquecer los datos
 - Soporte de procesado de streams con latencias de milisegundos sin estado.
 - Soporte de procesado de streams por ventana con estado.
- Posee tolerancia a fallos y soporte de procesamiento distribuido
- Posee su propio DSL
 - Con operaciones comunes como map, flatMap, count, etc.
- Existen implementaciones en diversos lenguajes.

15.1. Características

- Posee capacidades de streaming
- No requiere su propio cluster, se trata de una librería
- Puede ejecutarse en una o múltiples máquinas
- Se trata de una implementación concreta de Productor/Consumidor

15.2. Streams

- Un stream es un flujo de registros continuo
 - No pedimos registros, sino que nos llegan
- Los registros son de tipo clave → valor
- Un procesador de streams transforma los datos en streams

15.3. KStreams y KTables

15.3.1. KStream

- Un KStream es una abstracción de un stream de record
- Se pueden interpretar como inserts continuos.
- Ningún record reemplaza el anterior.
- Puede ser útil para operaciones de tipo serverLog.
 - Cada record representa un trozo de datos autocontenido
 - Ejemplo, (1,1) (1,2) → Como KStream su resultado podría ser 3 para id 1, ya que se tienen en cuenta todos los datos.

15.3.2. KTable

- Un KTable es una abstracción de un stream changelog.

- Se puede interpretar como un Upsert. (Si no existe, es Insert, si existe es Update)
 - Cada record representa una actualización
 - Ejemplo, (1,1) (1,2) → Como KTable, se trata de una actualización del primero, luego para el id 1, su valor es el último, 2.
 - Para KTables es lógico activar el *Log Compaction* en el topic asociado, ya que solo importa el último valor por cada key.

15.3.3. GlobalKTable

- Representa una abstracción de un stream de changelog.
- Permite compartir la información del global KTable con hilos.
- Si usaramos KTable, cada instancia tendría su propio KTable.
- Con GlobalKTable garantizamos que la información es compartida por todos los hilos.
- Muy útil para uso de joins, ya que podemos buscar tanto por claves como por valores, y además no tiene porqué estar co-particionados.
- Incrementa el consumo de almacenamiento local comparado con el particionado.
- Incrementa el consumo de red y de los brokers de kafka, ya que lee el topic completo.

15.4. Ventanas (Windows)

- Hay que tener en cuenta que el API de streams de Kafka se basa en ventanas de tiempo.
- Los datos se dividen en buckets temporales
- Podemos realizar agregaciones en los records, como sum o count.
- Podemos realizar join, merge para distintos sources.
- Existen distintos tipos de windows:
 - Tumbling: Es de tamaño fijo, no se solapan las ventanas de tiempo, no genera espacios entre ventanas.

```
TimeWindows.of(TimeUnit.MINUTES.toMillis(5)).advanceBy(TimeUnit.MINUTES.toMillis(5));
```

- Hopping: Tamaño fijo, Las ventanas se solapan

```
TimeWindows.of(TimeUnit.MINUTES.toMillis(5)).advanceBy(TimeUnit.MINUTES.toMillis(1));
```

- Sliding: Tamaño fijo, ventanas solapadas que trabaja con diferencias entre los timestamps de los records. Solo se usan en operaciones de tipo Join, por medio de la clase JoinWindows.
- Session: Tamaño dinámico. Sin solapamiento, ventana gestionada por datos.

```
SessionWindows.with(TimeUnit.MINUTES.toMillis(5));
```

15.5. Transformaciones

- Los datos pueden transformarse usando distintos operadores
- Algunos operadores devuelven un objeto KStream, como filter o map
- Otros devuelven KTables, como agregaciones

15.5.1. Transformaciones sin estado

- **branch:** Permite dividir un KStream en varios KStreams según un predicado definido. Solo para KStreams

```
KStream<String, Long>[] branches = stream.branch(  
    (key, value) -> key.equals("MADRID"), // KStream con los datos de clave madrid  
    (key, value) -> key.equals("BURGOS"), // KStream con los datos de burgos  
    (key, value) -> true // KStream con todo lo demás  
>);
```

- **filter:** Creación de un KStream con records que cumplan criterios concretos

```
KStream<String, Long> masDe30 = stream.filter((key, value) -> value.getDatos().getTemp() > 30);
```

- Todos los datos con temperatura mayor de 30°
- **inverseFilter:** Función booleana que rechaza todos los datos que devuelvan true

```
KStream<String, Long> menos0IgualA30 = stream.filterNot((key, value) -> value.getDatos().getTemp() > 30);
```

- **flatMap:** Creación de un KStream transformando cada elemento en 0, 1 o más elementos en el nuevo stream. Se permite la modificación de las claves y valores incluidos sus tipos de datos. Solo para KStreams

```
KStream<String, Integer> transformed = stream.flatMap(  
    (key, value) -> {  
        List<KeyValue<String, Integer>> result = new LinkedList<>();  
        result.add(KeyValue.pair(value.getDatos().getName()+"Max", value.getDatos().getTempMax()));  
        result.add(KeyValue.pair(value.getDatos().getName()+"Min", value.getDatos().getTempMin()));  
        return result;  
    }  
>);
```

- **flatMapValues:** Creación de un KStream transformando cada valor de cada elemento en 0 o 1 elemento distintos en el stream nuevo. Solo modifica el valor, manteniendo la clave. Solo para KStreams

```
datos.flatMapValues(value -> value.getDatos());
```

- **Foreach:** Permite una operación sin estado en cada record. Es una operación final, es decir, no devuelve un kstream o un ktable.

```
stream.foreach((key, value) -> System.out.println(key + " => temp: " + value.getDatos().getTemp()));
```

- **GroupByKey:** Agrupa records por claves (Para KStreams y KGroupedStreams)

- Es un prerequisito para agregaciones. Permite reparticionar si se marca explicitamente para ello.

```
KGroupedStream<byte[], String> streamGrouped = stream.groupByKey();  
KGroupedStream<byte[], String> groupedStream = stream.groupByKey(  
    Serialized.with(  
        Serdes.String(),  
        Serdes.Float())  
);
```

- **GroupBy:** Permite agrupar records por una nueva clave que puede ser de un tipo distinto.

- Al agrupar una tabla, se debe especificar el valor y el tipo
- Equivalente a selectKey().groupByKey()

```
KGroupedStream<String, String> groupedStream = stream.groupBy(  
    (key, value) -> value,  
    Serialized.with(  
        Serdes.String(),  
        Serdes.Float())  
);  
  
KGroupedTable<String, Float> groupedTable = table.groupBy(  
    (key, value) -> KeyValue.pair(value, value.getDatos().getTemp()),  
    Serialized.with(  
        Serdes.String(),  
        Serdes.Float())  
);
```

- **map:** Creación de un KStream transformando cada elemento en otro distinto en el nuevo stream

```
KStream<String, Long> transformed = stream.map(  
    (key, value) -> KeyValue.pair(value.getName().toUpperCase(), value.getDatos().getTemp()));
```

- **mapValues:** Creación de un KStream transformando el valor de cada elemento en otro

elemento distinto, pero solo modifica el valor

```
stream.mapValues(value → value.getDatos());
```

- **Peek:** Realiza una acción sin estado por cada record y devuelve un stream intacto. Similar a forEachm pero no finaliza.

```
KStream<byte[], String> noModificado = stream.peek((key, value) -> System.out.println(key + " => temp: " + value.getDatos().getTemp()));
```

- **Print:** Operación terminal. Permite mostrar los records por la salida estándar. Útil para desarrollo.

```
stream.print(Printed.toFile("salida.txt").withLabel("a-topic-nuevo"));
```

- **SelectKey:** Asigna una nueva key

```
stream.selectKey((key, value) -> value.getName())
```

Ejemplo de Stateless Processing

```
public class SimpleStreamsExample {

    public static void main(String[] args) throwsException {
        Properties streamsConfiguration = new Properties();
        // Aplicación con nombre único obligatorio para el cluster de Kafka
        streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "ejemplo-sencillo");
        streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "broker1:9092");
        streamsConfiguration.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        // Serializadores y deserializadores
        streamsConfiguration.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.ByteArray().getClass());
        streamsConfiguration.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
        // Transformación de datos y ejecución
        StreamsBuilder builder = new StreamsBuilder();

        // KStream desde el topic mi-topic-a-stream
        KStream<byte[], String> textLines = builder.stream("mi-topic-a-stream");

        // Uppercase del KStream
        KStream<byte[], String> uppercasedWithMapValues = textLines.mapValues(String::toUpperCase);

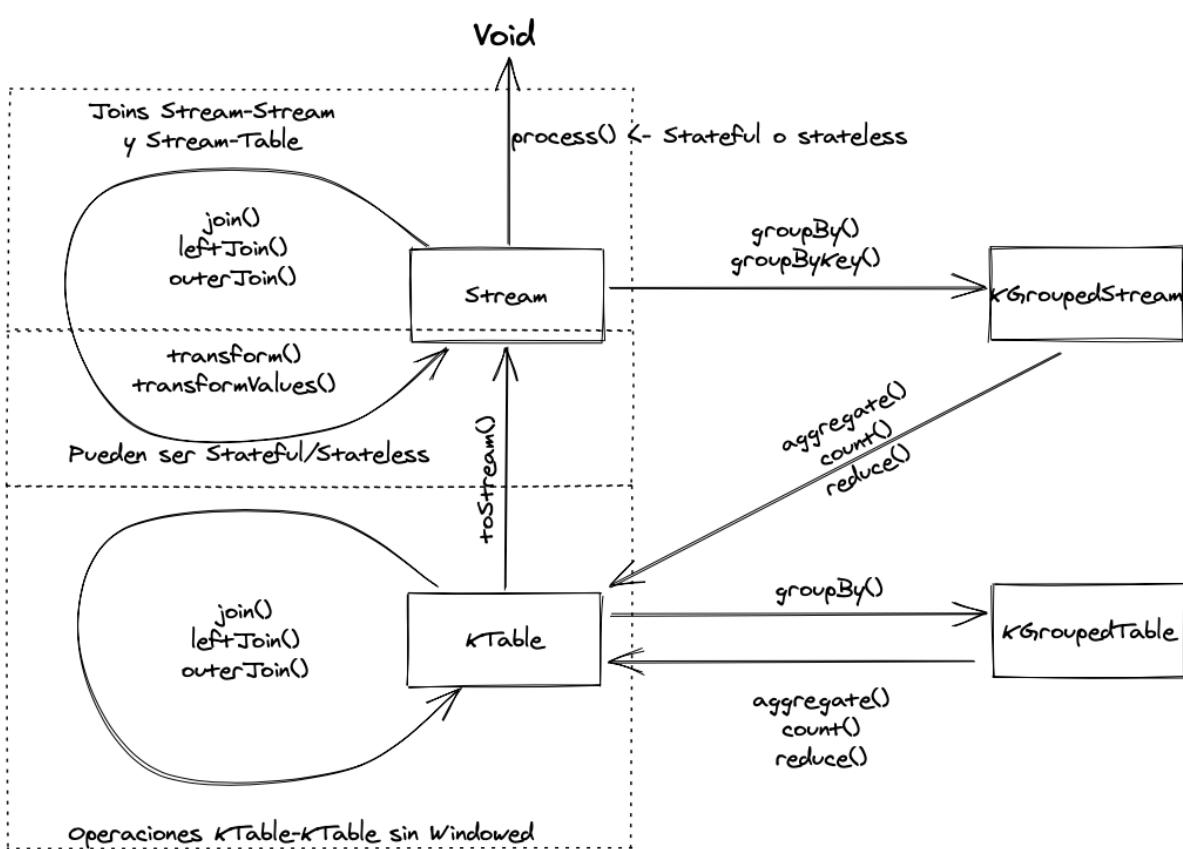
        // envio a un nuevo topic "mi-stream-a-topic"
        uppercasedWithMapValues.to("mi-stream-a-topic");

        // Inicio de la aplicación
        KafkaStreams streams = new KafkaStreams(builder.build(), streamsConfiguration);
        streams.start();

        //Apagado suave
        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
    }
}
```

15.5.2. Transformaciones con estado

- Las transformaciones con estado dependen del estado para procesar entradas y producir salidas.
- Debe poseer un state store asociado al procesador de stream.
- Estos almacenes permiten ser tolerantes a fallos
- Las transformaciones disponibles son:
 - agregaciones
 - joins
 - windows
 - transformaciones personalizadas
- Para saber como se pueden utilizar cada uno de los objetos, podemos ver el siguiente diagrama



- **aggregate:** Permite agregar los valores de records por medio de la clave agrupada. Es una generalización del Reduce y permite tener diferentes tipos a los valores de entrada.
 - Se usa con KGroupedStream y KGroupedTable devolviendo un KTable

```

KTable<byte[], Long> streamAgregado = groupedStream.aggregate(
    () -> 0L,
    (aggKey, newValue, aggValue) -> aggValue + newValue.length(),
    Materialized.as("aggregated-stream-store")
        .withValueSerde(Serdes.Long()));

KTable<byte[], Long> TablaAgregada = groupedTable.aggregate(
    () -> 0L,
    (aggKey, newValue, aggValue) -> aggValue + newValue.length(),
    (aggKey, oldValue, aggValue) -> aggValue - oldValue.length(),
    Materialized.as("aggregated-table-store")
        .withValueSerde(Serdes.Long()))

```

- Las claves nulas son ignoradas
- Al recibir el primer record, se inicializa el valor de agregación en el caso de KStreams, en KTables se ejecuta luego la función
- Al recibir un record con valor, se llama a la función lambda declarada.
 - **Agregación (Windowed)**
 - Agrega los valores de los records por window, por la clave de grupo.

```

KTable<Windowed<String>, Long> StreamWindowed = groupedStream.windowedBy(TimeUnit.MILLISECONDS.toMillis(5))
    .aggregate(
        () -> 0L,
        (aggKey, newValue, aggValue) -> aggValue + newValue,
        Materialized.<String, Long, WindowStore<Bytes, byte[]>>as("miStoreWindow")
            .withValueSerde(Serdes.Long()));

KTable<Windowed<String>, Long> sessionizedAggregatedStream = groupedStream.windowedBy(SessionWindows.with(TimeUnit
    .MINUTES.toMillis(5)))
    .aggregate(
        () -> 0L,
        (aggKey, newValue, aggValue) -> aggValue + newValue,
        (aggKey, leftAggValue, rightAggValue) -> leftAggValue + rightAggValue,
        Materialized.<String, Long, SessionStore<Bytes, byte[]>>as("MiStoreSession")
            .withValueSerde(Serdes.Long()));

```

- **Count:** Cuenta el número de records por clave agrupada. Necesita un KGroupedStream o un KGroupedTable

```

KTable<String, Long> aggregatedStream = groupedStream.count();

KTable<String, Long> aggregatedTable = groupedTable.count();

```

- **Count (Windowed):** Realiza la misma operación en una ventana de tiempo.

```

KTable<Windowed<String>, Long> aggregatedStream = groupedStream.windowedBy(
    TimeWindows.of(TimeUnit.MINUTES.toMillis(5)))
    .count();

KTable<Windowed<String>, Long> aggregatedStream = groupedStream.windowedBy(
    SessionWindows.with(TimeUnit.MINUTES.toMillis(5)))
    .count();

```

- **Reduce:** Combina los valores de los records por una clave agregada.

```

KTable<String, Long> aggregatedStream = groupedStream.reduce(
    (aggValue, newValue) -> aggValue + newValue);
KTable<String, Long> aggregatedTable = groupedTable.reduce(
    (aggValue, newValue) -> aggValue + newValue,
    (aggValue, oldValue) -> aggValue - oldValue);

```

- **Reduce (Windowed):** Combina los valores de los records por windows y por clave agrupada.

```

KTable<Windowed<String>, Long> timeWindowedAggregatedStream = groupedStream.
windowedBy(
    TimeWindows.of(TimeUnit.MINUTES.toMillis(5)))
    .reduce(
        (aggValue, newValue) -> aggValue + newValue
    );

KTable<Windowed<String>, Long> sessionizedAggregatedStream = groupedStream.windowedBy(
    SessionWindows.with(TimeUnit.MINUTES.toMillis(5)))
    .reduce(
        (aggValue, newValue) -> aggValue + newValue
    );

```

- Join:
- Los streams y las tablas pueden realizar las operaciones de Join.
- * Las operaciones que se pueden realizar son:
 - KStream-KStream
 - KStream-KTable
 - KStream-GlobalKTable
 - KTable-GlobalKTable
- Como condición, los joins deben estar co-particionados, es decir, que poseen la misma partición para poder hacer el join.

```

KeyValue<K, JV> joinOutputRecord = KeyValue.pair(
    leftRecord.key,
    joiner.apply(leftRecord.value, rightRecord.value)
);

```

- Inner join: Permite obtener solo los datos cuyas claves coincidan en ambos streams

```

// KStream + KStream
KStream<String, String> joined = left.join(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue,
    JoinWindows.of(TimeUnit.MINUTES.toMillis(5)),
    Joined.with(
        Serdes.String(), /* key */
        Serdes.Long(),   /* left value */
        Serdes.Double()  /* right value */
    );
// KTable + KTable
KTable<String, String> joined = left.join(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue
);
// KStream + KTable
KStream<String, String> joined = left.join(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue, /
    Joined.keySerde(Serdes.String())
        .withValueSerde(Serdes.Long())
);

```

- Left Join: Permite mantener las claves del primer stream y unir solo con las claves del segundo stream, ignorando las claves del segundo stream que no coincidan.

```

// KStream + KStream
KStream<String, String> joined = left.leftJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue,
    JoinWindows.of(TimeUnit.MINUTES.toMillis(5)),
    Joined.with(
        Serdes.String(),
        Serdes.Long(),
        Serdes.Double())
);
// KTable + KTable
KTable<String, String> joined = left.leftJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue
);
// KStream + KTable
KStream<String, String> joined = left.leftJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue,
    Joined.keySerde(Serdes.String())
        .withValueSerde(Serdes.Long())
);

```

- OuterJoin: Permite mantener las claves de ambos streams

```

// KStream + KStream
KStream<String, String> joined = left.outerJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue,
    JoinWindows.of(TimeUnit.MINUTES.toMillis(5)),
    Joined.with(
        Serdes.String(),
        Serdes.Long(),
        Serdes.Double())
);
// KTable + KTable
KTable<String, String> joined = left.outerJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue
);

```

15.6. Salida de datos

- En kafka, el resultado de un KStream o un KTable se envia a Kafka de nuevo.
- Para ello usamos los siguientes métodos:
 - To: Operación terminal. Devuelve a un topic los registros. Se puede implementar como se producen los mensajes:

```
stream.to("topic-salida", Produced.with(Serdes.String(), Serdes.Long()));
```

- Through: Permite enviar los datos a un topic y continuar con un stream.

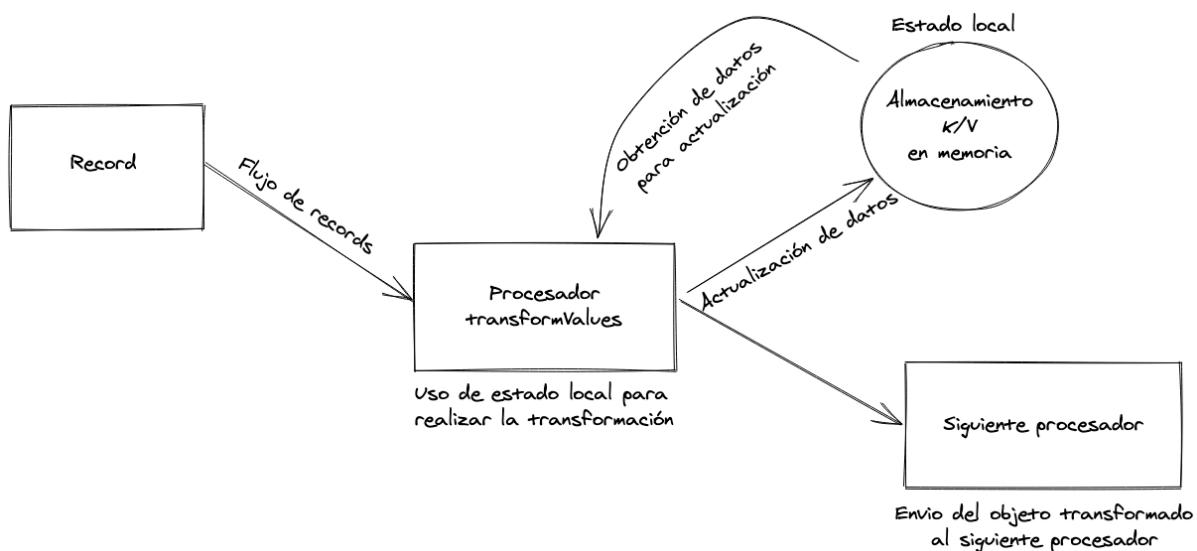
```
KStream<String, Long> newStream = stream.through("mitopic").map(...);  
KTable<String, Long> newTable = table.through("mitopic").map(...);
```

15.7. Estado en Streams

- Hasta ahora hemos estado trabajando en desarrollos sin estado
- Sin embargo, una de las grandes utilidades en kafka es el uso con estado.
- Permite mejorar la información recolectada por la aplicación

15.7.1. Transform Processor

- Se trata de la función más simple con estado de los KStreams
- **KStream.transformValues()**
- Es el mismo que mapValues() pero, en este caso, accede al **StateStore** para completar la tarea.
- Permite también programar tareas para que se ejecuten a intervalos por medio del método **punctuate()**



- Por medio del almacenamiento local podemos actualizar la información de los records.
- Como ejemplo, podríamos acumular la información de un cliente para ver cuanto se ha gastado.

15.8. Streams en java

- Lo primero que debemos tener en cuenta es la necesidad de librerías para poder desarrollar kafka streams.
- Se define como una topología de procesador:
 - Grafo de procesadores
 - Flujos de datos

15.8.1. Aproximaciones

15.8.1.1. Kafka Streams DSL

- Provee de operaciones comunes de transformación como map, filter, join y agregaciones.
- Se recomienda su uso como punto de partida para kafka streams
- Si estamos usando Scala, podemos usar el DSL para Scala

15.8.1.2. API del procesador

- Se trata de un API de bajo nivel que permite agregar y conectar procesadores e incluso interactuar directamente con los estados almacenados.
- Provee de mayor flexibilidad que el DSL, pero requiere de mayor trabajo y complejidad por el lado del desarrollador.

15.8.2. Librerías

- Las tres librerías que necesitamos son las siguientes:

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-clients</artifactId>
    <version>2.6.0</version>
</dependency>
```

- Para la implementación de Scala, debemos agregar al artefacto la versión de scala que queramos utilizar.
- Actualmente la 2.12 o la 2.13

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams-scala_2.12</artifactId>
    <version>2.6.0</version>
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams-scala_2.13</artifactId>
    <version>2.6.0</version>
</dependency>
```

15.8.3. Ejemplo básico

- Para el uso de kafka streams, primero se debe crear el StreamBuilder y el Topology

```
StreamsBuilder builder = new StreamsBuilder();
```

- Si queremos usar el API Processor debemos crear la topología

```
Topology topology = builder.build()
```

- Y crear el streams

```
KafkaStreams streams = new KafkaStreams(topology, properties)
```

- Una vez configurado, podemos iniciar el stream:

```
streams.start();
```

- Al parar la instancia, cerramos el stream

```
streams.close();
```

- Para optimizar el cierre, lo mejor es cerrar el flujo por medio de un hook de la vm de java

```
Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
```

15.9. Configuración

- Kafka Streams es altamente configurable.
- Las directivas obligatorias son:

```
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "mi-app-1.0.0");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
```

- Son obligatorias ya que no tienen valores por defecto
- La primera, **StreamsConfig.APPLICATION_ID_CONFIG**
 - Identifica la aplicación Kafka Stream, y debe ser un nombre único para todo el cluster de Kafka
 - También sirve como client.id y como group.id si no se han definido previamente.
 - El client.id identifica clientes distintos conectándose a kafka.
 - EL group.id permite gestionar los consumidores en un mismo grupo leyendo el mismo topic, y que todos los consumidores consuman el mismo topic a modo de cola.
- La segunda, **StreamsConfig.BOOTSTRAP_SERVERS_CONFIG** indica la lista de brokers a la que se conecta inicialmente.

15.10. Creación de Serde

- En kafka streams, la clase Serdes provee de métodos para crear instancias Serdes.

```
Serde<String> stringSerde = Serdes.String();
```

- Esta clase se requiere para serialización/deserialización.
- Permite:
 - String
 - Byte Array
 - Long
 - Integer
 - Double
- Esta clase permite definir de forma sencilla los cuatro parámetros de serialización/deserialización:
 - Serializador de clave
 - Serializador de valor
 - Deserializador de clave
 - Deserializador de valor
- En la siguiente clase, vemos un ejemplo de stream.

```

public class EjemploBasicoKafkaStreams {
    public static void main(String[] args) {
        // Generación de propiedades de conexión
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "ejemplo-basico-kafka-streams-1.0.0");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        StreamsConfig streamingConfig = new StreamsConfig(props);
        // Generación de serde para serializar/deserializar
        Serde<String> stringSerde = Serdes.String();
        // Construcción del StreamBuilder que genera la topología del procesador.
        StreamsBuilder builder = new StreamsBuilder();
        // Conexión del stream como consumidor desde topic-origen
        KStream<String, String> simpleFirstStream =
            builder.stream("topic-origen", Consumed.with(stringSerde, stringSerde));
        // Procesador (Primer nodo hijo) que pasa el texto a mayúsculas
        KStream<String, String> upperCasedStream =
            simpleFirstStream.mapValues(String::toUpperCase);
        // Conexión del stream al topic-destino como productor de mensajes
        upperCasedStream.to( "out-topic",
            Produced.with(stringSerde, stringSerde));

        KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsConfig);
        // Ejecución del hilo de kafka streams.
        kafkaStreams.start();
        Thread.sleep(1000);
        LOG.info("Apagando hilo");
        kafkaStreams.close();
    }
}

```

15.11. Topology

- Lo primero que debemos tener en cuenta es la topología.
- El API de kafka Streams permite generar una topología compleja de procesamiento de datos.
- En el siguiente ejemplo, vamos a procesar de una entrada, distintas salidas

```

public class TopologiaComplejaApp {
    public static void main(String[] args) {

        StreamsConfig streamsConfig = new StreamsConfig(getProperties());
        JsonSerializer<Compra> compraJsonSerializer = new JsonSerializer<>();
        JsonDeserializer<Compra> compraJsonDeserializer = new JsonDeserializer<>(Compra.class);
        Serde<Compra> compraSerde = Serdes.serdeFrom(compraJsonSerializer,
compraJsonDeserializer);
        Serde<String> stringSerde = Serdes.String();
        StreamsBuilder streamsBuilder = new StreamsBuilder();
        KStream<String, Compra> compraKStream =
            streamsBuilder.stream("topic-transacciones", Consumed.with(stringSerde,
compraSerde))
                .mapValues(compra -> Compra.builder(compra).enmascararTarjetaCredito()
).build());
        KStream<String, PatronCompra> patronKStream =
            compraKStream.mapValues(compra -> CompraPatron.builder(compra).build());
        patronKStream.to("topic-patrones", Produced.with(stringSerde
, compraPatronSerde));
        KStream<String, Recompensas> recompensasKStream =
            compraKStream.mapValues(compra -> Recompensas.builder(compra).build());
        recompensasKStream.to("topic-recompensas", Produced.with(stringSerde
, recompensasSerde));
        compraKStream.to("topic-compras", Produced.with(stringSerde, compraSerde));
        KafkaStreams kafkaStreams = new KafkaStreams(streamsBuilder.build()
, streamsConfig);
        kafkaStreams.start();
    }
}

```

- En este caso, las salidas son topic-compras, topic-recompensas y topic-patrones.
- Primero declaramos el nodo origen:

```

public class TopologiaComplejaApp {
    public static void main(String[] args) {
...
    KStream<String, Compra> compraKStream =
        streamsBuilder.stream("topic-transacciones", Consumed.with(stringSerde,
compraSerde))
            .mapValues(compra -> Compra.builder(compra).enmascararTarjetaCredito()
).build());
...
}

```

- Tenemos un Serde personalizado para compra.
- Podemos indicar solo un topic, o varios seguido de comas, e incluso un patrón.

- Lo siguiente que hacemos es el mapeo de los valores, que devuelve un objeto del mismo tipo, pero con la tarjeta de crédito enmascarada.

```
Compra.builder(compra).enmascararTarjetaCredito().build()
```

- Para el segundo procesador, creamos una nueva instancia de tipo KStream que permite crear objetos de tipo PatronCompra.

```
public class TopologiaComplejaApp {
    ...
    KStream<String, PatronCompra> patronKStream =
        compraKStream.mapValues(compra -> CompraPatron.builder(compra).build());
        patternKStream.to("topic-patrones", Produced.with(stringSerde
, compraPatronSerde));
    ...
}
```

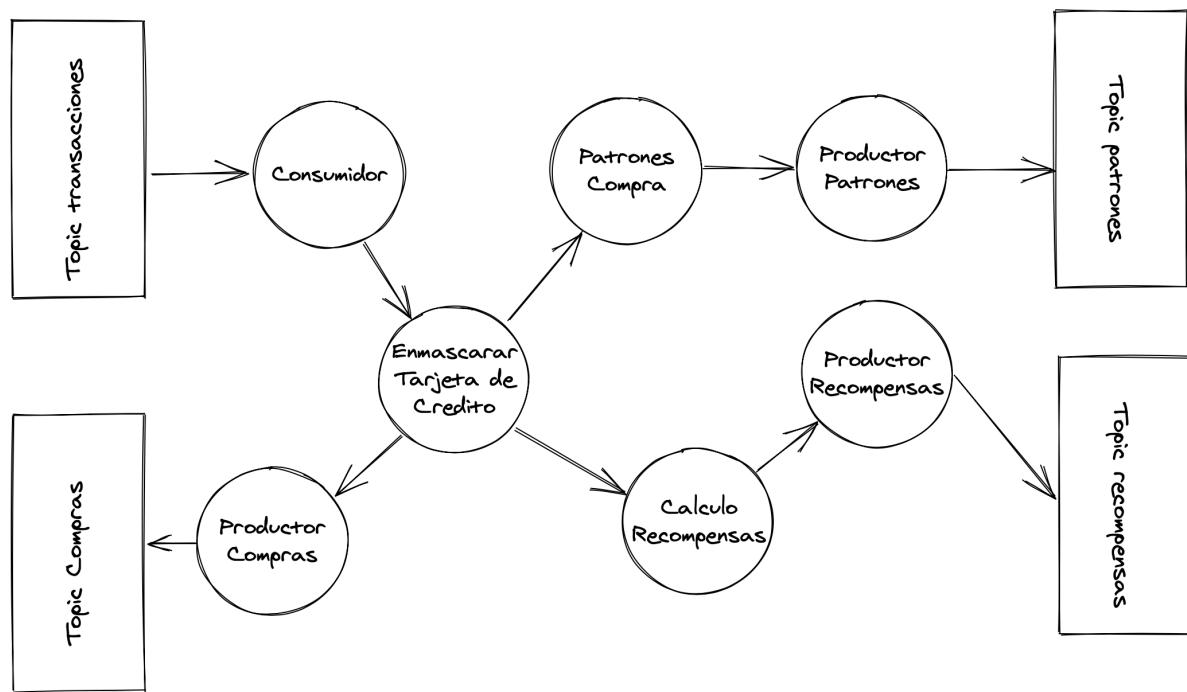
- En este caso, obtiene los objetos generados por el nodo anterior, lso mapea a un objeto mas simple y lo envía al topic-patrones.
- En el tercer nodo,

```
public class TopologiaComplejaApp {
    public static void main(String[] args) {
        ...
        KStream<String, Recompensas> recompensasKStream =
            compraKStream.mapValues(compra -> Recompensas.builder(compra).build());
            recompensasKStream.to("topic-recompensas", Produced.with(stringSerde
, recompensasSerde));
        ...
    }
}
```

- En este caso, creamos un nuevo KStream que obtiene los datos del KStream de compras, combierte el objeto en recompensas y lo envia al topic-recompensas.
- Por último, generamos el último procesador, que guarda los datos de compra con la tarjeta de crédito enmascarada en un nuevo topic.

```
public class TopologiaComplejaApp {
    public static void main(String[] args) {
        ...
        compraKStream.to("topic-compras", Produced.with(stringSerde, compraSerde));
        ...
    }
}
```

- Una representación de la topología generada sería la siguiente



15.12. Serdes personalizados

- Está claro que para poder serializar y deserializar objetos debemos usar tipos sencillos o más complejos
- Para crear un serde personalizado, debemos implementar el serializador de kafka.

```

package org.apache.kafka.common.serialization;

import org.apache.kafka.common.header.Headers;
import java.io.Closeable;
import java.util.Map;

public interface Serializer<T> extends Closeable {

    default void configure(Map<String, ?> configs, boolean isKey) {
    }

    byte[] serialize(String topic, T data);

    default byte[] serialize(String topic, Headers headers, T data) {
        return serialize(topic, data);
    }

    @Override
    default void close() {
    }
}
  
```

- Una forma de crear un serde personalizado sería la siguiente

```
public class JsonSerializer<T> implements Serializer<T> {
    private Gson gson = new Gson();

    @Override
    public void configure(Map<String, ?> map, boolean b) {}

    @Override
    public byte[] serialize(String topic, T t) {
        return gson.toJson(t).getBytes(Charset.forName("UTF-8"));
    }

    @Override
    public void close() {}

}
```

- En este caso obtenemos un objeto y lo pasamos a json por medio de la librería gson que hemos usado con anterioridad.
- Para poder deserializar, debemos implementar la interfaz de deserialización de kafka

```
package org.apache.kafka.common.serialization;

import org.apache.kafka.common.header.Headers;
import java.io.Closeable;
import java.util.Map;

public interface Deserializer<T> extends Closeable {

    default void configure(Map<String, ?> configs, boolean isKey) {}

    T deserialize(String topic, byte[] data);

    default T deserialize(String topic, Headers headers, byte[] data) {
        return deserialize(topic, data);
    }

    @Override
    default void close() {}

}
```

- Un ejemplo de deserializador sería el siguiente:

```

public class JsonDeserializer<T> implements Deserializer<T> {
    private Gson gson = new Gson();
    private Class<T> deserializedClass;
    public JsonDeserializer(Class<T> deserializedClass) {
        this.deserializedClass = deserializedClass;
    }

    public JsonDeserializer() {
    }

    @Override
    @SuppressWarnings("unchecked")
    public void configure(Map<String, ?> map, boolean b) {
        if(deserializedClass == null) {
            deserializedClass = (Class<T>) map.get("serializedClass");
        }
    }
    @Override
    public T deserialize(String s, byte[] bytes) {
        if(bytes == null){
            return null;
        }
        return gson.fromJson(new String(bytes),deserializedClass);
    }
    @Override
    public void close() {
    }
}

```

15.13. Branching

- Una de las operaciones más interesantes que se pueden hacer definiendo una topología es el branching.
- El branching permite definir una ramificación, una condición donde un record puede ir a una rama o a otra según la condición que cumplan.
- Para ello definimos un predicado que permite definir cuando cumple una determinada condición.

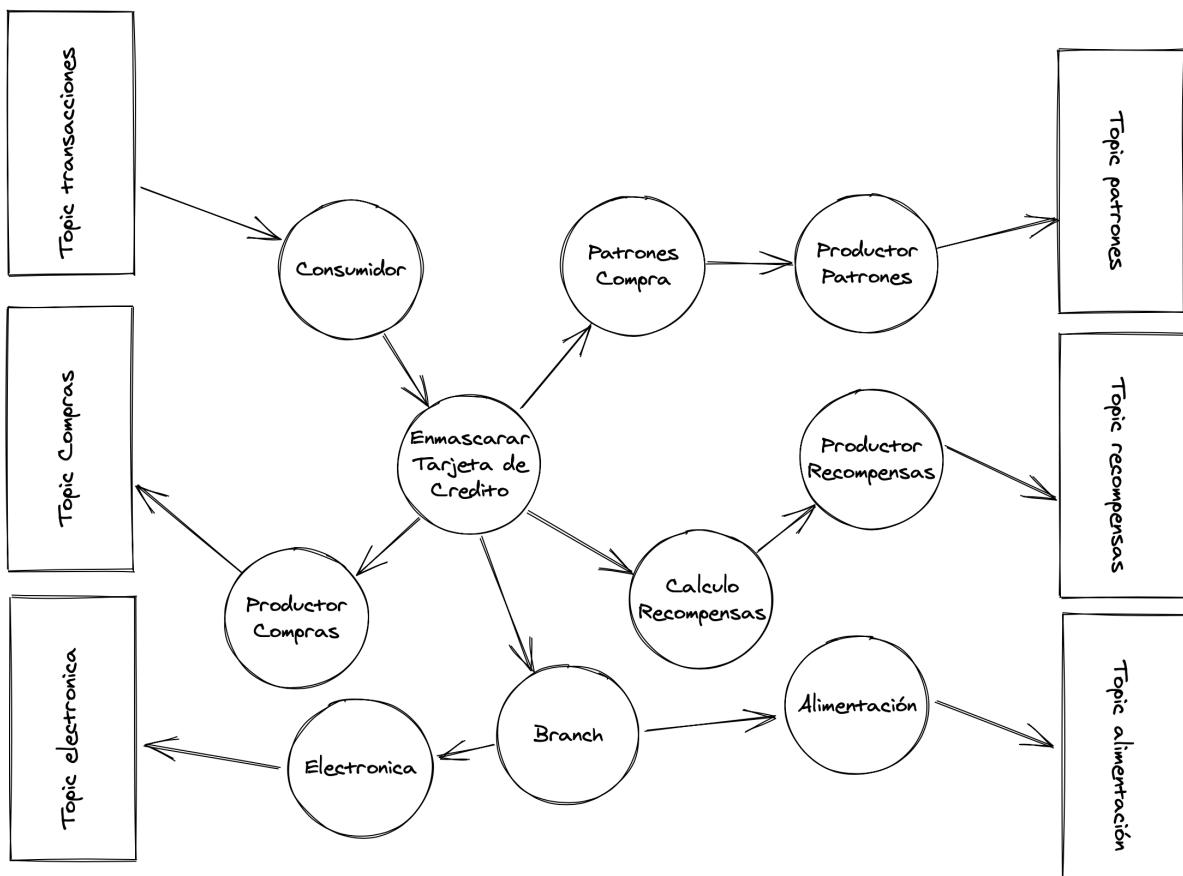
```

Predicate<String, Compra> esAlimentacion =
    (key, compra) -> compra.getDepartamento().equalsIgnoreCase("alimentacion");
Predicate<String, Compra> esElectronica =
    (key, compra) -> compra.getDepartamento().equalsIgnoreCase("electronica");

int posAlimentacion = 0;
int posElectronica = 1;
KStream<String, Compra>[] arrayKStream =
    compraKStream.branch(esAlimentacion, esElectronica);
arrayKStream[posAlimentacion].to( "topic-alimentacion", Produced.with(stringSerde, compraSerde));
arrayKStream[posElectronica].to("electronica", Produced.with(stringSerde, compraSerde));

```

- El esquema quedaría como sigue



15.14. Filtering

- Otra opción es el filtrado de información antes de pasarlo a otro topic.
- Para ello, creamos un nuevo nodo que mande la información filtrada a un topic.
- Un ejemplo sería evitar que todas las operaciones de compra se almacenaran, y que solo se almacenara información relevante.

```
KeyValueMapper<String, Compra, Long> fechaDeCompraComoKey =  
    (key, compra) -> compra.getFechaCompra().getTime();  
KStream<Long, Compra> filtroKStream = compraKStream((key, compra) ->  
    compra.getPrecio() > 10.00).selectKey(fechaDeCompraComoKey);  
filtroKStream.to("compras", Produced.with(Serdes.Long(), compraSerde));
```

15.15. Desarrollo con topologías

- Para poder visualizar las operaciones de las topologías, podemos tener consumidores que nos muestren el resultado de las operaciones.
- Sin embargo, es mas sencillo indicar un método de kstream llamado print.
- Para mostrar el resultado por consola solo tenemos que indicar que vamos a imprimir los resultados.
- Estos resultados se pueden definir con un label.
- La recomendación es que el label que cada uno tenga sea el nombre del topic destino para entender a donde iría cada resultado.

```
patronKStream.print(Printed.<String, PatronCompra>toSysOut().withLabel("topic-patrones"));  
recompensaKStream.print(Printed.<String, Recompensa>toSysOut().withLabel("topic-recompensas"));  
compraKStream.print(Printed.<String, Compra>toSysOut().withLabel("topic-compras"));
```

15.16. Lab: Kafka Streams en Java

- El objetivo es generar distintos ejemplos para poder visualizar el potencial de los kafka streams.

15.16.1. Ejemplo stream básico

- Primero vamos a generar un proyecto de stream básico que obtenga información de envío y lo transforme.
- Llamamos al proyecto Stream01Base
- Para ello, primero creamos un productor.
- Creamos un nuevo proyecto llamado Stream01Base basado en nuestra plantilla
- En el Pom, agregamos la dependencia de kafka-streams

```
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams</artifactId>
    <version>2.6.0</version>
</dependency>
```

- Actualizamos el proyecto con maven → update project pulsando botón derecho en el nombre del proyecto.

15.16.1.1. Creación del productor

- Usaremos un productor ya generado anteriormente:

```

package com.curso.kafka.streams;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

public class Productor {

    public static final String TOPIC = "stream-topic-origen";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());

        KafkaProducer<String, String> producer = new KafkaProducer<>(properties);

        Thread shutdownHook = new Thread(producer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);
        int i = 0;
        while(true) {
            i++;
            String key = "key " + i ;
            String value = "value " + i;
            ProducerRecord<String, String> record = new ProducerRecord<>(TOPIC, key, value);
            producer.send(record);
            Thread.sleep(1000);
        }
    }
}

```

- Este productor mandará mensajes al topic donde nos conectaremos desde el stream.

15.16.1.2. Creación de KafkaStream

- Ahora creamos la clase de stream llamada Stream01Base donde vamos a crear un flujo de un topic a otro realizando una pequeña transformación.
- Primero indicamos origen y destino en dos variables

```

private static final String TOPIC_ORIGEN = "stream-topic-origen";
private static final String TOPIC_DESTINO = "stream-topic-destino";

```

- Luego vamos a crear un método main donde vamos a agregar las propiedades mínimas del stream

```
public static void main(String[] args) throws InterruptedException {
    // Generación de propiedades de conexión
    Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "Stream01Base");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
```

- Creamos el objeto de serialización, deserialización que permita realizarlo con un String.

```
Serde<String> stringSerde = Serdes.String();
```

- Creamos la topología básica del procesador

```
StreamsBuilder builder = new StreamsBuilder();
```

- Conectamos el consumo de mensajes del topic origen y le indicamos como va a consumir la clave y el valor del registro.

```
KStream<String, String> simpleFirstStream =
    builder.stream(TOPIC_ORIGEN, Consumed.with(stringSerde, stringSerde));
```

- Creamos el primer nodo de la topología que transforma cada registro por medio de um mapeo

```
KStream<String, String> upperCasedStream =
    simpleFirstStream.mapValues(value -> value.toUpperCase());
```

- Indicamos que el nodo de la topología envie los mensajes a un topic destino serializando los mensajes con el Serde que indicamos anteriormente

```
upperCasedStream.to( TOPIC_DESTINO,
    Produced.with(stringSerde, stringSerde));
```

- Agregamos una instrucción de log que permite mostrar los resultados transformados por el nodo
- Definimos como etiqueta el topic destino para saber que hemos enviado al mismo

```
upperCasedStream.print(Printed.<String, String>toSysOut().withLabel(TOPIC_DESTINO));
```

- Construimos el KafkaStreams para que gestione e inicie el proceso con la configuración que hemos generado

```
KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),props);
```

- Agregamos el hook de parada de máquina virtual para cerrar correctamente la conexión

```
Thread shutdownHook = new Thread(kafkaStreams::close);
Runtime.getRuntime().addShutdownHook(shutdownHook);
```

- Por último, iniciamos el proceso

```
kafkaStreams.start();
```

- La clase completa quedará como sigue

```

package com.curso.kafka.streams;

import java.util.Properties;

import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Kstream.Consumed;
import org.apache.kafka.streams.Kstream.KStream;
import org.apache.kafka.streams.Kstream.Printed;
import org.apache.kafka.streams.Kstream.Produced;

public class Stream01Base {

    private static final String TOPIC_ORIGEN = "stream-topic-origen";
    private static final String TOPIC_DESTINO = "stream-topic-destino";

    public static void main(String[] args) throws InterruptedException {
        // Generación de propiedades de conexión
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "Stream01Base");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");

        // Generación de serde para serializar/deserializar
        Serde<String> stringSerde = Serdes.String();
        // Construcción del StreamBuilder que genera la topología del procesador.
        StreamsBuilder builder = new StreamsBuilder();
        // Conexión del stream como consumidor desde topic-origen
        KStream<String, String> simpleFirstStream =
            builder.stream(TOPIC_ORIGEN, Consumed.with(stringSerde, stringSerde));
        // Procesador (Primer nodo hijo) que pasa el texto a mayúsculas
        KStream<String, String> upperCasedStream =
            simpleFirstStream.mapValues(value -> value.toUpperCase());
        // Conexión del stream al topic-destino como productor de mensajes
        upperCasedStream.to( TOPIC_DESTINO,
            Produced.with(stringSerde, stringSerde));
        // Log de desarrollo
        upperCasedStream.print(Printed.<String, String>toSysOut().withLabel(TOPIC_DESTINO));

        KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),props);
        Thread shutdownHook = new Thread(kafkaStreams::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);
        // Ejecución del hilo de kafka streams.
        kafkaStreams.start();

    }
}

```

15.16.1.3. Creación de topics

- La clase stream necesita tanto el stream-topic-origen como el stream-topic-destino
- Para ello, vamos a generar los topics

```
kafka-topics.sh --bootstrap-server localhost:9092 --create --topic stream-topic-origen --partitions 3 --replication-factor 1
Created topic stream-topic-origen.
kafka-topics.sh --bootstrap-server localhost:9092 --create --topic stream-topic-destino --partitions 3 --replication-factor 1
Created topic stream-topic-destino.
```

15.16.1.4. Prueba de Ejecución

- Primero ejecutamos el productor para que vaya mandando mensajes al topic origen.
 - Pulsamos botón derecho en la clase y seleccionamos run as → java application
- El productor estará enviando mensajes continuamente.
- Ahora iniciamos el stream.
- Si observamos la consola:

```
[stream-topic-destino]: key 1, VALUE 1
[stream-topic-destino]: key 2, VALUE 2
[stream-topic-destino]: key 3, VALUE 3
[stream-topic-destino]: key 4, VALUE 4
[stream-topic-destino]: key 5, VALUE 5
[stream-topic-destino]: key 6, VALUE 6
[stream-topic-destino]: key 7, VALUE 7
```

- En el topic destino estamos insertando claves y valores, pero el valor está en mayúsculas, que es lo que estamos modificando con mapValues()

15.16.2. Uso de Avro con Kafka Streams

- Como ya tenemos una estructura de Avro generada, vamos a usar avro para integrarlo con los streams.
- Para ello, creamos un proyecto basado en la plantilla llamado Stream02Avro
- Modificamos el artifactId para que coincida con el nombre que hemos dado:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.curso.kafka.streams</groupId>
  <artifactId>Stream02Avro</artifactId>
  <version>0.0.1-SNAPSHOT</version>
```

- Luego agregamos un nuevo repositorio para usar las librerías de confluent.

```
<repositories>
  <repository>
    <id>confluent</id>
    <url>http://packages.confluent.io/maven/</url>
  </repository>
</repositories>
```

- Agregamos nuestro proyecto avro que tiene las clases que vamos a utilizar.

```
<dependency>
  <groupId>com.kafka</groupId>
  <artifactId>avro-tools</artifactId>
  <version>1.0.0</version>
</dependency>
```

- Agregamos la dependencia de Stream

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams</artifactId>
  <version>2.6.0</version>
</dependency>
```

- Agregamos el soporte de Serde para Avro. En este caso, sin el repositorio de confluent, no se descargará la librería

```
<dependency>
  <groupId>io.confluent</groupId>
  <artifactId>kafka-streams-avro-serde</artifactId>
  <version>6.0.0</version>
</dependency>
```

- Por último, agregamos la librería de apoyo gson para usar nuestro servicio de clima

```
<dependency>
  <groupId>com.google.code.gson</groupId>
  <artifactId>gson</artifactId>
  <version>2.8.1</version>
</dependency>
```

- El resultado de crear el pom.xml será el siguiente

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.curso.kafka.streams</groupId>
  <artifactId>Stream02Avro</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.0</version>
        <configuration>
          <source>11</source>
          <target>11</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <repositories>
    <repository>
      <id>confluent</id>
      <url>http://packages.confluent.io/maven/</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency>
      <groupId>com.kafka</groupId>
      <artifactId>avro-tools</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>2.6.0</version>
    </dependency>
    <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-streams</artifactId>
      <version>2.6.0</version>
    </dependency>
    <dependency>
      <groupId>io.confluent</groupId>
      <artifactId>kafka-streams-avro-serde</artifactId>
      <version>6.0.0</version>
    </dependency>
    <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
      <version>2.8.1</version>
    </dependency>
  </dependencies>
</project>

```

15.16.2.1. Creación del productor

- Para el productor, copiaremos la clase OpenWeatherMap.java del proyecto Avro anterior.
- La clase será la siguiente:

```

package com.curso.kafka.streams;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.curso.kafka.avro.model.Clima;
import com.curso.kafka.avro.model.Datos;
import com.curso.kafka.avro.model.Detalles;
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;;

```

```

public class OpenWeatherMap {
    private static String API_KEY = "<NUESTRA_API_KEY>";

    private static Map<String, Object> jsonToMap(String responseString) {
        return new Gson().fromJson(responseString, new TypeToken<HashMap<String, Object>>() {
            }.getType());
    }

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static Clima getWeatherFromOpenWeatherMap(String city) throws IOException {

        String url = "http://api.openweathermap.org/data/2.5/weather?q=" + city + "&appid=" + API_KEY;
        StringBuilder result = new StringBuilder();
        URL urlRequest = new URL(url);
        URLConnection connection = urlRequest.openConnection();
        BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String line = null;
        while ((line = reader.readLine()) != null) {
            result.append(line);
        }
        reader.close();
        System.out.println(result);
        Map resultMap = jsonToMap(result.toString());

        Map mainMap = jsonToMap(resultMap.get("main").toString());
        List<Object> list = (List<Object>) resultMap.get("weather");
        List<Detalles> detalles = new ArrayList();
        for (Object detail : list) {
            Map<String, Object> detailMap = (Map) detail;
            detalles.add(Detalles.newBuilder().setId((long) Double.parseDouble(detailMap.get("id").toString()))
                .setPrincipal(detailMap.get("main").toString()).setIcono(detailMap.get("icon").toString())
                .setDescripcion(detailMap.get("description").toString()).build());
        }

        Datos datos = Datos.newBuilder().setPresion((int) Double.parseDouble(mainMap.get("pressure").toString()))
            .setHumedad((int) Double.parseDouble(mainMap.get("humidity").toString()))
            .setTemp((int) Double.parseDouble(mainMap.get("temp").toString()))
            .setTempMax((int) Double.parseDouble(mainMap.get("temp_max").toString()))
            .setTempMin((int) Double.parseDouble(mainMap.get("temp_min").toString())).build();
        Clima clima = Clima.newBuilder().setId((long) Double.parseDouble(resultMap.get("id").toString()))
            .setNombre(resultMap.get("name").toString()).setDatos(datos).setDetalles(detalles).build();
        return clima;
    }
}

```

- Luego creamos nuestro productor, usando el mismo ejemplo que usamos para Avro.
- La clase quedaría como sigue

```

package com.curso.kafka.streams;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import com.curso.kafka.avro.model.Clima;

import io.confluent.kafka.serializers.KafkaAvroSerializer;
import io.confluent.kafka.serializers.KafkaAvroSerializerConfig;

public class ProductorSchemaRegistry {

    public static final String CITY = "madrid";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());
        properties.put(KafkaAvroSerializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");

        KafkaProducer<String,Clima> producer = new KafkaProducer<>(properties);

        Thread shutdownHook = new Thread(producer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);

        while(true) {
            Clima clima = OpenWeatherMap.getWeatherFromOpenWeatherMap(CITY);
            ProducerRecord<String, Clima> record = new ProducerRecord<>(Stream02avro.TOPIC_ORIGEN, CITY, clima);
            producer.send(record);
            Thread.sleep(1500);
        }
    }
}

```

- La clase hace referencia al topic origen de la clase Stream02Avro, que no existe.
- En cuanto la creemos compilará la clase sin problemas.

15.16.2.2. Creación del consumidor

- Usaremos el mismo consumidor ConsumidorSchemaRegistry para ver que los datos efectivamente están en destino.
- Para ello, copiamos la clase de nuevo usando como topic el topic destino del stream.

```

package com.curso.kafka.streams;

import java.io.IOException;
import java.time.Duration;
import java.util.Collections;
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;

import com.curso.kafka.avro.model.Clima;

import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializerConfig;

public class ConsumidorSchemaRegistry {

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.class.getName());
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "consumidorClimaSchemaRegistry");
        properties.put(KafkaAvroDeserializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");
        properties.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, true);

        KafkaConsumer<String, Clima> consumer = new KafkaConsumer<>(properties);

        Thread shutdownHook = new Thread(consumer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);

        consumer.subscribe(Collections.singletonList(Stream02avro.TOPIC_DESTINO));
        while(true) {
            ConsumerRecords<String, Clima> records = consumer.poll(Duration.ofMillis(100));
            for(ConsumerRecord<String, Clima> record : records) {
                System.out.println("ID: " + record.key() + " - value: " + record.value().toString());
            }
        }
    }
}

```

15.16.2.3. Creación del Stream

- En este caso, habrá que agregar la serialización Avro e indicar la ruta al Schema Registry para comprobar el esquema que le corresponde.
- Primero creamos la clase Stream02Avro y agregamos dos campos públicos de TOPIC_ORIGEN y TOPIC_DESTINO

```

public class Stream02avro {

    public static final String TOPIC_ORIGEN = "stream-avro-topic-origen";
    public static final String TOPIC_DESTINO = "stream-avro-topic-destino";

```

- En cuanto lo creemos, se solucionarán los errores en el productor y el consumidor.
- Luego, en un main, agregamos las propiedades de configuración comunes:

```

Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "ejemplo-basico-kafka-streams-1.0.0");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");

```

- No hemos configurado por defecto los Serdes ni el SchemaRegistry. Lo vamos a hacer en la creación del Serde, ya que podemos tener múltiples serdes de Avro y que apunten a distintas instancias de registro.
- Para ello, creamos el Serde de la clave y del valor

```

Serde<String> stringSerde = Serdes.String();
Serde<Clima> climaSerde = new SpecificAvroSerde<>();
climaSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://localhost:8081"), false);

```

- Comprobamos como definimos para climaSerde la localización de su Schema Registry, siendo un SpecificAvroSerde
- Creamos el StreamBuilder y lo iniciamos consumiendo el topic origen con los serdes declarados

```

StreamsBuilder builder = new StreamsBuilder();
KStream<String, Clima> simpleFirstStream =
    builder.stream(TOPIC_ORIGEN, Consumed.with(stringSerde, climaSerde));

```

- Como en el caso anterior vamos a mapear un campo de la clase Clima para que se pase a mayúsculas, el campo nombre.

```

KStream<String, Clima> upperCasedStream =
    simpleFirstStream.mapValues( (value) -> {
        value.setNombre((value.getNombre()+'').toUpperCase());
        return value;
    });

```

- Lo enviamos al topic destino, usando climaSerde y generando un log para ver el resultado enviado al topic

```
upperCasedStream.to( TOPIC_DESTINO,  
    Produced.with(stringSerde, climaSerde));  
upperCasedStream.print(Printed.<String,Clima>toSysOut().withLabel(TOPIC_DESTINO));
```

- Por último, creamos el KafkaStream, agregamos el hook e iniciamos el servicio.

```
KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),props);  
Thread shutdownHook = new Thread(kafkaStreams::close);  
Runtime.getRuntime().addShutdownHook(shutdownHook);  
// Ejecución del hilo de kafka streams.  
kafkaStreams.start();
```

- La clase completa quedará de la siguiente forma

```

package com.curso.kafka.streams;

import java.util.Collections;
import java.util.Properties;

import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Kstream.Consumed;
import org.apache.kafka.streams.KStream;
import org.apache.kafka.streams.Kstream.Printed;
import org.apache.kafka.streams.Kstream.Produced;

import com.curso.kafka.avro.model.Clima;

import io.confluent.kafka.serializers.AbstractKafkaSchemaSerDeConfig;
import io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde;

public class Stream02avro {

    public static final String TOPIC_ORIGEN = "stream-avro-topic-origen";
    public static final String TOPIC_DESTINO = "stream-avro-topic-destino";

    public static void main(String[] args) throws InterruptedException {
        // Generación de propiedades de conexión
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "ejemplo-basico-kafka-streams-1.0.0");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        //props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, SpecificAvroSerde.class.getName());
        //props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");

        // Generación de serde para serializar/deserializar
        Serde<String> stringSerde = Serdes.String();
        Serde<Clima> climaSerde = new SpecificAvroSerde<>();
        climaSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
        "http://localhost:8081"), false);
        // Construcción del StreamBuilder que genera la topología del procesador.
        StreamsBuilder builder = new StreamsBuilder();
        // Conexión del stream como consumidor desde topic-origen
        KStream<String, Clima> simpleFirstStream =
            builder.stream(TOPIC_ORIGEN, Consumed.with(stringSerde, climaSerde));
        // Procesador (Primer nodo hijo) que pasa el texto a mayúsculas
        KStream<String, Clima> upperCasedStream =
            simpleFirstStream.mapValues( (value) -> {
                value.setNombre((value.getNombre()+"").toUpperCase());
                return value;
            });
        // Conexión del stream al topic-destino como productor de mensajes
        upperCasedStream.to( TOPIC_DESTINO,
            Produced.with(stringSerde, climaSerde));
        // Log de desarrollo
        upperCasedStream.print(Printed.<String,Clima>toSysOut().withLabel(TOPIC_DESTINO));

        KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),props);
        Thread shutdownHook = new Thread(kafkaStreams::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);
        // Ejecución del hilo de kafka streams.
        kafkaStreams.start();

    }

}

```

15.16.2.4. Creación de los topics

- Volvemos a crear los topics con la shell

```
[kafka@kafka-server docker-kafka]$ kafka-topics.sh --bootstrap-server localhost:9092 --create --topic stream-avro-topic-origen
Created topic stream-avro-topic-origen.
[kafka@kafka-server docker-kafka]$ kafka-topics.sh --bootstrap-server localhost:9092 --create --topic stream-avro-topic-destino
Created topic stream-avro-topic-destino.
```

15.16.2.5. Prueba de Ejecución

- Para ejecutarlo, comenzamos con el productor y luego ejecutamos nuestro stream.
- Si nos fijamos en el resultado de la ejecución del stream, veremos como hemos transformado el nombre a mayúsculas.

```
[stream-avro-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1009, "humedad": 82, "tempMin": 286.0, "tempMax": 287.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
[stream-avro-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1009, "humedad": 82, "tempMin": 286.0, "tempMax": 287.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
[stream-avro-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1009, "humedad": 82, "tempMin": 286.0, "tempMax": 287.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
[stream-avro-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1009, "humedad": 82, "tempMin": 286.0, "tempMax": 287.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
[stream-avro-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1009, "humedad": 82, "tempMin": 286.0, "tempMax": 287.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
[stream-avro-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1009, "humedad": 82, "tempMin": 286.0, "tempMax": 287.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
[stream-avro-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1009, "humedad": 82, "tempMin": 286.0, "tempMax": 287.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
[stream-avro-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1009, "humedad": 82, "tempMin": 286.0, "tempMax": 287.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
```

- Si ejecutamos ahora el consumidor de Avro, veremos como los mensajes efectivamente están siendo inyectados al topic destino

```
ID: madrid - value: {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1010, "humedad": 87, "tempMin": 286.0, "tempMax": 288.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
ID: madrid - value: {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1010, "humedad": 87, "tempMin": 286.0, "tempMax": 288.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
ID: madrid - value: {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 286.0, "presion": 1010, "humedad": 87, "tempMin": 286.0, "tempMax": 288.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
```

- Paramos todas las instancias

15.16.3. Topology

- En este caso, vamos a mostrar como podemos generar topologías complejas que publiquen en distintos topics.
- Para ello usaremos el mismo Productor que anteriormente y la clase openweathermap
- La clase OpenWeatherMap es la que sigue:

```

package com.curso.kafka.streams;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.URL;
import java.net.URLConnection;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import com.curso.kafka.avro.model.Clima;
import com.curso.kafka.avro.model.Datos;
import com.curso.kafka.avro.model.Detalles;
import com.google.gson.Gson;
import com.google.gson.reflect.TypeToken;;

public class OpenWeatherMap {
    private static String API_KEY = "68a83fec9786050d20515df74b308c4a";

    private static Map<String, Object> jsonToMap(String responseString) {
        return new Gson().fromJson(responseString, new TypeToken<HashMap<String, Object>>() {
            }.getType());
    }

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static Clima getWeatherFromOpenWeatherMap(String city) throws IOException {

        String url = "http://api.openweathermap.org/data/2.5/weather?q=" + city + "&appid=" + API_KEY;
        StringBuilder result = new StringBuilder();
        URL urlRequest = new URL(url);
        URLConnection connection = urlRequest.openConnection();
        BufferedReader reader = new BufferedReader(new InputStreamReader(connection.getInputStream()));
        String line = null;
        while ((line = reader.readLine()) != null) {
            result.append(line);
        }
        reader.close();
        System.out.println(result);
        Map resultMap = jsonToMap(result.toString());

        Map mainMap = jsonToMap(resultMap.get("main").toString());
        List<Object> list = (List<Object>) resultMap.get("weather");
        List<Detalles> detalles = new ArrayList();
        for (Object detail : list) {
            Map<String, Object> detailMap = (Map) detail;
            detalles.add(Detalles.newBuilder().setId((long) Double.parseDouble(detailMap.get("id").toString()))
                .setPrincipal(detailMap.get("main").toString()).setIcono(detailMap.get("icon").toString())
                .setDescripcion(detailMap.get("description").toString()).build());
        }

        Datos datos = Datos.newBuilder().setPresion((int) Double.parseDouble(mainMap.get("pressure").toString()))
            .setHumedad((int) Double.parseDouble(mainMap.get("humidity").toString()))
            .setTemp((int) Double.parseDouble(mainMap.get("temp").toString()))
            .setTempMax((int) Double.parseDouble(mainMap.get("temp_max").toString()))
            .setTempMin((int) Double.parseDouble(mainMap.get("temp_min").toString())).build();
        Clima clima = Clima.newBuilder().setId((long) Double.parseDouble(resultMap.get("id").toString()))
            .setNombre(resultMap.get("name").toString()).setDatos(datos).setDetalles(detalles).build();
        return clima;
    }
}

```

- Lo segundo será crear el productor, que es el mismo que el anterior, solo que publicará en el topic declarado en la clase que crearemos para el stream.

```

package com.curso.kafka.streams;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import com.curso.kafka.avro.model.Clima;

import io.confluent.kafka.serializers.KafkaAvroSerializer;
import io.confluent.kafka.serializers.KafkaAvroSerializerConfig;

public class ProductorSchemaRegistry {

    public static final String CITY = "madrid";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());
        properties.put(KafkaAvroSerializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");

        KafkaProducer<String,Clima> producer = new KafkaProducer<>(properties);

        Thread shutdownHook = new Thread(producer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);

        while(true) {
            Clima clima = OpenWeatherMap.getWeatherFromOpenWeatherMap(CITY);
            ProducerRecord<String, Clima> record = new ProducerRecord<>(Stream03Topology.TOPIC_ORIGEN, CITY, clima);
            producer.send(record);
            Thread.sleep(1500);
        }
    }
}

```

15.16.3.1. Generación de la clase Stream03Topology

- Para ello creamos una nueva clase con un main.
- Generamos los nombres de los topics que vamos a usar. En este caso usaremos tres, origen, destino y un tercero.

```

public class Stream03Topology {

    public static final String TOPIC_ORIGEN = "stream-topology-topic-origen";
    public static final String TOPIC_DESTINO = "stream-topology-topic-destino";
    public static final String TOPIC_DATOS = "stream-topology-topic-datos";
}

```

- Generamos en el main la configuración del stream

```
public static void main(String[] args) throws InterruptedException {
    // Generación de propiedades de conexión
    Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "Stream03Topology");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
```

- Generamos los serdes particulares e indicamos donde está su Schema Registry

```
Serde<String> stringSerde = Serdes.String();
Serde<Clima> climaSerde = new SpecificAvroSerde<>();
Serde<Datos> datosSerde = new SpecificAvroSerde<>();
climaSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://localhost:8081"), false);
datosSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://localhost:8081"), false);
```

- Creamos el constructor de topología (StreamBuilder) y comenzamos consumiendo mensajes del topic origen.

```
----09:05 di
StreamsBuilder builder = new StreamsBuilder();
KStream<String, Clima> simpleFirstStream =
    builder.stream(TOPIC_ORIGEN, Consumed.with(stringSerde, climaSerde));
----
```

- Procesamos el nodo para pasar a mayúsculas el nombre del campo usando map

```
KStream<String, Clima> upperCasedStream =
simpleFirstStream.mapValues( (value) -> {
    value.setNombre((value.getNombre()+"").toUpperCase());
    return value;
});
```

- Creamos un segundo nodo basado en la salida del simpleFirstStream, que obtenga de clave el nombre del valor, y de valor solo los datos asociados

```
KStream<String, Datos> datosKStream = upperCasedStream.<String,Datos>map((key,value) -> new KeyValue<String,Datos>(value
    .getNombre().toString(), value.getDatos()));
```

- Imprimimos los datos por pantalla y publicamos los datos al topic destino

```
datosKStream.print(Printed.<String,Datos>toSysOut().withLabel(TOPIC_DATOS));
datosKStream.to(TOPIC_DATOS, Produced.<String, Datos>with(stringSerde, datosSerde));
```

- Por último, obtenemos el stream que ha transformado los valores y publicamos en destino

```
upperCasedStream.to( TOPIC_DESTINO, Produced.with(stringSerde, climaSerde));
upperCasedStream.print(Printed.<String,Clima>toSysOut().withLabel(TOPIC_DESTINO));
```

- Por último, generamos el kafkaStream y lo ejecutamos

```
KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),props);
Thread shutdownHook = new Thread(kafkaStreams::close);
Runtime.getRuntime().addShutdownHook(shutdownHook);
// Ejecución del hilo de kafka streams.
kafkaStreams.start();
```

- La clase final completa será la siguiente:

```
package com.curso.kafka.streams;

import java.util.Collections;
import java.util.Properties;

import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.Consumer;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Printed;
import org.apache.kafka.streams.kstream.Produced;

import com.curso.kafka.avro.model.Clima;
import com.curso.kafka.avro.model.Datos;

import io.confluent.kafka.serializers.AbstractKafkaSchemaSerDeConfig;
import io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde;

public class Stream03Topology {

    public static final String TOPIC_ORIGEN = "stream-topology-topic-origen";
    public static final String TOPIC_DESTINO = "stream-topology-topic-destino";
    public static final String TOPIC_DATOS = "stream-topology-topic-datos";

    public static void main(String[] args) throws InterruptedException {
        // Generación de propiedades de conexión
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "Stream03Topology");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");

        // Generación de serde para serializar/deserializar
        Serde<String> stringSerde = Serdes.String();
        Serde<Clima> climaSerde = new SpecificAvroSerde<>();
        Serde<Datos> datosSerde = new SpecificAvroSerde<>();
        climaSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
        "http://localhost:8081"), false);
        datosSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
        "http://localhost:8081"), false);
        // Construcción del StreamBuilder que genera la topología del procesador.
```

```

StreamsBuilder builder = new StreamsBuilder();
// Conexión del stream como consumidor desde topic-origen
KStream<String, Clima> simpleFirstStream =
    builder.stream(TOPIC_ORIGEN, Consumed.with(stringSerde, climaSerde));
// Procesador (Primer nodo hijo) que pasa el texto a mayúsculas
KStream<String, Clima> upperCasedStream =
    simpleFirstStream.mapValues( (value) -> {
        value.setNombre((value.getNombre()+"").toUpperCase());
        return value;
    });
// Segundo nodo

KStream<String, Datos> datosKStream = upperCasedStream.<String,Datos>map((key,value) -> new KeyValue<String,
Datos>(value.getNombre().toString(), value.getDatos()));
datosKStream.to(TOPIC_DATOS, Produced.<String, Datos>with(stringSerde, datosSerde));
datosKStream.print(Printed.<String,Datos>toSysOut().withLabel(TOPIC_DATOS));
// Conexión del stream al topic-destino como productor de mensajes
upperCasedStream.to( TOPIC_DESTINO,
    Produced.with(stringSerde, climaSerde));
// Log de desarrollo
upperCasedStream.print(Printed.<String,Clima>toSysOut().withLabel(TOPIC_DESTINO));

// Nodo que obtiene solo los datos

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),props);
Thread shutdownHook = new Thread(kafkaStreams::close);
Runtime.getRuntime().addShutdownHook(shutdownHook);
// Ejecución del hilo de kafka streams.
kafkaStreams.start();
}

}

```

15.16.3.2. Creación de topics

- En este caso, vamos a crear los topics desde Java.
- Para ello creamos una clase nueva llamada TopicAdmin.

```

package com.curso.kafka.streams;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.clients.admin.NewTopic;

public class TopicAdmin {

    public static void main(String[] args) {
        Map<String, Object> config = new HashMap<>();
        config.put("bootstrap.servers", "localhost:9092");
        AdminClient client = AdminClient.create(config);

        List<NewTopic> topics = new ArrayList<>();

        topics.add(new NewTopic(Stream03Topology.TOPIC_ORIGEN, 3, Short.parseShort("1")));
        topics.add(new NewTopic(Stream03Topology.TOPIC_DATOS, 3, Short.parseShort("1")));
        topics.add(new NewTopic(Stream03Topology.TOPIC_DESTINO, 3, Short.parseShort("1")));
        client.createTopics(topics);
        client.close();
    }
}

```

- Cuando lo ejecutemos, creará los topics con las librerías cliente.
- Ejecutamos con run as → Java Application

15.16.3.3. Prueba de ejecución

- Vamos a ejecutar el productor para que vaya mandando mensajes
- Luego ejecutamos el Stream y comprobamos como el topic destino y el topic datos están recibiendo la información correcta.

```

[stream-topology-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 287.0, "presion": 1011, "humedad": 93, "tempMin": 286.0, "tempMax": 289.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
[stream-topology-topic-datos]: MADRID, {"temp": 287.0, "presion": 1011, "humedad": 93, "tempMin": 286.0, "tempMax": 289.0}
[stream-topology-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 287.0, "presion": 1011, "humedad": 93, "tempMin": 286.0, "tempMax": 289.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion": "broken clouds", "icono": "04d"}]}
[stream-topology-topic-datos]: MADRID, {"temp": 287.0, "presion": 1011, "humedad": 93, "tempMin": 286.0, "tempMax": 289.0}

```

15.16.4. Stateful Stream

- En este caso, vamos a almacenar cierta información de estado
- El objetivo es que almacenemos en un store de tipo Key Value con persistencia en un topic de la información almacenada.
- La idea es dejar el servicio iniciado y almacenar la temperatura media total de todos los registros, pero para ver la funcionalidad vamos a sumar los valores
- Esta suma de valores comprobaremos como persiste en el tiempo, tiene estado y cuando paremos el servicio iniciaremos de nuevo y continuará donde lo dejó.

15.16.4.1. Productor

- Usaremos el mismo productor.
- Para ello, usaremos de nuevo OpenWeatherMap y el productor anterior, cambiando el topic destino.

```
ProducerRecord<String, Clima> record = new ProducerRecord<>(Stream04State.TOPIC_ORIGEN, CITY, clima);
```

15.16.4.2. Clase Stream

- Primero declaramos las variables de los topics a utilizar:

```
public class Stream04State {  
    public static final String TOPIC_ORIGEN = "stream-state-topic-origen";  
    public static final String TOPIC_DESTINO = "stream-state-topic-destino";  
    public static final String TOPIC_DATOS = "stream-state-topic-datos";  
    public static final String TOPIC_KSTREAM = "stream-state-topic-datos";  
    public static final String KV_STORE = "mediaTotalTemperaturas";
```

- Luego creamos las properties en el main:

```
public static void main(String[] args) throws InterruptedException {  
    Properties props = new Properties();  
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "ejemplo-basico-kafka-streams-1.0.0");  
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");  
    props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.FloatSerde.class.getName());  
    props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");
```

- Creamos los serdes a utilizar:

```

Serde<String> stringSerde = Serdes.String();
Serde<Clima> climaSerde = new SpecificAvroSerde<>();
Serde<Datos> datosSerde = new SpecificAvroSerde<>();
climaSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://localhost:8081"), false);
datosSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://localhost:8081"), false);

```

- Creamos un builder y vamos a crear una configuración para indicar las propiedades de un nuevo topic de tipo changeLog
- Esto lo usaremos con el almacenamiento

```

StreamsBuilder builder = new StreamsBuilder();
Map<String, String> changeLogConfigs = new HashMap<>();
changeLogConfigs.put("retention.ms", "172800000");
changeLogConfigs.put("retention.bytes", "10000000000");
changeLogConfigs.put("cleanup.policy", "compact,delete");

```

- Creamos el consumidor inicial para el primer procesador y mapeamos el valor del nombre

```

KStream<String, Clima> simpleFirstStream =
    builder.stream(TOPIC_ORIGEN, Consumed.with(stringSerde, climaSerde));
KStream<String, Clima> upperCasedStream =
    simpleFirstStream.mapValues( (value) -> {
        value.setNombre((value.getNombre() + "").toUpperCase());
        return value;
    });

```

- Creamos de nuevo el datosKStream para mapear clave y valor

```

KStream<String, Datos> datosKStream = upperCasedStream.<String, Datos>map((key, value) -> new KeyValue<String, Datos>(value
    .getNombre().toString(), value.getDatos()));
datosKStream.to(TOPIC_DATOS, Produced.<String, Datos>with(stringSerde, datosSerde));
datosKStream.print(Printed.<String, Datos>toSysOut().withLabel(TOPIC_DATOS));

```

- Ahora creamos un KGroupedStream que permita generar un valor a partir de una agregación de tipo reduce.

```

KGroupedStream<String, Float> temperaturasKStream = simpleFirstStream.mapValues((value)->value.getDatos().getTemp())
    .groupByKey();

```

- Ahora aplicamos el reduce, que permite realizar una materialización, es decir, almacenar el dato.
- Aprovechamos para persistir la información con la configuración que generamos anteriormente.

```
temperaturasKStream.reduce((tempAlmacenada,nuevaTemp)-> tempAlmacenada + nuevaTemp,  
    Materialized.<String,Float,KeyValueStore<Bytes, byte[]>>as(KV_STORE)  
    .withLoggingEnabled(changeLogConfigs));
```



- Para ser coherentes en el ejemplo, deberíamos usar la función $(tempAlmacenada + nuevaTemp)/2$
- Pero al ser datos actuales, no veríamos cambios.
- Enviamos el upperCasedStream al topic destino

```
upperCasedStream.to( TOPIC_DESTINO,Produced.with(stringSerde, climaSerde));  
upperCasedStream.print(Printed.<String,Clima>toSysOut().withLabel(TOPIC_DESTINO));
```

- Iniciamos el kafka Stream:

```
KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),props);  
Thread shutdownHook = new Thread(kafkaStreams::close);  
Runtime.getRuntime().addShutdownHook(shutdownHook);  
kafkaStreams.start();
```

- Por último, en el hilo principal, vamos a comprobar por medio del estado de KafkaStream si va todo bien
- Dejaremos 5 segundos para que se inicialice el stream y comience a dar resultados.
- En cuanto esté running que nos de el valor KV almacenado en el store

```
Thread.sleep(5000)  
while(!kafkaStreams.state().isRunningOrRebalancing()) {  
    System.out.println("Waiting 2s. State is : " + kafkaStreams.state());  
    Thread.sleep(2000);  
}  
while(kafkaStreams.state().isRunningOrRebalancing()) {  
    ReadOnlyKeyValueStore<String, Long> keyValueStore =  
        kafkaStreams.store(StoreQueryParameters.fromNameAndType(KV_STORE,  
        QueryableStoreTypes.keyValueStore()));  
    System.out.println("Media Temperatura Madrid:" + keyValueStore.get  
(ProductorSchemaRegistry.CITY));  
    Thread.sleep(2000);  
}
```

- La clase quedará como sigue:

```
package com.curso.kafka.streams;  
  
import java.util.Collections;
```

```

import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.common.utils.Bytes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StoreQueryParameters;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.KStreamConsumed;
import org.apache.kafka.streams.KGroupedStream;
import org.apache.kafka.streams.KStream;
import org.apache.kafka.streams.KStreamMaterialized;
import org.apache.kafka.streams.KStreamPrinted;
import org.apache.kafka.streams.KStreamProduced;
import org.apache.kafka.streams.KStreamReducer;
import org.apache.kafka.streams.state.KeyValueBytesStoreSupplier;
import org.apache.kafka.streams.state.KeyValueStore;
import org.apache.kafka.streams.state.QueryableStoreTypes;
import org.apache.kafka.streams.state.ReadOnlyKeyValueStore;
import org.apache.kafka.streams.state.StoreBuilder;
import org.apache.kafka.streams.state.Stores;

import com.curso.kafka.avro.model.Clima;
import com.curso.kafka.avro.model.Datos;

import io.confluent.kafka.serializers.AbstractKafkaSchemaSerDeConfig;
import io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde;

public class Stream04State {

    public static final String TOPIC_ORIGEN = "stream-state-topic-origen";
    public static final String TOPIC_DESTINO = "stream-state-topic-destino";
    public static final String TOPIC_DATOS = "stream-state-topic-datos";
    public static final String TOPIC_KSTREAM = "stream-state-topic-datos";
    public static final String KV_STORE = "mediaTotalTemperaturas";

    public static void main(String[] args) throws InterruptedException {
        // Generación de propiedades de conexión
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "ejemplo-basico-kafka-streams-1.0.0");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.FloatSerde.class.getName());
        props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");

        // Generación de serde para serializar/deserializar
        Serde<String> stringSerde = Serdes.String();
        Serde<Clima> climaSerde = new SpecificAvroSerde<>();
        Serde<Datos> datosSerde = new SpecificAvroSerde<>();
        climaSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
        "http://localhost:8081"), false);
        datosSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
        "http://localhost:8081"), false);
        // Construcción del StreamBuilder que genera la topología del procesador.
        StreamsBuilder builder = new StreamsBuilder();
        // Construimos un changelog para que la caché vaya a un topic en kafka:
        //KeyValueBytesStoreSupplier store = Stores.inMemoryKeyValueStore(KV_STORE);
        //StoreBuilder<KeyValueStore<String, Float>> storeBuilder = Stores.keyValueStoreBuilder(store, Serdes.String(),
        Serdes.Float());
        //builder.addStateStore(storeBuilder);
        // Construcción de caché en un topic
        Map<String, String> changeLogConfigs = new HashMap<>();
        changeLogConfigs.put("retention.ms", "172800000");
        changeLogConfigs.put("retention.bytes", "10000000000");
    }
}

```

```

changeLogConfigs.put("cleanup.policy", "compact,delete");
//storeBuilder.withLoggingEnabled(changeLogConfigs);
//;

// Conexión del stream como consumidor desde topic-origen
KStream<String, Clima> simpleFirstStream =
    builder.stream(TOPIC_ORIGEN, Consumed.with(stringSerde, climaSerde));
// Procesador (Primer nodo hijo) que pasa el texto a mayúsculas
KStream<String, Clima> upperCasedStream =
    simpleFirstStream.mapValues( (value) -> {
        value.setNombre((value.getNombre()+"").toUpperCase());
        return value;
    });
// Segundo nodo

KStream<String, Datos> datosKStream = upperCasedStream.<String,Datos>map((key,value) -> new KeyValue<String, Datos>(value.getNombre().toString(), value.getDatos()));
    datosKStream.to(TOPIC_DATOS, Produced.<String, Datos>with(stringSerde, datosSerde));
    datosKStream.print(Printed.<String, Datos>toSysOut().withLabel(TOPIC_DATOS));
// Conexión del stream al topic-destino como productor de mensajes
KGroupedStream<String,Float> temperaturasKStream = simpleFirstStream.mapValues((value)->value.getDatos().getTemp());
).groupByKey();
temperaturasKStream.reduce((tempAlmacenada,nuevaTemp)-> tempAlmacenada + nuevaTemp,
Materialized.<String,Float>,KeyValueStore<Bytes, byte[]>as(KV_STORE).withLoggingEnabled(changeLogConfigs));

upperCasedStream.to( TOPIC_DESTINO,
    Produced.with(stringSerde, climaSerde));
// Log de desarrollo
upperCasedStream.print(Printed.<String,Clima>toSysOut().withLabel(TOPIC_DESTINO));

// Nodo que obtiene solo los datos

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(),props);

Thread shutdownHook = new Thread(kafkaStreams::close);
Runtime.getRuntime().addShutdownHook(shutdownHook);
// Ejecución del hilo de kafka streams.
kafkaStreams.start();
Thread.sleep(5000);
while(!kafkaStreams.state().isRunningOrRebalancing()) {
    System.out.println("Waiting 2s. State is : " + kafkaStreams.state());
    Thread.sleep(2000);
}
while(kafkaStreams.state().isRunningOrRebalancing()) {
    ReadOnlyKeyValueStore<String, Long> keyValueStore =
        kafkaStreams.store(StoreQueryParameters.fromNameAndType(KV_STORE, QueryableStoreTypes.keyValueStore()));
    System.out.println("Media Temperatura Madrid:" + keyValueStore.get(ProducerSchemaRegistry.CITY));
    Thread.sleep(2000);
}
}
}

```

15.16.4.3. Creación de los topics

- Volvemos a crear los topics con la nueva clase

```

package com.curso.kafka.streams;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.clients.admin.NewTopic;

public class TopicAdmin {

    public static void main(String[] args) {
        Map<String, Object> config = new HashMap<>();
        config.put("bootstrap.servers", "localhost:9092");
        AdminClient client = AdminClient.create(config);

        List<NewTopic> topics = new ArrayList<>();

        topics.add(new NewTopic(Stream04State.TOPIC_ORIGEN, 3, Short.parseShort("1")));
        topics.add(new NewTopic(Stream04State.TOPIC_DATOS, 3, Short.parseShort("1")));
        topics.add(new NewTopic(Stream04State.TOPIC_DESTINO, 3, Short.parseShort("1")));
        topics.add(new NewTopic(Stream04State.TOPIC_KSTREAM, 3, Short.parseShort("1")));
        client.createTopics(topics);
        client.close();
    }
}

```

- Ejecutamos la aplicación java para crear los topics.

15.16.4.4. Prueba de ejecución

- Vamos a ejecutar primero el productor para que envíe mensajes

```

{"coord":{"lon":-3.7,"lat":40.42}, "weather":[{"id":803,"main":"Clouds","description":"broken clouds","icon":"04d"}], "base":"stations", "main":{"temp":288.46,"feels_like":286.22,"temp_min":287.04,"temp_max":289.82,"pressure":1012,"humidity":87}, "visibility":9000, "wind":{"speed":4.6,"deg":180}, "clouds":{"all":75}, "dt":1603281433, "sys": {"type":1, "id":6443, "country": "ES", "sunrise":1603261940, "sunset":1603301165}, "timezone":7200, "id":3117735, "name": "Madrid", "cod":200}
{"coord":{"lon":-3.7,"lat":40.42}, "weather":[{"id":803,"main":"Clouds","description":"broken clouds","icon":"04d"}], "base":"stations", "main":{"temp":288.46,"feels_like":286.22,"temp_min":287.04,"temp_max":289.82,"pressure":1012,"humidity":87}, "visibility":9000, "wind":{"speed":4.6,"deg":180}, "clouds":{"all":75}, "dt":1603281433, "sys": {"type":1, "id":6443, "country": "ES", "sunrise":1603261940, "sunset":1603301165}, "timezone":7200, "id":3117735, "name": "Madrid", "cod":200}
{"coord":{"lon":-3.7,"lat":40.42}, "weather":[{"id":803,"main":"Clouds","description":"broken clouds","icon":"04d"}], "base":"stations", "main":{"temp":288.46,"feels_like":286.22,"temp_min":287.04,"temp_max":289.82,"pressure":1012,"humidity":87}, "visibility":9000, "wind":{"speed":4.6,"deg":180}, "clouds":{"all":75}, "dt":1603281433, "sys": {"type":1, "id":6443, "country": "ES", "sunrise":1603261940, "sunset":1603301165}, "timezone":7200, "id":3117735, "name": "Madrid", "cod":200}
...

```

- Luego ejecutamos el stream
- Tras dejarlo un rato, veremos como comienza a enviar datos de los valores incrementales.

```
[stream-state-topic-datos]: MADRID, {"temp": 288.0, "presion": 1012, "humedad": 87, "tempMin": 287.0, "tempMax": 289.0}
[stream-state-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 288.0, "presion": 1012,
"humedad": 87, "tempMin": 287.0, "tempMax": 289.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion":
"broken clouds", "icono": "04d"}]}
Media Temperatura Madrid:18094.0
[stream-state-topic-datos]: MADRID, {"temp": 288.0, "presion": 1012, "humedad": 87, "tempMin": 287.0, "tempMax": 289.0}
[stream-state-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 288.0, "presion": 1012,
"humedad": 87, "tempMin": 287.0, "tempMax": 289.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion":
"broken clouds", "icono": "04d"}]}
[stream-state-topic-datos]: MADRID, {"temp": 288.0, "presion": 1012, "humedad": 87, "tempMin": 287.0, "tempMax": 289.0}
[stream-state-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 288.0, "presion": 1012,
"humedad": 87, "tempMin": 287.0, "tempMax": 289.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion":
"broken clouds", "icono": "04d"}]}
Media Temperatura Madrid:18670.0
```

- Paramos el stream y lo volvemos a encender.
- Podemos comprobar como los valores del almacenamiento, a pesar de haber caido, se han recuperado continuando su valor.

```
[stream-state-topic-datos]: MADRID, {"temp": 288.0, "presion": 1012, "humedad": 87, "tempMin": 287.0, "tempMax": 290.0}
[stream-state-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 288.0, "presion": 1012,
"humedad": 87, "tempMin": 287.0, "tempMax": 290.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion":
"broken clouds", "icono": "04d"}]}
Media Temperatura Madrid:47470.0
[stream-state-topic-datos]: MADRID, {"temp": 288.0, "presion": 1012, "humedad": 87, "tempMin": 287.0, "tempMax": 290.0}
[stream-state-topic-destino]: madrid, {"id": 3117735, "nombre": "MADRID", "datos": {"temp": 288.0, "presion": 1012,
"humedad": 87, "tempMin": 287.0, "tempMax": 290.0}, "detalles": [{"id": 803, "principal": "Clouds", "descripcion":
"broken clouds", "icono": "04d"}]}
[stream-state-topic-datos]: MADRID,
```

- Paramos los procesos java.

15.16.5. Uso de Join

- En este caso, vamos a usar la operación join para fusionar dos streams en un solo resultado.
- Para ello, crearemos un nuevo esquema avro, un productor de películas, otro de valoraciones y un joiner que indique como integrar ambos valores.

15.16.6. Avro

- Primero creamos los siguientes esquemas:

Pelicula.avsc

```
{  
  "namespace": "com.curso.kafka.avro.model",  
  "type": "record",  
  "name": "Pelicula",  
  "fields": [  
    {"name": "id", "type": "long"},  
    {"name": "titulo", "type": "string"},  
    {"name": "year_pub", "type": "int"}  
  ]  
}
```

Valoracion.avsc

```
{  
  "namespace": "com.curso.kafka.avro.model",  
  "type": "record",  
  "name": "Valoracion",  
  "fields": [  
    {"name": "id", "type": "long"},  
    {"name": "valoracion", "type": "double"}  
  ]  
}
```

PeliculaValorada.avsc

```
{  
  "namespace": "com.curso.kafka.avro.model",  
  "type": "record",  
  "name": "PeliculaValorada",  
  "fields": [  
    {"name": "id", "type": "long"},  
    {"name": "titulo", "type": "string"},  
    {"name": "year_pub", "type": "int"},  
    {"name": "valoracion", "type": "double"}  
  ]  
}
```

- Lo dejamos en el directorio src/main/avro para poder compilarlo.
- Ejecutamos Run As → Maven → Generate-sources pulsando botón derecho en el proyecto avro-tool.
- Luego generamos el artefacto para enviarlo a todos los proyectos dependientes:
 - Run As → Maven → install

15.16.6.1. Generación del productor de películas

- De forma sencilla, generaremos películas con ids sencillos incrementales.
- Para ello, creamos el siguiente productor:

```
package com.curso.kafka.streams;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import com.curso.kafka.avro.model.Pelicula;

import io.confluent.kafka.serializers.KafkaAvroSerializer;
import io.confluent.kafka.serializers.KafkaAvroSerializerConfig;

public class ProductorPeliculas {

    public static final String CITY = "madrid";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());
        properties.put(KafkaAvroSerializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");

        KafkaProducer<String,Pelicula> producer = new KafkaProducer<>(properties);

        Thread shutdownHook = new Thread(producer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);
        int i = 0;
        while(true) {
            i++;
            Pelicula pelicula = Pelicula.newBuilder()
                .setId(i)
                .setTitle("Pelicula "+i)
                .setYearPub(i+1990).build();

            ProducerRecord<String, Pelicula> record = new ProducerRecord<>(
                Stream05Join.TOPIC_PELICULAS,
                pelicula.getTitle().toString(),
                pelicula);
            producer.send(record);
            Thread.sleep(500);
        }
    }
}
```

- Por otro lado, vamos a usar otro productor de valoraciones.

```

package com.curso.kafka.streams;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringSerializer;

import com.curso.kafka.avro.model.Clima;
import com.curso.kafka.avro.model.Pelicula;
import com.curso.kafka.avro.model.Valoracion;

import io.confluent.kafka.serializers.KafkaAvroSerializer;
import io.confluent.kafka.serializers.KafkaAvroSerializerConfig;

public class ProductorValoraciones {

    public static final String CITY = "madrid";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class.getName());
        properties.put(KafkaAvroSerializerConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");

        KafkaProducer<String,Valoracion> producer = new KafkaProducer<>(properties);

        Thread shutdownHook = new Thread(producer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);
        int i = 0;
        while(true) {
            i++;
            Valoracion valoracion = Valoracion.newBuilder()
                .setId(i)
                .setValoracion(Math.random()).build();

            ProducerRecord<String, Valoracion> record = new ProducerRecord<>(
                Stream05Join.TOPIC_VALORACION,
                "Pelicula "+i,
                valoracion);
            producer.send(record);
            Thread.sleep(500);
        }
    }
}

```

15.16.6.2. Creación de stream

- Para crearlo, vamos a necesitar una clase de apoyo que indique como se fusionan los registros.
- Creamos una clase llamada PeliculasValoracionJoiner que reciba una valoracion, una pelicula y devuelva una pelicula PeliculaValorada

```

package com.curso.kafka.streams;

import org.apache.kafka.streams.kstream.ValueJoiner;

import com.curso.kafka.avro.model.Pelicula;
import com.curso.kafka.avro.model.PeliculaValorada;
import com.curso.kafka.avro.model.Valoracion;

public class PeliculasValoracionJoiner implements ValueJoiner<Valoracion, Pelicula, PeliculaValorada>{

    @Override
    public PeliculaValorada apply(Valoracion valoracion, Pelicula pelicula) {

        return PeliculaValorada.newBuilder()
            .setId(pelicula.getId())
            .setTitulo(pelicula.getTitulo())
            .setValoracion(valoracion.getValoracion())
            .setYearPub(pelicula.getYearPub()).build();
    }

}

```

- Ahora creamos el stream:
- Primero creamos la clase Stream05join e indicamos los topics necesarios
- Aprovechamos a crear un countdownlatch para tener una cuenta atrás de un segundo antes de pararla

```

public static final String TOPIC_PELICULAS = "stream-topic-peliculas";
public static final String TOPIC_PELICULAS_REKEYED = "stream-topic-rekeyed";
public static final String TOPIC_VALORACION = "stream-topic-valoracion";
public static final String TOPIC_PELICULAS_VALORADAS = "stream-topic-peliculas-valoradas";
private static final CountDownLatch latch = new CountDownLatch(1);

```

- Configuramos el stream.

```

public static void main(String[] args) throws InterruptedException {
    // Generación de propiedades de conexión
    Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "Stream05join");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
    props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
    props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, SpecificAvroSerde.class.getName());
    props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");

```

- Creamos el serde asociado a nuestro destino

```

Serde<PeliculaValorada> pVSerde = new SpecificAvroSerde<>();
pVSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://localhost:8081"), false);

```

- Creamos el StreamBuilder

```
StreamsBuilder builder = new StreamsBuilder();
```

- Generamos nuestro primer stream que obtiene los datos del topic, remapea los datos para poder aprovecharlos en la fusión.

```
KStream<String, Pelicula> peliculasStream = builder.<String,Pelicula>stream(TOPIC_PELICULAS);
peliculasStream.map((key,pelicula) -> new KeyValue<>(String.valueOf(pelicula.getId()), pelicula));
peliculasStream.to(TOPIC_PELICULAS_REKEYED);
peliculasStream.print(Printed.<String, Pelicula>toSysOut().withLabel(TOPIC_PELICULAS_REKEYED));
```

- Conectamos al topic remapeado y obtenemos un KTable. Solo queremos los datos de actualización

```
KTable<String, Pelicula> peliculas = builder.table(TOPIC_PELICULAS_REKEYED);
```



- Podríamos haber usado peliculasStream para obtener un KTable, sin embargo, crearía implicitamente un topic.
 - Es mejor explicitarlo para mejorar la optimización.
- Obtenemos un KStream con las valoraciones desde el builder (dos entradas distintas)

```
KStream<String, Valoracion> valoraciones = builder.<String,Valoracion>stream
(TOPIC_VALORACION);
```

- Ahora realizamos el join y guardamos los resultados

```
KStream<String, PeliculaValorada> peliculasValoradas = valoraciones.join(peliculas, new PeliculasValoracionJoiner());
peliculasValoradas.to(TOPIC_PELICULAS_VALORADAS,Produced.<String, PeliculaValorada>with(Serdes.String(),pVSerde));
peliculasValoradas.print(Printed.<String, PeliculaValorada>toSysOut().withLabel(TOPIC_PELICULAS_VALORADAS));
```

- Por último iniciamos, y esta vez usamos el countDownLatch para finalizar los hilos.

```

final KafkaStreams streams = new KafkaStreams(builder.build(),props);
Runtime.getRuntime().addShutdownHook(new Thread("stream-shutdown-hook") {

    @Override
    public void run() {
        streams.close();
        latch.countDown();
    }
});
try {
    streams.start();
    latch.await();
} catch (Throwable e) {
    System.exit(1);
}
System.exit(0);

```

- El resultado de la clase será la siguiente:

```

package com.curso.kafka.streams;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;

import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.clients.admin.NewTopic;
import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Printed;
import org.apache.kafka.streams.kstream.Produced;

import com.curso.kafka.avro.model.Pelicula;
import com.curso.kafka.avro.model.PeliculaValorada;
import com.curso.kafka.avro.model.Valoracion;

import io.confluent.kafka.serializers.AbstractKafkaSchemaSerDeConfig;
import io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde;

public class Stream05Join {

    public static final String TOPIC_PELICULAS = "stream-topic-peliculas";
    public static final String TOPIC_PELICULAS_REKEYED = "stream-topic-rekeyed";
    public static final String TOPIC_VALORACION = "stream-topic-valoracion";
    public static final String TOPIC_PELICULAS_VALORADAS = "stream-topic-peliculas-valoradas";
    private static final CountDownLatch latch = new CountDownLatch(1);
}

```

```

public static void main(String[] args) throws InterruptedException {
    // Generación de propiedades de conexión
    Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "Stream05join");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9091,localhost:9092,localhost:9093");
    props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
    props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, SpecificAvroSerde.class.getName());
    props.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8081");

    // Generación de serde para serializar/deserializar

    Serde<PeliculaValorada> pVSerde = new SpecificAvroSerde<>();
    pVSerde.configure(Collections.singletonMap(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
    "http://localhost:8081"), false);

    // Construcción del StreamBuilder que genera la topología del procesador.
    StreamsBuilder builder = new StreamsBuilder();
    KStream<String, Pelicula> peliculasStream = builder.<String, Pelicula>stream(TOPIC_PELICULAS);
    peliculasStream.map((key, pelicula) -> new KeyValue<>(String.valueOf(pelicula.getId()), pelicula));
    peliculasStream.to(TOPIC_PELICULAS_REKEYED);
    peliculasStream.print(Printed.<String, Pelicula>toSysOut().withLabel(TOPIC_PELICULAS_REKEYED));
    KTable<String, Pelicula> peliculas = builder.table(TOPIC_PELICULAS_REKEYED);

    KStream<String, Valoracion> valoraciones = builder.<String, Valoracion>stream(TOPIC_VALORACION);
    KStream<String, PeliculaValorada> peliculasValoradas = valoraciones.join(peliculas, new
    PeliculasValoracionJoiner());
    peliculasValoradas.to(TOPIC_PELICULAS_VALORADAS, Produced.<String, PeliculaValorada>with(Serdes.String(), pVSerde)
    );
    peliculasValoradas.print(Printed.<String, PeliculaValorada>toSysOut().withLabel(TOPIC_PELICULAS_VALORADAS));

    final KafkaStreams streams = new KafkaStreams(builder.build(), props);
    Runtime.getRuntime().addShutdownHook(new Thread("stream-shutdown-hook") {

        @Override
        public void run() {
            streams.close();
            latch.countDown();
        }

    });
    try {
        streams.start();
        latch.await();
    } catch (Throwable e) {
        System.exit(1);
    }
    System.exit(0);
}
}

```

15.16.6.3. Creación de topics.

- Por último, creamos los topics en un nuevo main

```

package com.curso.kafka.streams;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.clients.admin.NewTopic;

public class TopicAdmin {

    public static void main(String[] args) {
        Map<String, Object> config = new HashMap<>();
        config.put("bootstrap.servers", "localhost:9092");
        AdminClient client = AdminClient.create(config);

        List<NewTopic> topics = new ArrayList<>();

        topics.add(new NewTopic(Stream05join.TOPIC_PELICULAS, 3, Short.parseShort("1")));
        topics.add(new NewTopic(Stream05join.TOPIC_PELICULAS_REKEYED, 3, Short.parseShort("1")));
        topics.add(new NewTopic(Stream05join.TOPIC_PELICULAS_VALORADAS, 3, Short.parseShort("1")));
        topics.add(new NewTopic(Stream05join.TOPIC_VALORACION, 3, Short.parseShort("1")));

        client.createTopics(topics);
        client.close();
    }
}

```

- Ejecutamos el main para crear los topics.
- Run As → Java Application

15.16.6.4. Prueba de ejecución

- Por último, ejecutamos primero el productor de películas:
- Ahora ejecutamos el stream
- Podemos ver como está consumiendo las películas, pero solo está haciendo el rekey:

```

[stream-topic-rekeyed]: Pelicula 142, {"id": 142, "titulo": "Pelicula 142", "year_pub": 2132}
[stream-topic-rekeyed]: Pelicula 143, {"id": 143, "titulo": "Pelicula 143", "year_pub": 2133}
[stream-topic-rekeyed]: Pelicula 144, {"id": 144, "titulo": "Pelicula 144", "year_pub": 2134}
[stream-topic-rekeyed]: Pelicula 145, {"id": 145, "titulo": "Pelicula 145", "year_pub": 2135}

```

- Ahora ejecutamos el productor de valoraciones y vemos como inyecta los valores en películas valoradas.

```
stream-topic-peliculas-valoradas]: Pelicula 17, {"id": 17, "titulo": "Pelicula 17", "year_pub": 2007, "valoracion": 0.7739762343828737}
[stream-topic-rekeyed]: Pelicula 242, {"id": 242, "titulo": "Pelicula 242", "year_pub": 2232}
[stream-topic-peliculas-valoradas]: Pelicula 18, {"id": 18, "titulo": "Pelicula 18", "year_pub": 2008, "valoracion": 0.15049030700375288}
[stream-topic-rekeyed]: Pelicula 243, {"id": 243, "titulo": "Pelicula 243", "year_pub": 2233}
[stream-topic-peliculas-valoradas]: Pelicula 19, {"id": 19, "titulo": "Pelicula 19", "year_pub": 2009, "valoracion": 0.33440786546691537}
[stream-topic-rekeyed]: Pelicula 244, {"id": 244, "titulo": "Pelicula 244", "year_pub": 2234}
```

Capítulo 16. Kafka Connect

- Se trata de un framework para envío de flujos de datos entre Kafka y otros sistemas
- Open Source y forma parte de la distribución de Apache Kafka
- Simple, escalable y seguro



- Permite streams de bases de datos SQL en kafka
- Permite streams de kafka topics en HDFS para procesado en streams
- Permite streams de topics en ElasticSearch para indexados
- Básicamente, usa la misma filosofía de productor y consumidor para los clientes de Kafka Connect.
- Se trata de clientes preconstruidos para realizar conexiones sencillas
- Pueden ser extendidas por programadores
- Los conectores son Jobs lógicos que gestionan la ingesta de datos entre kafka y otro sistema
 - **Connector sources:** Leen datos de una fuente externa y lo inyectan en kafka (Productor)
 - **Connector Sinks:** Escriben datos de kafka en un sistema de datos externo (Consumidor)

16.1. Tipos

- La versión Opensource de Confluent provee de los siguientes conectores:
 - JDBC: Envío de filas nuevas o modificadas en mensajes kafka
 - HDFS: Envío de kafka a Hadoop Distributed File System. Integrado con Hive, y soporte de particiones
 - Elasticsearch
 - AWS S3
 - FileStream: Obtención del fin de fichero como un kafka message (logs), o viceversa, mensajes kafka a un fichero
- La versión enterprise
 - Replicator

16.2. Modos de ejecución

- Kafka Connect posee dos modos de ejecución
 - Standalone:

- Proceso como un solo worker en una máquina
 - Uso para pruebas o procesos que no se van a distribuir
- Distribuido:
- Multiples procesos worker en una o más máquinas
 - Uso para tolerancia a fallos y escalabilidad

Capítulo 17. Seguridad en Kafka

- **Kafka** ofrece distintas opciones de seguridad, son bastante interesantes si queremos aplicarlas a nuestro clúster.
- Las distintas funcionalidades que podemos tener son:
 - Autenticación mediante **SSL**
 - Autenticación de conexiones entre los **Brokers** y **Zookeeper**
 - Autorización de lecturas y/o escrituras para los clientes
 - Soporte de servicios externos de autenticación
 - Cifrado de los datos mediante **SSL** para la comunicación.

17.1. Cómo funciona la seguridad

- La seguridad en las comunicaciones es algo vital.
- Los primeros ejemplos documentados de comunicación cifrada, datan de la época del Imperio Romano, donde Julio César usaba un método simple para comunicarse con sus generales, desplazaban 13 veces en el alfabeto la letra leída, de tal manera que el mensaje "Ataqueren" se convertía en "MFMCGQZ".
- Cuando los generales recibían el mensaje realizaban la operación inversa (es decir, se movían 13 veces en el alfabeto en sentido inverso).
- Este es un ejemplo de **cifrado con clave simétrica**, en el que para cifrar y descifrar el mensaje, hacemos uso de una misma clave.
- De hecho, las claves simétricas eran la única opción que conocíamos hasta hace relativamente poco.
- Un ejemplo claro de esto fue la máquina enigma, utilizada durante la segunda guerra mundial.
- De forma esquemática, podríamos decir que el proceso es como se muestra en la imagen:



- A día de hoy seguimos contando con cifrados de clave simétrica, como AES, DES, TripleDES o Blowfish.
 - Su gran ventaja, son algoritmos muy veloces, ideales para el cifrado de gran cantidad de datos.
 - Su gran inconveniente, las dos partes deben poseer la clave, lo que supone un riesgo a la seguridad, ya que esta clave debe ser distribuida y en este momento podría ser interceptada y robada, comprometiendo la seguridad. (¿Os acordáis de las típicas películas de espías, que iban con un maletín esposado?)



- Frente a la criptografía de clave simétrica, tenemos la **criptografía de clave asimétrica**. Dicha técnica de cifrado no consiste en una única clave, si no en una pareja de ellas:
 - Clave privada, que jamás es distribuida, y cuyo propietario debe custodiar.
 - Clave pública, conocida por todos los usuarios y distribuida sin problemas.
- Este sistema es mucho mas seguro, ya que suprime la necesidad del envío de una clave que da poder total y absoluto. (!Se acabaron los espías con maletines!)
- Por otro lado, tiene un importante inconveniente, es mucho más lento (la operación es más compleja).
- Para generar la pareja de claves se debe:
 - Escoger dos números primos, **p** y **q**
 - Calcular **N**, que es el resultado de multiplicar **p x q**.
 - Escoger un número **e**, que debe ser primo a **(p-1) x (q-1)** (que no tengan divisores comunes entre ellos)
 - Calcular un número **d**, que es el número que cumple que **(e x d) mod p-1 x (q-1) = 1**

- La clave pública es **(N,e)**
- La clave privada es **(d)**
- La información **I** se cifra con **$I^e \text{ mod } N$**
- La información cifrada **C** se descifra con **$C^d \text{ mod } N$**
- Un ejemplo sería el siguiente
 - Escojo **p=3** y **q=7**, ambos números primos
 - Calculo **N=3 x 7=21**
 - Busco número primo relativo a **(p-1) x (q-1)=2 x 6=12**, cojo **e=5** que es primo relativo a **12**
 - Busco **d** en **(e x d) mod p-1 x (q-1 = 1**, nos sale **d=5** (ya que **25 mod 12 = 1**)
 - Clave pública **(N,e)** es **(21,5)**
 - Clave privada **d** es **5**
 - Voy a cifrar un **M** que vale **4**: **$C=M^e \text{ mod } N$** que es **$4^5 \text{ mod } 21 = 1024 \text{ mod } 21=16$**
 - Voy a descifrar **C** que es **16**: **$D=C^d \text{ mod } N$** que es **$16^5 \text{ mod } 21 = 1048576 \text{ mod } 21 = 4$**
- Ahora que entendemos cómo se generan las claves de cifrado asimétricas, vamos qué opciones tenemos para comunicar dos partes:
 - Una de las partes (A), posee una pareja de claves pública/privada.
 - La otra parte (B), no posee ninguna:
 - A puede usar su clave privada para cifrar la información y enviarla a **B**.
 - Cualquiera puede usar la clave pública de **A** para abrir el mensaje, con lo que no existe confidencialidad, pero **B** se asegura de que el mensaje fue emitido por **A**, puesto que tuvo que cifrarlo con su clave privada. * **Existe *autenticidad** de emisor.
 - **B** puede cifrar mensajes con la clave pública de **A** y enviárselos a **A**.
 - Nadie más puede ver este mensaje, por lo que aseguras la **confidencialidad**, ya que para descifrarlo sólo puede usarse la clave privada de **A**.
 - Sin embargo, cualquier emisor podría suplantar a **B** (tan sólo tiene que usar la clave pública de **A**), **A** no tiene la certeza de que su mensaje provenga de **B**.
 - Las dos partes tienen una pareja de claves pública/privada:
 - Ahora, podemos asegurar la **confidencialidad** y el **autenticidad**.
 - Cuando se van a intercambiar mensajes, se cifran dos veces.
 - Si **A** quiere enviar un mensaje a **B**, lo cifra primero con su clave privada, y luego con la clave pública de **B**.
 - Para que **B** lo pueda descifrar usa primero la clave pública de **A** (aseguramos emisor), y luego su propia clave privada (aseguramos confidencialidad).
 - De manera análoga, **B** puede enviar mensajes a **A** cifrandolos primero con su clave privada y luego con la clave pública de **A**.
 - Esto, es tremadamente costoso, con lo que por lo general, se usa esta comunicación para intercambiar de forma segura una clave simétrica, y tras intercambiar esta clave, el resto de la

comunicación se cifra mediante ella.

- Al intercambiar la clave como hemos dicho, nos aseguramos que nadie nos la intercepte.
- Un ejemplo de una comunicación usando sólo una pareja de claves.



17.2. Certificados, Keystores y Trustores

- En el tema anterior vimos qué eran las claves de cifrado, y cómo estaban compuestas de dos partes, la clave pública y la clave privada.
- Con la intención de almacenar las claves privadas propias, y las claves públicas de otros, surge el **keystore**.
 - Este no es nada más que un almacén de claves.
 - Un fichero de nuestro disco que va a contener un conjunto de claves.
 - Llegados a este punto, podíamos asegurarnos de que la comunicación con otro sistema era segura, pero ¿quién nos certifica que quien está usando unas claves, es quien dice ser?.
 - Por ejemplo, yo me conecto a la web de la **AEAT**, y voy a hacer mi declaración.
 - Si un tercero lo suplantara y tuviera su propio conjunto de claves, ¿habría manera de saber si mi comunicación segura es realmente con dicho dominio, o no?
 - Para cubrir este supuesto surgen los **certificados**.
- Un **certificado** es una pareja de claves (pública/privada) cuya parte pública ha sido **firma**da por una **entidad certificadora**, es decir, se han asegurado de que el dueño de dicho certificado es quien dice ser.
- Ahora si nos roban nuestro dominio y nos intentan suplantar, podrán tener comunicación segura pero les saltará una advertencia, de que no han podido comprobar la autenticidad de dicho dominio.
- Pero **who watches the watchmen?**. ¿quién me asegura que la **entidad certificadora** ha hecho su trabajo?.
- Existen varias entidades certificadoras, nosotros podemos decidir en cuál confiar y en cuál no.

- Para aquellas entidades en las que confiemos, guardaremos su clave pública de cifrado en nuestro **Trustore**, que es otro almacén, donde almaceno sólo aquellas entidades en las que confío.
- Ahora, si me comunico con un nuevo sistema, puedo confiar en quién dice ser porque posee una firma que puedo comprobar.
- Brevemente, para conseguir un certificado digital, necesitamos:
 - Generar una pareja de claves pública/privada
 - Emitir una petición de firmado de certificado (que basicamente es la clave pública)
 - Enviar dicha petición a la entidad certificadora
 - La entidad verificadora se asegurará de que eres quien dices ser (nombres, direcciones, se pondrá en contacto, etc...)
 - La entidad certificadora firma el certificado, y nos lo devuelve
 - Ya puedo comunicarme sin problemas con todos los demás sistemas que confíen en dicha entidad certificadora.

17.3. Seguridad en clientes

- La implementación de seguridad en productores y consumidores se realiza a partir del certificado cliente.
- Este certificado cliente de confianza generado con los certificados del servidor permite autenticar y autorizar un cliente
- La configuración que se debe definir es la siguiente:
- Primero definimos el protocolo SSL como activo

```
props.put("security.protocol", "SSL");
```

```
props.put("ssl.truststore.location", trustStoreLocation);
props.put("ssl.truststore.password", trustStorePassword);
```

- Si no está habilitada la autenticación de clientes, esta segunda parte no es necesaria.

```
props.put("ssl.key.password", keyStorePassword);
props.put("ssl.keystore.password", keyStorePassword);
props.put("ssl.keystore.location", keyStoreLocation);
```

- Otras opciones que podemos configurar:
 - **ssl.provider**: El nombre del proveedor de seguridad usado para conexiones SSL
 - **ssl.cipher.suites**: Un conjunto de cifrados usados para negociación
 - **ssl.enabled.protocols**: Debe poseer alguno de los protocolos del broker

(TLSv1.2,TLSv1.1,TLSv1)

- **ssl.truststore.type:** JKS
- **ssl.keystore.type:** JKS

17.3.1. SASL

- Kafka usa JAAS (Java Authentication and Authorization Service) para configurar SASL
- Para configurar JAAS, debemos generar un fichero estático

Contenido del fichero /etc/kafka/kafka_client_jaas.conf

```
KafkaClient {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    keyTab="/etc/security/keytabs/kafka_client.keytab"  
    principal="kafka-client@kafka.local";  
};
```

- Para poder cargar JAAS, en la ejecución de java, debemos indicar la siguiente directiva:

```
-Djava.security.auth.login.config=/etc/kafka/kafka_client_jaas.conf
```

17.4. Lab: Aplicando seguridad en Kafka

- Para poder tener una comunicación segura con nuestro **brokers**, vamos a cifrar la comunicación mediante el uso de claves de cifrado asimétricas.
- Para ello tendremos que crear
 - Una pareja de claves pública/privada, que tendremos en nuestro **keystore**
 - Una entidad certificadora, que deberemos añadir a nuestro **trustore**
 - Un certificado firmado (por nuestra entidad certificadora)
- Para ello, usaremos la herramienta **keytool** que viene disponible en el propio **JDK**.

17.4.1. Creación del keystore y la pareja de claves pública/privada

Llamaremos a **keytool** (desde nuestro \$HOME), indicando dónde va a crear ese **keystore**:

- Indicaremos que genere una clave con **-genkey**, y hemos especificado la validez de dicha clave (360 días).
- El **alias** con el que almacenaremos esta clave en nuestro **keystore** es **kafkakafka**
- Al ejecutar dicho comando, nos van a pedir distintos datos, (para nuestro ejemplo usaremos la clave **usuario**)

```
[kafka@kafka-server ~]$ keytool -keystore server.keystore.jks -alias kafka -validity 360 -genkey
Enter keystore password:
Re-enter new password:
What is your first and last name?
[Unknown]: Curso Kafka
What is the name of your organizational unit?
[Unknown]: N/A
What is the name of your organization?
[Unknown]: Curso
What is the name of your City or Locality?
[Unknown]: Madrid
What is the name of your State or Province?
[Unknown]: Madrid
What is the two-letter country code for this unit?
[Unknown]: ES
Is CN=Curso Kafka, OU=N/A, O=Curso, L=Madrid, ST=Madrid, C=ES correct?
[no]: yes
```

- Al acabar, veremos que hay un fichero **server.keystore.jks** en la ruta en la que hemos ejecutado nuestro comando.
- Siempre podemos consultar la información de nuestro **keystore** (siempre que tengamos su clave, claro) con el siguiente comando:

```
[kafka@kafka-server ~]$ keytool -list -v -keystore server.keystore.jks
```

- Desde la versión de JDK 8.51 a veces tiene problemas con los idiomas, si da error al ejecutarlo, probad a añadirle **-J-Duser.language=en**



```
[kafka@kafka-server ~]$ keytool -J-Duser.language=en -list -v -keystore server.keystore.jks
```

- Vamos a crear ahora nuestra entidad certificadora, y para ello vamos a usar **openssl**.
- Generaremos tanto la clave privada (que llamaremos ca-key) como la parte pública (que llamaremos ca-cert):

```
[kafka@kafka-server ~]$ openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
```

Al igual que cuando generamos nuestro **keystore**, nos solicitará cierta información (y vuelvo a usar **usuario** como clave)

```
Generating a 2048 bit RSA private key
.....+++
.....+++
writing new private key to 'ca-key'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
[...]
Country Name (2 letter code) [XX]:ES
State or Province Name (full name) []:Madrid
Locality Name (eg, city) [Default City]:Madrid
Organization Name (eg, company) [Default Company Ltd]:Entidad Certificadora
Organizational Unit Name (eg, section) []:NA
Common Name (eg, your name or your server's hostname) []:
Email Address []:
```

- Bien, tenemos nuestro **keystore**, tenemos una **entidad certificadora**.
- Vamos a decir que confiamos en dicha entidad añadiendo su clave pública a nuestro **truststore** (que todavía no existe, lo crearemos con el nombre de **server.truststore.jks**).
- Recordemos, si estamos con Java 8, mejor especificar el idioma con **-J-Duser.language=en**:

```
% keytool -J-Duser.language=en -keystore server.truststore.jks -alias EntidadNoFake
-import -file ca-cert
```

Una vez hecho esto, podemos confirmar que el certificado está disponible con **-list**

```
[kafka@kafka-server ~]$ keytool -J-Duser.language=en -list -v -keystore server.truststore.jks
```

- Ya tenemos un **keystore** y un **trustore**, en sus respectivos ficheros.
- Generamos una petición de certificación para nuestra clave almacenada con el **alias** de **kafka**, es decir, vamos a generar un fichero que contiene nuestra clave pública y nuestros datos para enviar a una **entidad certificadora**, y que nos lo devuelva firmado, corroborando que dicha información es correcta. Para ello usaremos la opción **-certreq**, que nos generará un fichero para firmar (especificado en **-file**).

```
[kafka@kafka-server ~]$ keytool -keystore server.keystore.jks -alias kafka -certreq -file cert-file
```

- Tras introducir la clave de nuestro **keystore**, generamos el fichero **cert-file** que hemos solicitado, con lo que ya podemos realizar el firmado de nuestra clave.
- Bueno, ahora vamos a usar **openssl** para firmar la petición de certificación que hemos creado en el punto anterior:

```
[kafka@kafka-server ~]$ openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file -out cert-signed -days 360 -CAcreateserial -passin pass:usuario
```

- **-CA** → La clave privada de la entidad certificadora
- **-in** → fichero sobre el que vamos a realizar el firmado
- **-out** → Fichero destino
- Al acabar, tendremos en **cert-signed** nuestro certificado, ahora si, firmado.
- Por último, vamos a importar el certificado generado y la clave pública de la entidad certificadora a nuestro **keystore** (acordaros del **-J-Duser.language=en** si usáis Java 8):

```
[kafka@kafka-server ~]$ keytool -keystore server.keystore.jks -alias EntidadNoFake -import -file ca-cert  
[kafka@kafka-server ~]$ keytool -keystore server.keystore.jks -alias kafka -import -file cert-signed
```

- Si listáis ahora con la siguiente instrucción, veréis que tenéis dos entradas, y dentro de **kafka** figuran los datos de la entidad certificadora que nos ha firmado.

```
[kafka@kafka-server ~]$ keytool -list -v -keystore server.keystore.jks
```

17.4.2. Securizando los Brokers

- En primer lugar, tenemos que editar los ficheros de configuración de nuestros **brokers**, y

cambiar la propiedad **listeners**, donde especificaremos cómo vamos a escuchar.

- Antes teníamos algo como:

```
listeners=PLAINTEXT://host.broker1:9092
```

- Ahora tendremos algo como:

```
listeners=SSL://host.broker1:9102
```

Podríamos configurar para escuchar de las dos formas, en distintos puertos, podríamos hacer algo como:



```
listeners=SSL://host.broker1:9102,PLAINTEXT://host.broker1:9092
```

- Modificaremos los 3 ficheros y cambiaremos los listeners por **SSL**, dejándolos así:

KAFKA_HOME/config/server.properties1

```
listeners=SSL://host.broker1:9102
```

KAFKA_HOME/config/server.properties2

```
listeners=SSL://host.broker1:9103
```

KAFKA_HOME/config/server.properties3

```
listeners=SSL://host.broker1:9104
```

- Hemos pedido que sólo se puedan comunicar vía **SSL**, pero ¿cómo saben dónde está su **keystore**? ¿y su **truststore**?
- Tenemos que especificarlo, así que tendremos que añadir a cada fichero las siguientes configuraciones:

```
ssl.keystore.location=/home/kafka/server.keystore.jks  
ssl.keystore.password=usuario  
ssl.key.password=usuario  
ssl.truststore.location=/home/kafka/server.truststore.jks  
ssl.truststore.password=usuario  
security.inter.broker.protocol=SSL  
ssl.endpoint.identification.algorithm=
```

- Iniciamos los tres **Brokers**

```
[kafka@kafka-server ~]$ zkServer.sh start  
[kafka@kafka-server ~]$ kafka-server-start.sh -daemon  
$KAFKA_HOME/config/server.properties1  
[kafka@kafka-server ~]$ kafka-server-start.sh -daemon  
$KAFKA_HOME/config/server.properties2  
[kafka@kafka-server ~]$ kafka-server-start.sh -daemon  
$KAFKA_HOME/config/server.properties3
```

- Recomendamos hacerlo de uno en uno, y vigilar las trazas con:

```
[kafka@kafka-server ~]$ tail -f $KAFKA_HOME/logs/kafkaServer.out
```

Podemos verificar que está escuchando mediante **SSL** con el siguiente comando:

```
[kafka@kafka-server ~]$ openssl s_client -debug -connect host.broker1:9104 -tls1  
CONNECTED(00000003)  
write to 0x2683900 [0x269d383] (181 bytes => 181 (0xB5))  
0000 - 16 03 01 00 b0 01 00 00-ac 03 01 a9 ad 97 c1 1f .....  
0010 - 4b 1c 3c b0 84 95 cd 68-67 b8 de ca d6 b0 e2 63 K.<....hg.....c  
0020 - d8 f4 17 f8 de 9f 22 44-e9 2b bd 00 00 64 c0 14 ....."D.+....d..  
0030 - c0 0a 00 39 00 38 00 37-00 36 00 88 00 87 00 86 ...9.8.7.6.....  
0040 - 00 85 c0 0f c0 05 00 35-00 84 c0 13 c0 09 00 33 .....5.....3  
0050 - 00 32 00 31 00 30 00 9a-00 99 00 98 00 97 00 45 .2.1.0.....E  
0060 - 00 44 00 43 00 42 c0 0e-c0 04 00 2f 00 96 00 41 .D.C.B...../...A  
0070 - c0 12 c0 08 00 16 00 13-00 10 00 0d c0 0d c0 03 .....  
0080 - 00 0a 00 07 c0 11 c0 07-c0 0c c0 02 00 05 00 04 .....  
0090 - 00 ff 01 00 00 1f 00 0b-00 04 03 00 01 02 00 0a .....  
00a0 - 00 0a 00 08 00 17 00 19-00 18 00 16 00 23 00 00 .....#..  
00b0 - 00 0f 00 01 01 .....  
...
```

17.4.3. Securizando los Clientes

Si queremos que nuestros clientes puedan conectarse mediante **SSL** a nuestros **Brokers**, tenemos que dar de alta la entidad certificadora en su propio **keystore**

```
[kafka@kafka-server ~]$ keytool -keystore client.truststore.jks -alias EntidadNoFake  
-import -file ca-cert
```

Y en su fichero **properties**, tenemos que añadir las siguientes opciones:

```
security.protocol=SSL  
ssl.truststore.location=client.truststore.jks  
ssl.truststore.password=usuario  
ssl.endpoint.identification.algorithm=
```

- Vamos a hacer esto mismo con el productor y consumidor de consola. Editamos los ficheros **\$KAFKA_HOME/config/consumer.properties** y **\$KAFKA_HOME/config/producer.properties**, añadiendo a su final:

```
security.protocol=SSL  
ssl.truststore.location=/root/client.truststore.jks  
ssl.truststore.password=usuario  
ssl.endpoint.identification.algorithm=
```

- Creamos un **Topic** sobre el que hacer las pruebas:

```
[kafka@kafka-server ~]$ kafka-topics.sh --bootstrap-server localhost:9092 --create  
--topic topic-seguro --partitions 5 --replication-factor 1
```

- Y lanzamos nuestro productor:

```
[kafka@kafka-server ~]$ kafka-console-producer.sh --broker-list host.broker1:9102  
--topic topic-seguro --property parse.key=true --property key.separator=,  
--producer.config $KAFKA_HOME/config/producer.properties
```

- Lanzamos nuestro consumidor, y verificamos que todo marcha como esperábamos:

```
[kafka@kafka-server ~]$ kafka-console-consumer.sh --bootstrap-server host.broker1:9102  
--topic topic-seguro --property print.key=true --consumer.config  
$KAFKA_HOME/config/consumer.properties --from-beginning
```

17.4.4. Autenticar clientes

- Si os veis en la necesidad de autenticar clientes contra el broker, vamos a dejar escrito los pasos que se deberían dar:
 - Habitilar la opción **ssl.client.auth=required** en el **server.properties** de los **brokers**
 - Generar un certificado para cada cliente, y firmarlo mediante la entidad certificadora
 - Importar el certificado del cliente firmado en el **trustore** de cada **broker**
- Añadir **ssl.client.auth=required**:

```
[kafka@kafka-server ~]$ echo "ssl.client.auth=required" >>  
$KAFKA_HOME/config/server.properties
```

- Generar y firmar certificados

```
[kafka@kafka-server ~]$ keytool -keystore client.keystore.jks -alias kafka -certreq  
-file cert-file-client  
[kafka@kafka-server ~]$ openssl x509 -req -CA ca-cert -CAkey ca-key -in cert-file-  
client -out cert-signed-client -days 360 -CAcreateserial -passin pass:usuario
```

- Añadir al **trustore** del servidor

```
[kafka@kafka-server ~]$ keytool -keystore server.truststore.jks -alias localhost  
-import -file cert-signed-client
```

- Una vez hecho esto, bastaría con reiniciar los **brokers**.

17.4.4.1. Generación de cliente

- En nuestro caso, vamos a usar un productor, pero podemos usar tanto un productor como un consumidor, ya que son las mismas.
- Para ello copiamos un productor como el siguiente

```

package com.curso.kafka.productorclima;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.ByteArraySerializer;
import org.apache.kafka.common.serialization.StringSerializer;

import com.curso.kafka.avro.model.Clima;

public class Productor {

    public static final String CITY = "madrid";
    public static final String TOPIC = "avro-clima";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer
        .class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        ByteArraySerializer.class.getName());

        KafkaProducer<String,byte[]> producer = new KafkaProducer<>(properties);

        Thread shutdownHook = new Thread(producer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);

        while(true) {
            Clima clima = OpenWeatherMap.getWeatherFromOpenWeatherMap(CITY);
            byte[] value = serializeClima(clima);
            ProducerRecord<String, byte[]> record = new ProducerRecord<>(TOPIC, CITY,
            value);
            producer.send(record);
            Thread.sleep(1500);
        }
    }

    private static byte[] serializeClima(Clima clima) throws IOException {
        return clima.toByteBuffer().array();
    }

}

```

- Ahora agregamos las directivas para que pueda autenticar

```

properties.put("security.protocol", "SSL");
properties.put("ssl.truststore.location", "/opt/kafka/keys/client.truststore.jks");
properties.put("ssl.truststore.password", "usuario");
properties.put("ssl.endpoint.identification.algorithm", "");

```

- La clase quedaría como sigue

```

package com.curso.kafka.productorclima;

import java.io.IOException;
import java.util.Properties;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.ByteArraySerializer;
import org.apache.kafka.common.serialization.StringSerializer;

import com.curso.kafka.avro.model.Clima;

public class Productor {

    public static final String CITY = "madrid";
    public static final String TOPIC = "avro-clima";

    public static void main(String[] args) throws InterruptedException, IOException {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer
                .class.getName());
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        ByteArraySerializer.class.getName());
        properties.put("security.protocol", "SSL");
        properties.put("ssl.truststore.location",
        "/opt/kafka/keys/client.truststore.jks");
        properties.put("ssl.truststore.password", "usuario");
        properties.put("ssl.endpoint.identification.algorithm", "");

        KafkaProducer<String,byte[]> producer = new KafkaProducer<>(properties);

        Thread shutdownHook = new Thread(producer::close);
        Runtime.getRuntime().addShutdownHook(shutdownHook);

        while(true) {
            Clima clima = OpenWeatherMap.getWeatherFromOpenWeatherMap(CITY);
            byte[] value = serializeClima(clima);
            ProducerRecord<String, byte[]> record = new ProducerRecord<>(TOPIC, CITY,
            value);
            producer.send(record);
        }
    }
}

```

```
        Thread.sleep(1500);
    }
}

private static byte[] serializeClima(Clima clima) throws IOException {
    return clima.toByteBuffer().array();
}

}
```

- Con esta configuración, se podrá iniciar la comunicación con el broker y se enviarán los mensajes por SSL