Procesos por lotes (batch) en Java

1.0.0.RELEASE

Tabla de contenidos

Capítulo 1. Introducción

El objetivo de este documento es presentar los principios para realizar los desarrollos de procesos batch basándose en tecnología Java y más concretamente con el framework Spring Batch.

1.1. Alcance funcional

Las distintas casuísticas que se cubren son las siguientes:

- **Procesamiento de ficheros**.
  - Cruzar dos ficheros de texto por clave, que se repite en ambos, generando uno o varios ficheros de salida en función de si se cumplen determinadas condiciones.
  - Ordenar la información de un fichero, de forma ascendente o descendente.
  - Discriminar la información de un fichero que tenga o no un cierto valor.
  - Contar el número de ocurrencias de una cadena.
  - Procesar la información de determinadas posiciones (ejemplo: de la 5 a la 20, únicamente las 500 primeras, etc.).

Estos puntos son aplicables vía etiqueta/campo en el caso de fichero en formato JSON o vía posición/longitud si se trata de un fichero de texto plano.

- **Procesamiento de base de datos**.
  - Leer información desde una o varias tablas de base de datos (Oracle, DB2, PostgreSQL o MongoDB).
  - Grabar información hacia base de datos (mismas que en el punto anterior).
- **Transferencia de ficheros**.
  - Leer un fichero desde una máquina.
  - Transferir un fichero a una máquina.
- **Invocación a servicios**. Independientemente de la fuente de la información, para cada registro, invocar a uno o varios servicios, los cuales y si aplica gestionarán la transaccionalidad. La invocación se realiza utilizando la librería de invocación existente en Bankia.
- **Datos funcionales y seguridad**. Durante el proceso de la información se consideran ciertos datos funcionales y de seguridad que influyen o pueden influir en dicho procesamiento. Para todo el proceso, se ha de tener en cuenta:
  - Fecha/hora del sistema que ejecuta el proceso.
  - Fecha/hora de referencia (por ejemplo, fecha contable).
  - Entorno operativo sobre el que se está ejecutando el servicio (pruebas, preproducción, producción).

En función del caso de uso es posible que el usuario (personal o aplicación) esté incluido dentro de la información a procesar. Será necesario, mediante el uso de una librería proporcionada por Bankia,

comprobar si el usuario tiene los permisos adecuados para ejecutar la funcionalidad en ese entorno (el proceso o registro correspondientes).

- **Monitorización**. Es preciso conocer, para cada proceso batch, si un registro se ha ejecutado correctamente o no. En caso de haber fallado, en qué paso se quedó y qué error se produjo. Para ello, dentro del procesamiento de la información, por cada registro, se genera la traza de procesamiento OK o erróneo correspondiente con la información del registro.

Asimismo, a la finalización del proceso batch completo, se genera una traza informando de si se ha producido algún error o no.

Todas las trazas se vuelcan a fichero, según especificación en el archivo de configuración del proceso batch.

1.2. Solución técnica

En el marco del proyecto, para cubrir las necesidades funcionales, se realizan distintas UseCases que definen e implementan los requisitos anteriores:

- **UseCase 1 - Lectura/escritura de ficheros de texto**. Leer un fichero de texto plano grande y combinarlo con fichero JSON pequeño, donde en ambos existe al menos un campo común, procesar únicamente los m registros desde la posición n del primer fichero y los x primeros registros del segundo, discriminando los registros que cumplen cierta condición en cada fichero, comprobando para cada registro la seguridad de ejecución para ese registro y generando dos ficheros:
  - Si campo X contiene valor, se genera fichero1 en texto plano, con ordenación ascendente.
  - Si campo X no contiene valor, se genera fichero2 en JSON, con ordenación descendente.

Para simulación, cada 10 registros procesados, el décimo se procesa como si hubiera sido erróneo, para mostrar la traza de error correspondiente, continuando el proceso hasta el final.

Asimismo, para optimizar el procesamiento, se cacheará / indexará la información.

- **UseCase 2 - Lectura/escritura de base de datos**. Lectura de registros desde una base de datos X relacional, transformar los registros y almacenarlos en una base de datos Y no relacional.
- **UseCase 3 – Transferencia de ficheros**. Lectura de fichero de texto plano desde un origen FTP, generar un fichero con el nº de ocurrencias de un patrón y subir.
- **UseCase 4 – Lectura fichero / escritura en base de datos**. Lectura de fichero de texto plano pequeño, transformar los registros y almacenarlos en una base de datos relacional.
- **UseCase 5 – Invocación a servicios**. Extendiendo la POC 2, por cada registro, invocar a un servicio REST utilizando para ello el cliente RestTemplate disponible en Spring Boot. La invocación se realizará al final del proceso. Si se desea invocar a un servicio vía mensajería (Kafka), se ha de implementar la seguridad Kerberos. == Guía del usuario

1.3. Introducción

Esta guía tiene como objetivo proveer de los conocimientos necesarios de las herramientas desarrolladas por el Departamento de Arquitectura de Aplicaciones de Bankia, para ejecutar procesos

Batch haciendo uso de las librerías creadas, así como configurar las distintas entidades de las que dependen los procesos.

1.4. Procesamiento por lotes

Se puede definir como proceso por lotes o *Batch processing* a la ejecución de un software sin el control o supervisión directa de un usuario. De esta manera, se tiene una ejecución que no precisa de interactuación ninguna con el usuario.

Su utilización es frecuente en grandes volúmenes de datos, que requieren de la realización de tareas repetitivas como su interpretación, procesamiento y salida, y suelen ser ejecutados en horarios con baja carga de trabajo a tenor de no influir en el entorno transaccional.

En esta guía se explica, paso a paso, la implementación de procesos de lotes basada en *Spring Batch* y ejecutándose con *Spring Boot*, añadiendo ejemplos y facilitando la creación y ejecución de los distintos procesos.

Como toda aplicación *Spring Boot* en Bankia, los procesos por lotes desarrollados soportan la carga de propiedades desde el servicio de configuración.

1.5. Componentes

El diseño de Spring Batch para la construcción de los procesos, como se aprecia en la figura, consta de:

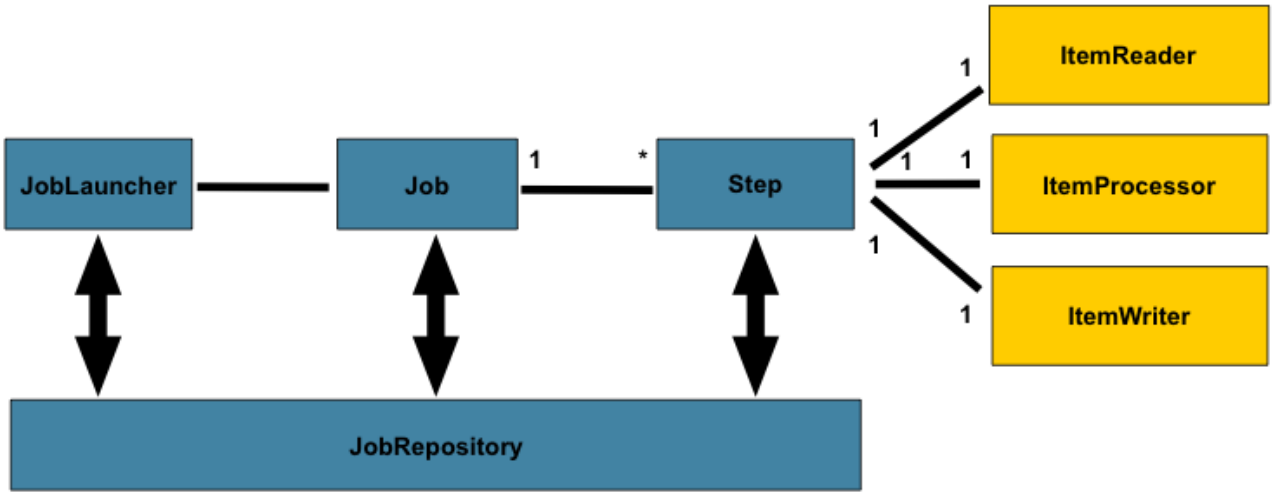


Figura 1.1. Esquema funcional de Spring Batch

- **JobRepository**: se encarga de la persistencia de los datos que requieren los procesos en curso.
- **JobLauncher**: se encarga de lanzar los procesos y suministrar los párametros de entrada.
- **Job**: es la entidad del trabajo en sí mismo,y engloba a todos los pasos o *steps* que se desarrollan en él. Cada Job se corresponde con una única aplicación (un único Spring Batch), el cual tendrá un único archivo de configuración `application.yml`.
- **Step**: Un *step* (paso) es un elemento independiente dentro de un Job (un proceso) que representa una de las fases de las que está compuesto dicho proceso. Un proceso (Job) debe tener, al menos, un step.

Sugerencia

Aunque no es obligatorio, un step puede estar compuesto de tres elementos: *reader*, *processor* y *writer*.

- **ItemReader**: Elemento responsable de leer datos de una fuente de datos de cualquier tipo.
- **ItemProcessor**: Elemento responsable del tratamiento de la información leída por el ` ItemReader`.
- **ItemWriter**: Elemento responsable de guardar la información leída por el *reader* ya sea o no tratada por el `ItemProcessor`. Si hay un *reader* debe haber un *writer*.

Se puede decir entonces que el diseño del frame se basa, en último término, en los *steps* o pasos que conforman el proceso, normalmente mediante una conversión de los datos tratados a **Chunks**, que se explica a continuación.

1.6. Chunk Oriented Processing

Spring Batch hace uso de la técnica **Chunk-Oriented** para la ejecución de las fases de un proceso, sin ser necesario que todos los pasos incorporen este enfoque.

El funcionamiento es el siguiente:

- El *reader* lee una porción de datos de la fuente y lo convierte en un *chunk*.
- Si el *processor* existe, este *chunk* es tratado, y el proceso se repite tantas veces como se indique hasta que con una cantidad de *chunks* determinadas se realice el trabajo de persistencia por el *writer*.

Para un fichero de texto plano con una estructura por tratamiento de líneas, si se establece el *step* con un intervalo de commit de 10, la orden ejecutada sería leer una línea del fichero, luego tratarla, leer otra línea del fichero, volver a tratarla, así hasta 10 veces. Una vez leídas y tratadas 10 líneas, el *writer* recibiría esa información (los 10 chunks) y los persistiría en base de datos. Este proceso se repetiría hasta terminar con todas las líneas el fichero.
Ejemplo 1.1. Procesamiento de datos

Procesos creados en la aplicación

La aplicación hace uso de todo lo anteriormente expuesto para realizar las tareas programadas en las distintas PoC. En el siguiente apartado se detallan la utilidad de dichas tareas programadas, especificadas no según los steps, si no según los propios procesos que realizan en función de si son *Reader*, *Processor* o *Writer*. Asimismo se desglosa la interacción que tienen con el archivo de configuración `application.yml`, archivo contenedor de todas las expresiones y datos de los que dependen. Tras éstos se desglosa la creación de los distintos *steps* que integrarán el *job* de Spring Batch.

Un ejemplo de PojoObj, aplicable a cualquier otro, y debiendo contener todos sus métodos setters y getters sería:

```
public class Cliente implements Serializable {

    private String var1;

    private String var2;
```

```
    private String var3;

    //... Se incluyen las variables que se desean manejar

    public String getVar1() {
        return var1;
    }
    public void setVar1(String var1) {
        this.var1 = var1;
    }
    //... Se incluyen los getters y setters de todas las variables
```

Todas las entidades de lectura, procesado y escritura en batch están pensadas para trabajar sobre estos POJOs, haciendo uso de ellos como contenedores de datos.

1.7. Readers

A continuación se explican los campos presentes en el `application.yml` referentes a la generación de los distintos `ItemReader` del proyecto, así como la creación de su objeto `@Bean` en el archivo de configuración Batch.

Según el tipo de fuente de lectura de los datos, se distingue entre lectura desde ficheros de texto o desde base de datos MongoDB.

Configuración y creación del `ItemReader` para archivos

Propiedades de Readers de fichero de texto.

```
bankia:
  batch:
    input:
      files:
        <identificadorYML>:
          file-type: json / plain-text
          format-type: delimited /position # only plain-text
          location-type: local / remote
          local-location:
          remote-location:
          delimiter:      # Only for format-type: delimited
          fields:         # Only for plain-text
          positions:      # Only for format-type: position

          #OPTIONAL
          strict:         # Only for format-type: position
          lines-to-skip:
          max-item-count:
```

- `<identificadorYML>:` nombre que permitirá el acceso a la configuración de los distintos Beans.
- `file-type:` esta variable contiene la definición del tipo de archivo a procesar por el *Reader*. Sus opciones son:
  - `json:` para procesar archivos de este tipo.
  - `plain-text:` para procesar tanto archivos con delimitadores como por posiciones fijas.
- `format-type:` esta variable solo es aplicable a los archivos de texto plano, y sus opciones son:
  - `delimited:` configura el archivo de texto plano con separador entre campos mediante caracter, como puede ser .csv (*Character-Separated-Values*).

  - `position:` configura el fichero de texto plano como uno de posiciones y longitudes fijas para los campos.

- `location-type:` esta variable contiene el tipo de ruta al archivo a procesar. Puede ser
  - `local:` la ruta se especifica entonces en la variable `local-location`.
  - `remote:` la ruta se especifica entonces en la variable `remote-location`. En este caso, es necesario indicar la ruta local donde se almacenará temporalmente el fichero para su lectura.
- `delimiter:` especifica la cadena delimitadora. Para archivos de tipo de formato `delimited`.
- `fields`: especifica los campos requeridos para su lectura. Se han de colocar con un separador `-` en las líneas subsiguientes. Para archivos de tipo `plain-text`.
- `positions`: especifica la posición inicial y final de los campos requeridos. Debe tener el mismo número de *items* que el campo `fields`. Para archivos de tipo de formato `position`.

OPCIONALES

- `strict:` Booleano que controla si el programa continua tras encontrar un campo cuyos datos no cumplen con los parámetros de posición (`false`) o no (`true`). **De manera predeterminada está inicializado como `true`**. Para archivos de tipo de formato `position`.
- `lines-to-skip:` permite indicar el inicio de la lectura del fichero, indicándole con un entero cuantas líneas debe ignorar al inicio del documento.
- `max-item-count:` permite indicar el máximo número de filas que deben ser leídas durante el proceso.

La configuración para un *@Bean* `ItemReader` de archivo, creado mediante estas propiedades de manera genérica es:

```
@Bean("<identificadorBean>")
   public ItemReader<PojoObj> PojoObjReader() {

       return new BankiaFlatFileItemReaderBuilder<PojoObj>() 1
               .batchProperties(batchProperties) 2
               .input("<identificadorYML>") 1
               .targetType(PojoObj.class) 4
               //Exclusivo para JSON---------
               .transformer(new PojoObjFromMapper()) 5
               //----------
               .build(); 6
```

`BankiaFlatFileItemReaderBuilder<PojoObj>` es una clase perteneciente a la librería generada para la aplicación.

El seguimiento en el proceso es el siguiente:

1. Se crea la instancia desde el `builder`.
2. Se indica donde tiene que buscar la configuración, en este caso, `BankiaBatchProperties`.
1. Se enlaza el BankiaBatchProperties con el fichero application.yml, especificando donde se encuentra la configuración en el mismo. Esta relación se establece mediante la línea de código: `private static final String INPUT_FILETYPE_POSITION = "<identificadorYML>"` presente al inicio de la clase `batchProperties`.

4. Se indica el POJO contenedor donde se guardarán los datos provisionalmente para su procesamiento posterior.
5. En el caso de generar un `bean` de lectura de json se requiere un transformer definiendo esta propiedad que a partir de un `Map` permite generarlo.
6. Devuelve mediante el método un objeto ` ItemReader` configurado según los parámetros anteriores.

Ejemplo 1.2. Definiendo Bean de lectura de fichero

Importante

Durante los ejemplos de código, se notarán como `PojoObj` o `PojoObjOutput` las clases que se dispongan para contener los campos obtenidos de la lectura del fichero o los campos deseados para su salida. Estas clases deben contener, al menos, todos los campos descritos en el fichero de configuración al que se hace referencia a lo largo de las demostraciones, llamado `application.yml`.

Para cada caso, los términos `"<identificadorBean>"` y `"<identificadorYML>"` deben ser inequívocos e irrepetibles en los distintos Beans y archivo de configuración de la aplicación. De no ser así, la aplicación falla.

Ejemplo de configuración para fichero JSON

La generación de su `ItemReader` es:

```
@Bean("json-reader-example-bean")
   public ItemReader<PojoObj> PojoObjReader() {

       return new BankiaFlatFileItemReaderBuilder<PojoObj>()
               .batchProperties(batchProperties)
               .input("json-reader-example")
               .targetType(PojoObj.class)
               .transformer(new PojoObjFromMapper())
               .build();
```

Ejemplo 1.3. ItemReader para JSON

El seguimiento del proceso está definido y explicado en el apartado de creación mencionado anteriormente.

Aviso

El código del `ItemReader` para lectura de fichero JSON es el único que ejecuta el comando `transformer`, generando el POJO con los campos automáticamente.

En el siguiente ejemplo se puede ver una configuración de las variables del fichero `Application.yml`, según las pautas explicadas en la sección previa.

```
bankia:
  batch:
    input:
      files:
        json-reader-example:
          file-type: json
          location-type: local
          local-location: ${INPUT_LOCATION:D:/tmp/FileInput.json}
```

Conviene indicar que para la ruta del archivo se usa la notación de valores por defecto de *Spring*, utilizando indistintamente la variable de entorno `INPUT_LOCATION` o el valor por defecto `FileInput`.

Ejemplo 1.4. JSON

Ejemplo de generación de Bean para fichero de texto plano

El siguiente bloque de código es un ejemplo de configuración de su `ItemReader`:

ItemReader para texto plano.

```
@Bean("<identificadorBean>")
    public ItemReader<PojoObj> pojoObjReader() {

        return new BankiaFlatFileItemReaderBuilder<PojoObj>()
                .batchProperties(batchProperties)
                .input(<"IdentificadorYML">)
                .targetType(PojoObj.class)
                .build();
```

Esta configuración es idéntica independientemente del tipo de formato del fichero de texto plano. Será la configuración del `batchProperties.java` la que cambie sustancialmente.

Ejemplo de configuración para texto plano con separación por posiciones fijas

texto plano por posiciones.

```
bankia:
  batch:
    input:
      files:
        position-example: 1
          file-type: plain-text 2
          format-type: position 1
          location-type: local  4
          local-location:  ${INPUT_LOCATION:D:/tmp/Input.csv} 5

          fields: 6
             - var1
             - var2
             - var3
             - var4
             - ...
             - varCondition
          positions: 7
             - 1-2
             - 3-9
             - 10-20
             - 21-24
             - ...
             - 98

          # Optional Parameters
          strict: false 8
          max-item-count: 3000 9

          # Processor Parameters...
```

1 Este es el `<identificadorYML>`.
2 Se especifica el tipo de archivo, texto plano.
1 Se especifica la separación de campos, en este caso por posición.
4 Se especifica el modo de acceso al archivo de lectura y la ruta al mismo.
5 Campos a capturar en el `PojoObj` por el ` ItemReader`.
6 Posición inicial y final de cada uno de los campos a capturar en el archivo. La lista debe contener el mismo número de *items* que `fields` se desean procesar.
7 8 OPCIONAL. En este caso, se configura como `false` para que ignore errores en caso de no coincidencia de posiciones en campos de alguna de las filas.

9 OPCIONAL. Delimita a una cantidad fija las filas que se procesarán en la lectura. Está definido, en este caso, un número máximo de 3000 *items* leídos.

Nota

Para el último campo de `positions`, si se corresponde con el último campo del archivo, basta indicar la posición inicial.

Ejemplo de configuración para texto plano con separación por delimitador

texto plano por delimitador.

```
bankia:
  batch:
    input:
      files:
        delimiter-example: 1
          file-type: plain-text
          format-type: delimited 2
          delimiter: ";" 1
          location-type: local
          local-location: ${INPUT_LOCATION:D:/tmp/FileInput.csv}
          fields:
            - var1
            - var2
            - var3
            - ...
            - varCondition

          # Optional parameters

          lines-to-skip: 100 4
          max-item-count: 4500

          # Processor Parameters...
```

1 Este es el `<identificadorYML>`.
2 Se especifica la separación de campos, en este caso mediante delimitador.
1 Se especifica el delimitador de los campos.
4 OPCIONAL. Indica la posición de la fila desde la cual se efectúa la lectura del archivo. En este caso, se procesarían los *items* $_{100 \rightarrow 4600}$.

Se han obviado los campos repetidos en el anterior apartado.

Configuración y creación del `ItemReader` para base de datos MongoDB

```
bankia:
  batch:
    collection: bankiaCollection-${random.value}
    input:
      mongo:
        <identificadorYML>:
            json-query:
            parameters:
            sorts:
          codigoFuncion: ASC / DESC
```

- `collection:` genera un valor aleatorio para cada colección generada. Se recomienda no cambiar el valor por defecto.

- `<identificadorYML>:` debe coincidir con el incluido en el método input de la generación del bean.

- `json-query:` en esta variable se incluyen las expresiones de consultas para el filtrado de datos.
- `parameters:` en esta variable se incluye los parámetros que utilizarán las consultas para el filtrado.
- `codigoFuncion:` especifica el modo de salida de las consultas. Puede ser:
  - ASC: indica un orden ascendente de los datos filtrados.
  - DESC: indica un orden descendente de los datos filtrados.

Ejemplo 1.5. Configuración de ItemReaders de MongoDB

La aplicación incluye las configuraciones necesarias para crear, tan solo cambiando los datos del `application.yml`, un `ItemReader` que lea los *items* procesados y almacenados por los `steps` previos en MongoDB. La generación de su `bean` tiene la siguiente estructura:

```
@Bean("<identificadorBean>")
    public ItemReader<PojoObjOutput> pojoObjOutputReader(MongoOperations template) {

        return new BankiaMongoItemReaderBuilder<PojoObjOutput>() 1
            .batchProperties(batchProperties) 2
            .collection(mongoProperties.getCollection()) 1
            .input("<identificadorYML>") 4
            .targetType(PojoObjOutput.class) 5
            .template(template) 6
            .build(); 7
        }
```

- En este caso, el `ItemReader` necesita una instancia de tipo `MongoOperations` de la librería Spring Data MongoDB, debiendo estar incluida esta dependencia en el archivo `pom.xml` del proyecto.
- `BankiaMongoItemReaderBuilder` es una clase perteneciente a la librería generada para la aplicación.

El seguimiento en el proceso es el siguiente:

1. Se crea una instancia de la clase `builder`.
2. Se determina donde tiene que buscar la configuración.
1. Obtiene el valor de la colección de mongo a utilizar, incluida al inicio del archivo.
4. Se enlaza el BankiaBatchProperties con el fichero `application.yml`, especificando donde se encuentra la configuración en el mismo. El identificador debe coincidir con el identificador contenido en el archivo de configuración.
5. Se especifica el POJO donde se guardarán los datos provisionalmente para su procesamiento posterior.
6. Método necesario por defecto para la generación de un `ItemReader` de MongoDB.
7. Se devuelve mediante el método un objeto `ItemReader` configurado según los parámetros anteriores.

Ejemplo 1.6. ItemReader para MongoDB

Ejemplo de configuración para lectura desde MongoDB, con consulta inclusiva y salida ascendente.

```
bankia:
  batch:
  collection: bankiaCollection-${random.value}
    input:
      mongo:
        mongo-asc-example: 1
```

```
          json-query: "{var1 : ?0}" 2
          parameters: "0000583" 1
          sorts:
            codigoFuncion: ASC 4
```

1 Valor a asignar en el método `input()` en el código de generación del bean.
2 1 Consulta de todos los *items* que contienen el valor contenido en `parameters` en su `var1`.
4 Salida del filtrado de datos según la consulta ascendente.

Ejemplo de configuración para lectura desde MongoDB, con consulta exclusiva y salida descendente.

```
bankia:
  batch:
  collection: bankiaCollection-${random.value}
    input:
      mongo:
        mongo-desc-example:
          json-query: "{var2 : { $ne : ?0 }}" 1
          parameters: "000444" 2
          sorts:
            codigoFuncion: DESC 1
```

1 2 Consulta de todos los *items* que no contienen el valor contenido en `parameters` en su `var2`.
1 Salida del filtrado de datos según la consulta descendente.

1.8. Processors

A continuación se explican los campos presentes en el `application.yml` referentes a la generación de los distintos `ItemProcessor` del proyecto, así como la generación de su objeto `@Bean` en el archivo de configuración Batch.

Configuración y creación del `ItemProcessor` para los distintos *steps* de fichero

Propiedades de Processors de fichero de texto.

```
bankia:
  batch:
    input:
      files:
        <identificadorYML>:
          # ReaderProperties...
          ...#
          filter-expression:
          security:
            field:
            function:
```

- `<identificadorYML>`: nombre que permitirá el acceso a la configuración de los distintos Beans.
- `filter-expression:`
- `security:` Esta sección contiene la definición de seguridad a aplicar para cada registro leído. Sus opciones son:
  - `field:` campo elegido para la comprobación de seguridad.
  - `function:` función de seguridad provista por la entidad en función del campo introducido.

La estructura para un *@Bean* `ItemProcessor`, creado mediante estas propiedades de manera genérica es:

```
@Bean("<identificadorBean>")
   public ItemProcessor<PojoObjUsed,PojoObjObjetive> 1
   pojoObjUsedProcessor(/* different instances */ ) { 2

       BatchInputFile batchInputFile =
batchProperties.getInput().getFiles().get("<identificadorYML>"); 1

       return new BankiaCompositeItemProcessorBuilder<PojoPobj, PojoPobj>() 4
               .filterProcessor() 5
               .processor() 6
               .build(); 7
       }
```

`BankiaCompositeItemProcessorBuilder<PojoObj, PojoObj>` es una clase perteneciente a la librería generada para la aplicación.

El seguimiento en el proceso es el siguiente:

1. Según la finalidad, se introducen genéricos del Pojo para el proceso (`PojoObjUsed`) y la salida que se desea tras el proceso (`PojoObjObjetive`).
2. Las instancias dependerán del `filterProcessor` a desarrollar. Para un filtrado de seguridad, se deberá definir `SecurityValidation securityValidation`. Para el uso de caché, `CacheService cacheservice`, y para la generación de un nuevo archivo que incluya campos de distintos PojoObj, `PojoObjUsedMapper mapper` .
1. Carga la configuración `<identificadorYML>` presente en el `application.yml`.
4. `BankiaCompositeItemProcessorBuilder<PojoObjUsed, PojoObjObjetive>` es una clase perteneciente a la librería generada para la aplicación.
5. Se especifican los filtros a aplicar por el *processor* .
6. Se especifica el procesador que va a llevar a cabo los filtros..
7. Con el método `build()`, se devuelve un `ItemProcessor()` con el `return`.

Ejemplo 1.7. Definiendo Bean genérico de proceso de fichero

Las definiciones de los distintos `filter processor` hacen uso de las siguientes instancias de clases de la librería propia de la aplicación según la finalidad del filtrado:

- De seguridad: Genera un filtrado según unos parámetros de seguridad de usuario dependientes del campo y función introducidos. Los campos `<field>` y `<function>` son llamados mediante sus correspondientes métodos desde el la clase `BatchInputFile`.

`BankiaSecurityItemProcessor<>(SecurityValidation securityValidation, "field", "function")`.

- De filtrado:
  - Simple: Realiza un filtrado simple mediante expresión de tipo Spring Expression Language (SpEL).

`BankiaFilterItemProcessor<>("expression")`

- En caché: realiza un filtrado comparativo de un archivo y otro previamente procesado y guardado en caché, generando un nuevo fichero mezcla de los dos anteriores según la función indicada.

`BankiaCacheFilterItemProcessor<PojoObjUsed,PojoObjObjetive> (cacheService, cacheClientes, "expression", "varPojoObjUsed", "varPojoObjObjetive")`

Para el método `processor()` existen dos maneras de generar la instancia que necesita:

- Código para *processor* de un solo fichero: este método únicamente devuelve la instancia introducida.

```
.processor(PojoObjUsedItemProcessor())
```

- Código para *processor* de dos ficheros: este método configura el proceso que generará el bean según los campos introducidos previamente.

```
.processor(PojoObjUsedItemProcessor(cacheService,cacheCliente,mapper))
```

Este proceso requiere de la obtención de los datos de ambos archivos mediante dos cachés para generar el nuevo fichero con los campos especificados por el mapper.

Ejemplo de configuración para proceso de un fichero json

application.yml.

```
bankia:
    batch:
      input:
        files:
          json-example:
            # ReaderProperties...
            ...#
            filter-expression:"var1 == 'A170914'" (1)
            security:
              field: "var2" (2)
              function: "FFFADADF" (1)
```

(1) Filtrado inicial según expresión.
(2) (1) Filtrado de expresión de seguridad.

Bean.

```
    @Bean("json-processor-bean")
    public ItemProcessor<JsonPojoObj, JsonPojoObj> clientesProcessor(SecurityValidation
securityValidation) {
        BatchInputFile batchInputFile = batchProperties.getInput().getFiles().get("json-example");
        return new BankiaCompositeItemProcessorBuilder<JsonPojoObj, JsonPojoObj>()
                .filterProcessor(new
BankiaFilterItemProcessor<>(batchInputFile.getFilterExpression()))
                .filterProcessor(new BankiaSecurityItemProcessor<>(securityValidation,
                        batchInputFile.getSecurity().getField(),
batchInputFile.getSecurity().getFunction()))
                .processor(new JsonPojoObjItemProcessor()).build();
        }
```

Ejemplo de configuración para procesar dos ficheros en uno nuevo, con uno almacenado en caché

Application.yml.

```
bankia:
  batch:
    input:
      files:
        delimiter-example:
          # ReaderProperties...
          ...#
          filter-expression:"json-example.var1 == cache-example.var1" (1)
```

1 Filtrado según expresión. Esta expresión nos generará un nuevo fichero con los valores de las filas de ambas tablas que cumplan la condición indicada.

Nótese que la configuración del *processor* forma parte de los procesos de lectura, proceso y escritura de un determinado tipo de fichero. En este caso, esta configuración podría ser la continuación del ejemplo de texto plano separado mediante delimitador.

Bean.

```
@Bean("ptext-processor-bean")
    public ItemProcessor<PojoObj,PojoObjOutput> 1
    clientesProductoProcessor(CacheService cacheService,
            ClienteProductoMapper mapper) {
        BatchInputFile batchInputFile = batchProperties.getInput().getFiles().get("delimiter-example");
        return new BankiaCompositeItemProcessorBuilder<PojoObj,PojoObjOutput>()
                .filterProcessor(new BankiaCacheFilterItemProcessor<PojoObj,PojoObjOutput>(cacheService,
                        cacheClientes, batchInputFile.getFilterExpression(), "json-example", "delimiter-
example"))
                .processor(new PojoObjItemProcessor(cacheService, cacheClientes, mapper)).build();
        }
```

1 En este caso, en los genéricos se tienen dos clases distintas de POJO, al ser el fichero de salida diferente del fichero de entrada, teniendo éste último campos pertenecientes tanto al fichero del texto delimitado por comas como al json previamente procesado y escrito en caché. `PojoObjOutput` contendrá campos pertenecientes a ambos ficheros, siendo el mapper el encargado de crear esta combinación de los campos.

Configuración y creación del `ItemProcessor` para MongoDB

No existen en el `application.yml` variables para la configuración de su `ItemProcessor`, ya que la clasificación ascendente o descendentes de los *items* se realiza durante la propia lectura de la información de la base de datos. De esta manera, la configuración de su bean es únicamente metodológica, siendo:

Definiendo Bean genérico de proceso de MongoDB.

```
    @Bean("mongo-processor-bean")
    public ItemProcessor<PojoObjOutput, PojoObjOutput> pojoObjOutputProcessor() {
        return new PojoobjOutputItemProcessor();
        }
```

1.9. Writers

Tras los `ItemProcessor` se explican los campos presentes en el `application.yml` referentes a la generación de los distintos `ItemWriter` del proyecto, así como la generación de su objeto `@Bean` en el archivo de configuración Batch.

Configuración y creación del `ItemWriter` para los distintos *steps* de fichero

Propiedades de Writers de fichero de texto.

```
bankia:
  batch:
    output:
      files:
        <identificadorYML>:
          file-type: plain-text / json
```

```
        format-type: delimited /position #(plain-text only)
        delimiter:    #(delimited only)
        format:       #(position only)
        location-type: local / remote
        local-location:
        remote-location:
        fields:
          - var1
          - var2
          ...
```

- `<identificadorYML>`: nombre que permitirá el acceso a la configuración de los distintos Beans.
- `file-type:` se especifica el modo de salida del fichero ya procesado.
- `format.type:` concreta el formato de salida según el tipo de fichero seleccionado.
- `delimiter:` si el tipo de formato es `delimiter`, determina la separación de los campos del fichero de salida `.csv`.
- `format` si el tipo de formato es `plain-text`, determina la separación de los campos del fichero de salida mediante lenguaje SpEL.
- `location-type:` especifica el tipo de directorio donde se almacenará el archivo generado.
- `fields` especifica los campos a generar por el proceso de escritura.

Como se puede observar, las distintas configuraciones son homólogas a las encontradas en los `beans` de lectura, la diferencia principal del proceso es que éste hace uso de esta configuración para generar los archivos.

La estructura de los *beans* para los `ItemWriter` de escritura de fichero es genérica para todos ellos:

Bean.

```
1  @Bean("<identificadorBean>")
      public ItemWriter<PojoObjOutput> pojoObjOutputWriter() {
          return new BankiaFlatFileItemWriterBuilder<PojoObjOutput>() 1
          .batchProperties(batchProperties) 2
5         .output("<identificadorYML>") 1
          .build(); 4
          }
```

1 Se genera la clase incluida en la librería creada para la aplicación, instanciando el POJO con las variables de salida.
2 Se enlaza el `bean` con el archivo de configuración que contiene las variables indicadas en el `application.yml`.
1 Se especifica, dentro del archivo de configuración, el identificador que contiene las variables configuradas para la salida del archivo.
4 Se llama al método build, que devuelve la clase requerida por el método.

Configuración y creación del `ItemWriter` para escritura en caché

Para la escritura de los datos en caché se invoca al servicio de caché incluido en la librería específica para Bankia, `CacheService`.

Para la creación de un _bean_de escritura de este tipo, la estructura es la siguiente:

```
@Bean("<IdentificadorBean")
 public ItemWriter<PojoObjOutput> PojoObjOutputWriter(CacheService cacheService){

 return BankiaCacheItemWriter<> (cacheService, 1
 cacheIdName, 2
 FilterExpression) 1
    }
```

1 Se inyecta a una clase writer de escritura en caché de la librería propia (o una adaptación de la misma) el servicio de caché creado.
2 Se nombra la caché para su identificación.
1 Se introduce la expresión de filtrado que empleará el proceso de escritura.

Ejemplo de escritura de fichero de texto plano mediante posiciones

```
bankia:
  batch:
    output:
      files:
        ptext-posicion-example:
          file-type: plain-text
          format-type: position
          format: "%-36s%-36s%-7s"
          fields:
            - var1
            - var2
            - var3

          location-type: local
          local-location: ${OUTPUT_LOCATION}/ptext-posicion-example.txt
```

Ejemplo 1.8. Configuración

Su configuración en el `BankiaBatchProperties` sería:

Bean.

```
1  @Bean("ptext-position-example-bean")
      public ItemWriter<PojoObjOutput> pojoObjOutputWriter() {
          return new BankiaFlatFileItemWriterBuilder<PojoObjOutput>()
          .batchProperties(batchProperties)
5         .output("ptext-position-example")
          .build();
          }
```

Importante

Si el *bean* es creado para utilizarse como una de las salidas según una expresión de filtrado, debe sustituirse la expresión:

```
public ItemWriter<PojoObjOutput> pojoObjOutputWriter() {
...
}
```

por:

```
public FlatFileItemWriter<PojoObjOutput> pojoObjOutputWriter() {
...
}
```

Configuración y creación del `ItemWriter` para escritura en MongoDB

Para la escritura de los datos en Mongo, debido a las clases que incorpora Spring no hay que indicar ninguna variable específica en el `application.yml`.

Para la creación de un _bean_de escritura en MongoDB, la estructura es la siguiente:

Bean.

```
@Bean("<IdentificadorBean")
   public ItemWriter<PojoObjOutput> PojoObjOutputWriter(MongoOperations template){

   return MongoItemWriter<PojoObjOutput> mongoItemWriter =
     new MongoItemWriterBuilder<PojoObjOutput>() <1>
.collection(mongoProperties.getCollection()) 1
      .template(template) 2
      .build(); 1
      }
```

Se crea la clase propia de Spring-Batch para MongoDB y se lanza su clase constructora.

1 Se obtiene la colección.

2 Se instancia la plantilla, propia de la dependencia `Spring Data MongoDB`.

1 Se llama al método build, que devuelve la clase requerida por el método.

Ejemplo de configuración de *Bean* para generación de distintos tipos de salida según filtrado por condiciones

A raíz de todo lo anterior, se explica a continuación un `ItemWriter` que generaría distintas salidas (tanto fichero de texto como base de datos) según unas consultas preestablecidas en el `application.yml`.

Bean.

```
@Bean("OutputMix-example-Bean")
    public ItemWriter<PojoObjOutput> PojoObjOutputWriter( 1
            @Qualifier("ptext-position-example-bean")
            ItemWriter<PojoObjOutput> ptext-position-example-beanWriter,
         /* @Qualifier("json-example-bean") 2
            ItemWriter<PojoObjOutput> ptext-position-example-beanWriter, */
            MongoOperations template) {

        MongoItemWriter<PojoObjOutput> mongoItemWriter = new
 MongoItemWriterBuilder<ClienteProductoSalida>()
                .collection(mongoProperties.getCollection())
                .template(template)
                .build();

        String classifierExpression = batchProperties.getOutput().getClassifier().get("filtrado"); 1

        Map<String, ItemWriter<? super PojoObjOutput>> matcherMap = new HashMap<>(); 4
        matcherMap.put("expression1", ptext-position-example-beanWriter);
        matcherMap.put("others", mongoItemWriter);

        BackToBackPatternClassifier<PojoObjOutput, ItemWriter<? super PojoObjOutput>> classifier = new
 BackToBackPatternClassifier<>();
        classifier.setRouterDelegate(new BankiaOutputClassifier<PojoObjOutput>(classifierExpression));
        classifier.setMatcherMap(matcherMap);

        ClassifierCompositeItemWriter<PojoObjOutput> result = new ClassifierCompositeItemWriter<>();
        result.setClassifier(classifier);

        return result;
```

1. Se crea el *bean* de escritura que ejecutará el filtrado. Éste debe instanciar los diferentes `FlatFileItemWriter` (ItemWriter para filtrado), así como identificarlos mediante la anotación `@Qualifier("<identificadorWriterBean>")`.
1. Se obtiene la expresión del filtrado del `application.yml`.
4. Se llama al método `put`, enlazando cada identificador de la expresión SpEL de filtrado con el *bean* de escritura objetivo. Si se programaran más de dos salidas, habría que añadir otro línea que volviera a llamar al método y que contuviera el respectivo identificador introducido en el filtrado.
2. Comentario que indica la manera de implementar una nueva salida para json, previa creación de su `FlatFileItemWriter` y de su configuración en el `application.yml`.

Un ejemplo de consulta para este *bean* es:

```
bankia:
  batch:
    output:
      classifier:
        filtrado: "(var1 == 'XXXXXX-0000') ? 'expression1' : 'others')"
```

1.10. Tasklets

Cuando se requieran procesos durante la ejecución de la aplicación que no requieran procesos por lotes, como descargas de ficheros desde servidor, conexiones…, se desarrollarán *tasklet* que luego se incorporarán en los *steps*.

Ejemplo de *tasklet* para ordenación de archivo mediante subdivisión de fichero

Para la creación del *bean* de un tasklet de este tipo se escribe:

```
@Bean("sorterExampleTasklet")
    public Tasklet sorterExampleTasklet() {

        return new BankiaSortFileTasklet(output.getTemporalLocation(), output.getLocalLocation(), 1
                (String u1, String u2) -> { 2
                    String t1 = ...
                    String t2 = ...
                    return t1.compareTo(t2);
                });
```

1. Utilizamos la librería de Bankia creada para el efecto, `BankiaSortFileTasklet`, e introducimos las variables de directorio.
2. Se crea un comparador de caracteres, dependiendo del tipo de fichero de texto que tengamos.

1.11. Steps

Para generar un `step` o paso en Spring Batch, se hace uso de las entidades `ItemReader`, `ItemProcessor` e `ItemWriter` generadas según lo explicado anteriormente.

De forma genérica, para crear un `step` se define en el `BatchConfiguration`:

```
@Bean("<identificadorStep1>")
public Step stepMiStep(
            @Qualifier("<Step1Reader>") ItemReader<PojoObj> reader, 1
            @Qualifier("<Step1Processor>") ItemProcessor<PojoObj, PojoObjOutput> processor, 2
            @Qualifier("<Step1Writer>") ItemWriter<PojoObjOutput> writer, 1
```

```
            @Qualifier("<Step1OutputWriter>") FlatFileItemWriter<PojoObjOutput> step1OutputWriter, 4
            @Qualifier("taskExecutor") TaskExecutor taskExecutor) { 5

        return StepBuilderFactory.get("<step1bean>") 6
                .<PojoObj, PojoObjOutput> chunk(batchProperties.getChunkSize()) 7

                .reader(reader)
                .processor(processor) 8
                .writer(writer)

                .stream("<Step1OutputWriter>") 9

                .listener(new BankiaItemReaderListener<>())
                .listener(new BankiaChunkCountListener("PojoObj", <number>)) 10

                .taskExecutor(taskExecutor)
                .throttleLimit(batchProperties.getThrottleLimit()) 11

                .build(); 12
```

1. Se inyecta la entidad de lectura del `step`.
2. Se inyecta la entidad de procesado del `step`.
1. Se inyecta la entidad de escritura del `step`.
4. En caso de escritura con varias salidas, se introducen las distintas salidas a fichero. En caso de tener más salidas que la mostrada en el código, habrían de introducirse con sus respectivos `@qualifier()`.
5. Se instancia el `bean` de `taskExecutor`.
6. Se introduce en el método `get()` el identificador del `bean`.
7. Se llama a la variable `ChunkSize` del archivo `application.yml`.
8. Se introducen los tres procesos que forman el `step` con los métodos del `StepBuilder`.
9. Si hay multisalida, se introducen las llamadas necesarias al método `strem()` introduciendo un `FlatFileItemWriter` de los inyectados al constructor por cada invocación.
10. Controla los avisos del conteo de *items*. Cambiando `<number>` se altera el contador de activación del logger del proceso de lectura.
11. En caso de querer limitar el número de *items* que se leen durante el proceso de lectura, se llama a este método, que captura del archivo de configuración el valor para dicho límite.
12. Se llama al método del `StepBuilder` que genera el `step`.

Nota

En este modelo genérico `PojoObj` es el POJO que contiene las variables que se desean obtener en la lectura del `step`, mientras `PojoObjOutput` corresponde al que tiene las variables de los *items* que se desean generar en la salida.

El tamaño de los chunks se especifica en el `application.yml` en `bankia.batch`, donde también se encuentran otras variables de las que dependen los `steps`:

```
bankia:
  batch:
    thread-pool:
      core-pool-size: 32
#      max-pool-size:
#      queue-capacity:
    step:
      <indentificadorYML>:
        chunk-size: 100
        throttle-limit: 20
```

- `core-pool-size:` indica el número máximo de hilos que se pueden abrir simultáneamente.

Las siguientes variables se especifican para cada *step*:

- `chunk-size:` determina el tamaño de los chunks en los procesos.
- `throttle-limit:` indica el número límite de *items* a procesar por el `step`.

Ejemplo de configuración de *step* con *tasklet* incorporado

La configuración java para un bean de tasklet es:

```
@Bean("taskletStep")
   public Step stepTasklet(
           @Qualifier("exampleTasklet") Tasklet exampleTasklet,
           @Qualifier("taskExecutor") TaskExecutor taskExecutor) {

       return stepBuilderFactory.get("taskletStep").tasklet(exampleTasklet)
       .taskExecutor(taskExecutor).build();
```

Con el método `.tasklet(exampleTasklet)` inyectamos el *tasklet* en el stepBuilderFactory.

Ejemplo de configuración de step para un mismo POJO para entrada y salida

El código para el 'BatchConfiguration` sería:

```
@Bean("stepProductoSalida")
    public Step stepProductoSalidaDesc(
            @Qualifier("stepProductoSalidaReader") ItemReader<ProductoSalida> reader,
            @Qualifier("stepProductoSalidaProcessor") ItemProcessor<ProductoSalida, ProductoSalida>
 processor,
            @Qualifier("stepProductoSalidaWriter") ItemWriter<ProductoSalida> writer,
            @Qualifier("taskExecutor") TaskExecutor taskExecutor) {
        // @formatter:off
        return stepBuilderFactory.get("stepProductoSalida")
                .<ProductoSalida, ProductoSalida> chunk(batchStep.getChunkSize())
                .reader(reader)
                .processor(processor)
                .writer(writer)
                .taskExecutor(taskExecutor)
                .throttleLimit(batchStep.getThrottleLimit())
                .build();
```

En este ejemplo, se observa un proceso que tiene como variables de entrada (aquellas que desean ser leídas y procesadas) del *item* las mismas que las deseadas para su escritura, es decir, se desea que independientemente del tipo de `ItemWriter` se graben los valores de las mismas variables que de entrada).

Son los *bean* los que determinan, mediante su configuración, la manera de proceder con los datos haciendo uso de los POJOs.

Flow

Son procesos que funcionan como controladores del flujo de trabajo de los steps. De esta manera, los ´ `flow` pueden ser ejecutados como un *job* o como parte de uno, pudiendo conectar diferentes *steps* que dependen del estado de salida del *step* previo. Principalmente se utilizan estos procesos cuando se desean realizar tareas en paralelo incluyendo una determinada secuencia de *steps*.

De manera genérica, un *bean* de tipo *flow* se crea:

```
//Identificador e instancias

@Bean("flowParal")
    public Flow flowClientesProductoSalidaOrdenadoParalelo(
            @Qualifier("<IdentificadorBean1>") Step IdentificadorBean1,
            @Qualifier("<IdentificadorBean2>") Step IdentificadorBean2,
            @Qualifier("taskExecutor") TaskExecutor taskExecutor) {...}

// Creación de cada flow
Flow flow = new FlowBuilder<Flow>("<step-IdentificadorBean1>").from(<IdentificadorBean1>).end();


// Create the job
SimpleJobBuilder builder = new JobBuilder("flow").repository(jobRepository).start(step2);

// Create split providing an async task executor so the flows are executed in parallel
builder.split(new SimpleAsyncTaskExecutor()).add(flow).end();

// Build the job and execute it
builder.preventRestart().build().execute(execution);
```

1.12. Job

Es la representación de todos los procesos que se llevarán a cabo. Está compuesto por los diferentes *steps* que se deseen implementar en la ejecución. La interfaz `JobLauncher` es la encargada de lanzar el trabajo y configurarlo según una serie de métodos.

Un ejemplo de *bean* del *job* en la aplicación puede ser:

```
    public Job job(@Qualifier("bankiaJobCompletionNotificationListener") JobExecutionListenerSupport
 listener, 1
            @Qualifier("flow-startUp") Flow flowStartUp,
            @Qualifier("<identificadorStep1>") Step identificadorStep1,
            @Qualifier("<identificadorStep2>") Step identificadorStep2,
            @Qualifier("<identificadorFlow>") Flow identificadorFlow )
            {
        // @formatter:off
        return jobBuilderFactory.get(applicationName) 2
                .preventRestart()
                .incrementer(new RunIdIncrementer())
                .listener(listener)

                .start(flowStartUp) 1
                .next(identificadorStep1)
            //  .next(identificadorFlow)
                .next(identificadorStep2)

                .next(flowEnding)
                .end()

                .build();
            }
```

1 Se instancian todas las entidades *bean* necesarias para la creación del *job*. Esto incluirá todos los *steps* que se vayan a utilizar durante la ejecución, así como el *listener* del propio proceso de trabajo, propio de la aplicación.

2 Configuración de la aplicación para cualquier *job*. Se introducen tanto el nombre como distintas configuraciones para evitar la finalización del proceso si algún paso falla, así como un contador de procesos. Se recomienda no cambiar.

1 Se escribe todo el proceso que se realizará en el *job*.

1. `start(flowStartUp)`: Inicializa el trabajo. En la aplicación, éste se inicia con un flow predeterminado, que incluye una validación de seguridad para Bankia.
2. `next(step/flow)`: se introducen, en orden de ejecución, los *steps* y *flows* que formarán parte de la ejecución.
3. `end()`: indica la finalización del proceso.
4. `build()`: método para construir el `Job` según todos los parámetros anteriores. *****

1.13. Arquetipo

Para crear un proyecto de procesamiento por lotes se partirá del arquetipo disponible para este caso: `com.bankia.aq.batch:bankia-aq-batch-archetype:1.0.0.RELEASE`.

Los proyectos se podrán generar bien desde el propio IDE, seleccionando el arquetipo anteriormente indicando o bien por línea de comandos teniendo instalada la herramienta Maven, mediante el uso del siguiente comando:

```
mvn archetype:generate                                \
    -DarchetypeGroupId=com.bankia.aq.batch            \
    -DarchetypeArtifactId=bankia-aq-batch-archetype   \
    -DarchetypeVersion=1.0.0.RELEASE                  \
    -DgroupId=com.bankia.aq.prueba                    \ 1
    -DartifactId=bankia-aq-prueba-batch               \ 2
    -Dversion=0.0.1-SNAPSHOT                            1
```

1 Se indica el nombre del `groupId` del proyecto que se desea generar.
2 Se indica el nombre del `artifactId` del proyecto que se desea generar.
1 Se indica la versión del proyecto.

Capítulo 2. Casos de uso

En este apartado se detallan las distintas Pruebas de Concepto desarrolladas en el framework `Spring Batch` para el proyecto. El objetivo de cada una de ellas se puede ver en el apartado Solución técnica.

2.1. Caso de uso I (UseCase1) V-1.0

Configuración general para la UseCase1

Para la prueba, la configuración general del `application.yml` ha sido la siguiente:

```
bankia:
  batch:
    thread-pool: 1
      core-pool-size: 32
#      max-pool-size:
#      queue-capacity:
    step: 2
      clientes: 1
        chunk-size: 100 4
        throttle-limit: 20 5
      clientesProducto: 6
        chunk-size: 1000
        throttle-limit: 20
      clientesProductoSalida: 7
        chunk-size: 50
        throttle-limit: 20
    collection: bankiaCollection-${random.value} 8
    cache: 9
      clientes: clientes
```

1 `thread-pool:` Contiene la configuración referente a los hilos. En este caso, el `core-pool-size` se ha establecido en 32 hilos.

2 `step`: contiene la configuración de los distintos *steps* que contiene la UseCase.

1 6 7 `<identificadorYML>`: identifica el *step* al que se referencian las variables.

4 `chunk-size:` determina el tamaño de los *chunks* en cada uno de los *steps*. En este caso, para el *step* encargado del archivo Json se ha establecido en 100, para el fichero de texto plano en 1000 y para el de lectura desde MongoDB en 50.

5 `throttle-limit`: determina el número de hilos de ejecución simultáneos. En este caso, en todos los *steps* está definido a 20.

8 `collection:` define el nombre de la colección donde se grabarán los datos en MongoDB. Genera un valor aleatorio para cada proceso.

9 `cache`: Indica, para cada uno de los procesos que lo requieran, el nombre de la caché donde se almacenarán los datos procesados. En este caso, solo el *step* clientes que trata el Json graba datos temporalmente en caché.

Configuración y procesos específicos:

Para el caso de uso 1 (UseCase1) se explican a continuación los distintos pasos que ejecuta la aplicación, que se componen, en orden de ejecución, en el siguiente `job`:

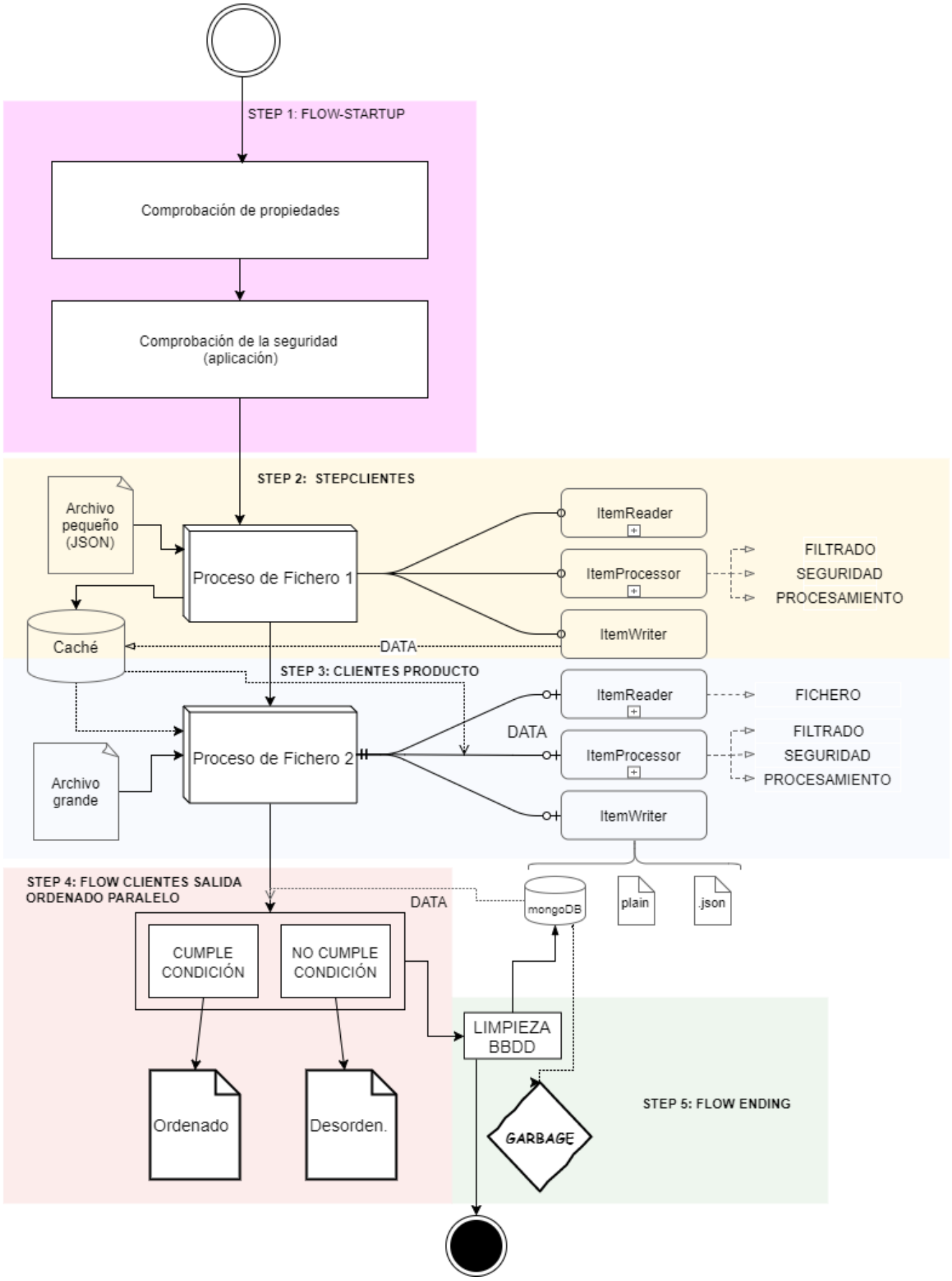
```
@Bean
    public Job job(@Qualifier("bankiaJobCompletionNotificationListener") JobExecutionListenerSupport
 listener,
            @Qualifier("flow-startUp") Flow flowStartUp, @Qualifier("stepClientes") Step stepClientes,
            @Qualifier("stepClientesProducto") Step stepClientesProducto,
```

```
            @Qualifier("flowClientesProductoSalidaOrdenadoParalelo") Flow
flowClientesProductoSalidaOrdenadoParalelo,
            @Qualifier("flow-ending") Flow flowEnding) {
        // @formatter:off
        return jobBuilderFactory.get(applicationName)
                .preventRestart()
                .incrementer(new RunIdIncrementer())
                .listener(listener)
                .start(flowStartUp) 1
                .next(stepClientes) 2
                .next(stepClientesProducto) 1
                .next(flowClientesProductoSalidaOrdenadoParalelo) 4
                .next(flowEnding) 5
                .end()
                .build();
        // @formatter:on
```

Cada uno de estos pasos cumple con alguno de los criterios funcionales solicitados. Éstos son:

1. `start(flow-startup)`: uso de una librería proporcionada por Bankia, comprobación de si el usuario tiene los permisos adecuados para ejecutar la funcionalidad en ese entorno.
2. `next(stepClientes)`: lectura de archivo Json, filtrado de la información por condición y grabado en caché de memoria.
1. `next(stepClientesProducto)`: Lectura de archivo de texto plano, de extensión csv, y combinación con información del Json en caché mediante campo común y expresión de condición. Grabación en BBDD (MongoDB). Generación discrecional de otros ficheros (Json y texto plano) según expresiones de condición.
4. `next(flowClientesProductoSalidaOrdenadoParalelo)`: lectura de datos de BBDD para ordenación, ascendente o descendente, según condición, a dos ficheros de texto plano.
5. `next(flowEnding)`: Eliminación de registros de la base de datos.

El diagrama del UseCase1 es el siguiente:

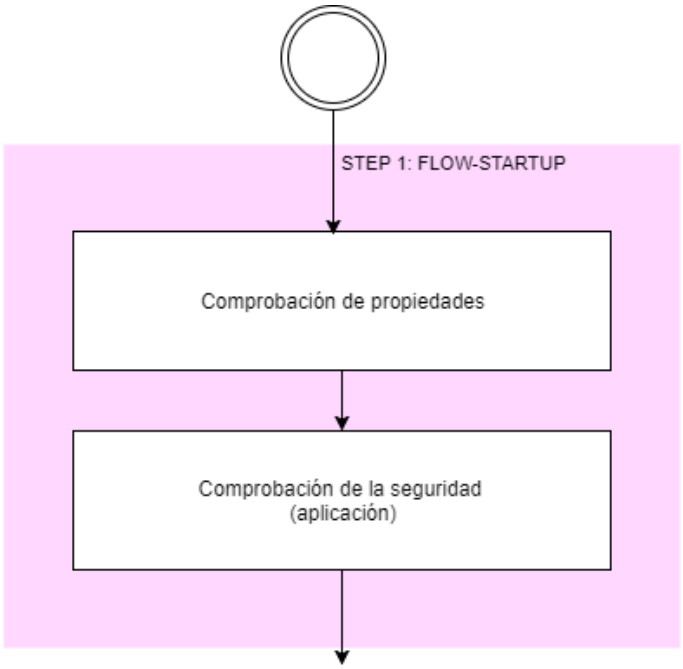

STEP 1: FLOW-STARTUP
Comprobación de propiedades
Comprobación de la seguridad (aplicación)
STEP 2: STEPCLIENTES
Archivo pequeño (JSON)
Proceso de Fichero 1
ItemReader
ItemProcessor
ItemWriter
FILTRADO
SEGURIDAD
PROCESAMIENTO
Caché
DATA
STEP 3: CLIENTES PRODUCTO
Archivo grande
Proceso de Fichero 2
DATA
ItemReader
ItemProcessor
ItemWriter
FICHERO
FILTRADO
SEGURIDAD
PROCESAMIENTO
STEP 4: FLOW CLIENTES SALIDA ORDENADO PARALELO
DATA
mongoDB
plain
.json
CUMPLE CONDICIÓN
NO CUMPLE CONDICIÓN
LIMPIEZA BBDD
Ordenado
Desorden.
GARBAGE
STEP 5: FLOW ENDING

STEP 1: FLOW-STARTUP

Inicio de la aplicación:

- Comprobación de la necesidad de ejecución de la aplicación y de sus propiedades.
- Comprobación de la configuración introducida en la aplicación.

La caso de uso se inicia con la ejecución del `flow` encargado de comprobar la existencia de las condiciones necesarias para iniciar la cadena de procesos.



STEP 2: CLIENTE

- Lectura del fichero pequeño y escritura sobre caché en memoria.

1. ItemReader: Lectura del archivo Json.
2. ItemProcessor:
   a. Filtrado según campos preconfigurados.
   b. Procesamiento de los datos.
3. ItemWriter: escritura de los archivos procesados en caché.

El `step` generado es:

```
@Bean("stepClientes")
    public Step stepClientes(@Qualifier("clientesReader") ItemReader<Cliente> reader,
            @Qualifier("clientesProcessor") ItemProcessor<Cliente, Cliente> processor,
            @Qualifier("clientesWriter") ItemWriter<Cliente> writer,
            @Qualifier("taskExecutor") TaskExecutor taskExecutor) {
        BatchStep batchStep = batchProperties.getBatchStep("clientes");
        // @formatter:off
        return stepBuilderFactory.get("stepClientes").<Cliente, Cliente> chunk(batchStep.getChunkSize())
                .reader(reader)
```

```
                .processor(processor)
                .writer(writer)
                .exceptionHandler(new BankiaExceptionHandler())
                .listener(new BankiaItemReaderListener<>())
                .listener(new BankiaChunkCountListener("clientes", 1000))
                .taskExecutor(taskExecutor)
                .throttleLimit(batchStep.getThrottleLimit())
                .build();
        // @formatter:on
```

La configuración del proceso se establece en el `application.yml` de la carpeta `profiles/bankia-aq-batch-pocs-poc1`, y para un archivo Json:

```
bankia:
    batch:
        input:
          files:
            clientes:

file-type: json
              location-type: local
              local-location: ${INPUT_LOCATION}/Clientes.json
              filter-expression: "indbaja == 'N'"
              key-fields-expression: "idCliente"
```

Los *bean* que forman su estructura batch (*reader*, *processor*, *writer*) son:

```
@Bean("clientesReader")
    public ItemReader<Cliente> clientesReader() {
        return new
 BankiaFlatFileItemReaderBuilder<Cliente>().batchProperties(batchProperties).input("clientes")
                .targetType(Cliente.class).transformer(new ClienteFromMapper()).build();
    }
```

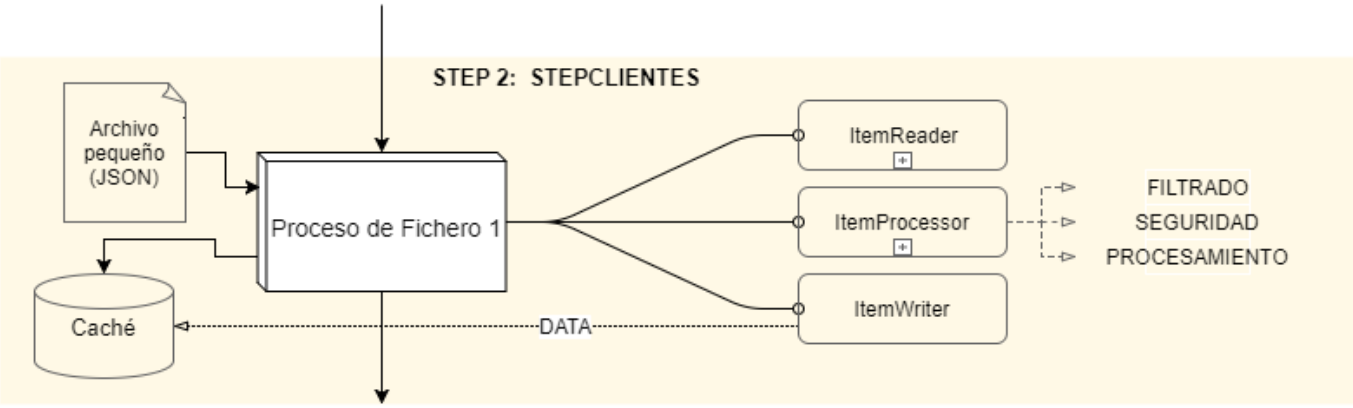
El *reader* genera un mapa del archivo Json, según lo definido en el `application.yml`, asociándolo con el POJO `Clientes.java`.

```
@Bean("clientesProcessor")
public ItemProcessor<Cliente, Cliente> clientesProcessor(SecurityValidation securityValidation) {
    BatchInputFile batchInputFile = batchProperties.getInput().getFiles().get("clientes");
    return new BankiaCompositeItemProcessorBuilder<Cliente, Cliente>()
            .filterProcessor(new BankiaFilterItemProcessor<>(batchInputFile.getFilterExpression())) (1)
            .processor(new ClienteItemProcessor()).build();
    }
```

(1) El *processor* realiza filtrados según expresiones contenidas en el `application.yml`; en este caso, según la condición `"filter-expresion: indbaja=='N'"`.

```
@Bean("clientesWriter")
public ItemWriter<Cliente> clientesWriter(CacheService cacheService) {
    BatchInputFile batchInputFile = batchProperties.getInput().getFiles().get("clientes");
    return new BankiaCacheItemWriter<>(cacheService, cacheClientes,
 batchInputFile.getKeyFieldsExpression());
}
```

El `writer` genera la escritura en caché, creando una estructura de mapa según la expresión `key-fields-expression:"idCliente"`.



STEP 3: CLIENTEPRODUCTO

- Lectura y procesos iniciales del fichero grande.

  1. ItemReader: Lectura del archivo de texto plano ordenado según posicionamiento fijo.
  2. ItemProcessor:
     a. Filtrado según datos almacenados en caché.
     b. Procesamiento de los datos, incluida la eliminación de espacios vacíos de los campos.
  3. `ItemWriter`: escritura de los archivos procesados en caché, según criterios:
     a. Generando un JSON.
     b. Generando un plaint text.
     c. Guardándose en BBDD (mongoDB).

El `step` generado es:

```
@Bean("stepClientesProducto")
    public Step stepClientesProducto(@Qualifier("clientesProductoReader") ItemReader<ClienteProducto>
 reader,
            @Qualifier("clientesProductoProcessor") ItemProcessor<ClienteProducto,
 ClienteProductoSalida> processor,
            @Qualifier("clientesProductoSalidaWriter") ItemWriter<ClienteProductoSalida> writer,
            @Qualifier("clientesProductoSalidaUsuario1Writer") FlatFileItemWriter<ClienteProductoSalida>
 clientesProductoSalidaUsuario1Writer,
            @Qualifier("clientesProductoSalidaUsuario2Writer") FlatFileItemWriter<ClienteProductoSalida>
 clientesProductoSalidaUsuario2Writer,
            @Qualifier("taskExecutor") TaskExecutor taskExecutor) {
        BatchStep batchStep = batchProperties.getBatchStep("clientesProducto");
        // @formatter:off
        return stepBuilderFactory.get("stepClientesProducto")
                .<ClienteProducto, ClienteProductoSalida> chunk(batchStep.getChunkSize())
                .reader(reader)
                .processor(processor)
                .writer(writer)
                .stream(clientesProductoSalidaUsuario1Writer)
                .stream(clientesProductoSalidaUsuario2Writer)
                .listener(new BankiaItemReaderListener<>())
                .listener(new BankiaChunkCountListener("clientesProducto", 10000))
                .taskExecutor(taskExecutor)
                .throttleLimit(batchStep.getThrottleLimit())
                .build();
```

Es un `step` complejo y que requiere de la creación de `FlatFileItemWriter`, necesarios para realizar la salida de los datos de manera simultánea tanto a base de datos como a dos ficheros de texto plano distintos. Esto se consigue instanciando los `bean` de escritura inyectados en su constructor mediante el método `stream`, previa creación de un ` ItemWriter` que permita procesos paralelos según un mapa (uso de classifier).

La configuración del proceso se establece en el `application.yml` de la carpeta `profiles/bankia-aq-batch-pocs-poc1`, y se ha definido para los input:

```
bankia:
    batch:
      input:
        files:
          clientes-producto:
              file-type: plain-text
              format-type: position
              strict: false
              location-type: local
              local-location: ${INPUT_LOCATION}/ClientesProducto.txt
    #         lines-to-skip: 100
              max-item-count: 10000
              fields:
                - idCliente
                - idProducto
                - descripcionProducto
                - ssa
              positions:
                - 1-36
                - 37-72
                - 73-133
                - 134
              filter-expression: "ssa == 'A184762'"
              key-fields-expression: "idCliente"
```

A su vez, para los distintos output de fichero:

```
    output:
      files:
        cliente-producto-salida-usuario1:
          file-type: plain-text
          format-type: delimited
          delimiter: ";"
          location-type: local
          local-location: ${OUTPUT_LOCATION}/cliente-producto-salida-usuario1.csv
          fields:
            - idCliente
            - nombre
            - primerApellido
            - segundoApellido
            - direccion
            - ssaCliente
            - idProducto
            - descripcionProducto
            - ssaProducto

        cliente-producto-salida-usuario2:
          file-type: plain-text
          format-type: position
          format: "%-36s%-36s%-61s%-7s"
          fields:
            - idCliente
            - idProducto
            - descripcionProducto
            - ssaProducto
          location-type: local
          local-location: ${OUTPUT_LOCATION}/cliente-producto-salida-usuario2.txt
```

Los *bean* que forman su estructura batch (*reader*, *processor*, *writer*) son:

Reader.

```
@Bean("clientesProductoReader")
    public ItemReader<ClienteProducto> clientesProductoReader() {
        return new BankiaFlatFileItemReaderBuilder<ClienteProducto>().batchProperties(batchProperties)
                .input("clientes-producto").targetType(ClienteProducto.class).build();
```

El *reader*, según lo definido en el `application.yml`, asocia las variables indicadas en `fields` con sus respectivas posiciones (indicadas en `positions`) con el POJO `ClienteProducto`.

Processor.

```
@Bean("clientesProductoProcessor")
   public ItemProcessor<ClienteProducto, ClienteProductoSalida> clientesProductoProcessor(CacheService
cacheService,
   ClienteProductoMapper mapper) {
       BatchInputFile batchInputFile = batchProperties.getInput().getFiles().get("clientes-producto");
       return new BankiaCompositeItemProcessorBuilder<ClienteProducto, ClienteProductoSalida>() (1)
               .filterProcessor(new BankiaCacheFilterItemProcessor<Cliente,
ClienteProducto>(cacheService,
                       cacheClientes, batchInputFile.getFilterExpression(), "cliente",
"clienteProducto")) (2)
               .processor(new ClienteProductoItemProcessor(cacheService, cacheClientes,
mapper)).build();
```

El *processor* realiza la fusión de los datos contenidos en caché (provenientes del tratamiento del Json) con los generados por el *reader* del archivo de posicionamiento:

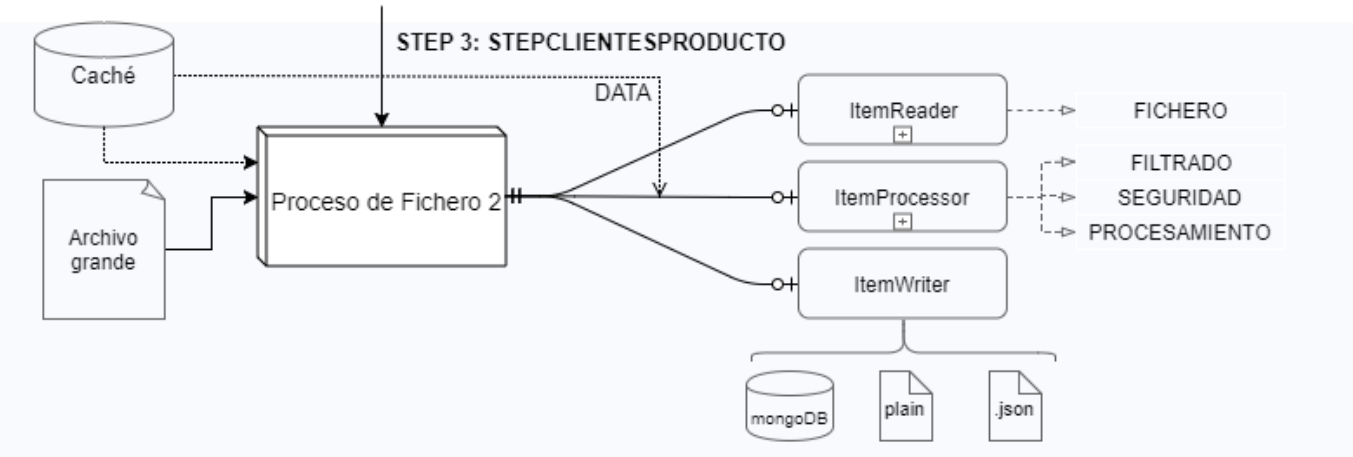
1. Los datos obtenidos se contienen en un POJO ClienteProductoSalida que contiene los parámetros de ambos archivos, haciendo uso de la clase `BankiaCompositeItemProcessorBuilder`.
2. Se establece una expresión para filtrar los datos, en este caso, `filter-expression: "ssa == 'A184762'"`.

Writer.

```
@Bean("clientesProductoSalidaWriter")
    public ItemWriter<ClienteProductoSalida> clientesProductoSalidaWriter(
            @Qualifier("clientesProductoSalidaUsuario1Writer") ItemWriter<ClienteProductoSalida>
clientesProductoSalidaUsuario1Writer,
            @Qualifier("clientesProductoSalidaUsuario2Writer") ItemWriter<ClienteProductoSalida>
clientesProductoSalidaUsuario2Writer,
            MongoOperations template) {
        MongoItemWriter<ClienteProductoSalida> mongoItemWriter = new
MongoItemWriterBuilder<ClienteProductoSalida>()
                .collection(mongoProperties.getCollection()).template(template).build();

        String classifierExpression = batchProperties.getOutput().getClassifier().get("filtrado"); (1)

        Map<String, ItemWriter<? super ClienteProductoSalida>> matcherMap = new HashMap<>(); (2)
        matcherMap.put("usuario1", clientesProductoSalidaUsuario1Writer);
        matcherMap.put("usuario2", clientesProductoSalidaUsuario2Writer);
        matcherMap.put("resto", mongoItemWriter);

        BackToBackPatternClassifier<ClienteProductoSalida, ItemWriter<? super ClienteProductoSalida>>
classifier = new BackToBackPatternClassifier<>(); (1)
        classifier.setRouterDelegate(new
BankiaOutputClassifier<ClienteProductoSalida>(classifierExpression));
        classifier.setMatcherMap(matcherMap);

        ClassifierCompositeItemWriter<ClienteProductoSalida> result = new
ClassifierCompositeItemWriter<>();
        result.setClassifier(classifier);
```

```
        return result;
```

1. El *writer* obtiene la expresión `filtrado` contenida en los output del `application.yml`, y que genera la condición para que los datos sean procesados por uno de los *Writer* del *Map*.
2. Se especifica que `ItemWriter` procesará cada serie de datos, siendo `usuario1`, `usuario2` o `resto` las expresiones devueltas por la expresión de `filtrado` introducida previamente.
1. Se genera el classifier necesario para crear la clase que generará el `ItemWriter`, haciendo uso de la expresión condicional obtenida de la configuración en el punto 1 y de la estructura de mapa creada en el punto 2.



STEP 4: FLOW CLIENTEPRODUCTOSALIDAORDENADOPARALELO

- Generación de documentación según datos contenidos en la BBDD:

1. `ItemReader`: Lectura de los datos desde mongoDB, de manera ordenada y según expresión, distinguiendo aquellos que se almacenan para su orden ascendente y aquellos que lo hacen para el fichero ordenado de manera descendente.

2. `ItemProcessor`: en este caso no se requieren procesos.

3. `ItemWriter`: escritura de los datos procesados en ficheros de texto plano, ordenados de forma ascendente en un fichero de tipo `csv` y de forma descendente en un `json`.

En este caso, se generan dos steps, uno para cada tipo de archivo de salida, teniendo para el fichero ordenado de manera ascendente:

```
@Bean("stepClientesProductoSalidaAsc")
    public Step stepClientesProductoSalidaAsc(
            @Qualifier("clientesProductoSalidaAscReader") ItemReader<ClienteProductoSalida> reader,
            @Qualifier("clientesProductoSalidaProcessor") ItemProcessor<ClienteProductoSalida,
 ClienteProductoSalida> processor,
            @Qualifier("clientesProductoSalidaAscWriter") ItemWriter<ClienteProductoSalida> writer,
            @Qualifier("taskExecutor") TaskExecutor taskExecutor) {
        BatchStep batchStep = batchProperties.getBatchStep("clientesProductoSalida");
        // @formatter:off
        return stepBuilderFactory.get("stepClientesProductoSalidaAsc")
                .<ClienteProductoSalida, ClienteProductoSalida> chunk(batchStep.getChunkSize())
                .reader(reader)
                .processor(processor)
                .writer(writer)
                .taskExecutor(taskExecutor)
                .throttleLimit(batchStep.getThrottleLimit())
                .build();
```

Para el archivo ordenado de manera descendente:

```
@Bean("stepClientesProductoSalidaDesc")
    public Step stepClientesProductoSalidaDesc(
            @Qualifier("clientesProductoSalidaDescReader") ItemReader<ClienteProductoSalida> reader,
            @Qualifier("clientesProductoSalidaProcessor") ItemProcessor<ClienteProductoSalida,
 ClienteProductoSalida> processor,
            @Qualifier("clientesProductoSalidaDescWriter") ItemWriter<ClienteProductoSalida> writer,
            @Qualifier("taskExecutor") TaskExecutor taskExecutor) {
        BatchStep batchStep = batchProperties.getBatchStep("clientesProductoSalida");
        // @formatter:off
        return stepBuilderFactory.get("stepClientesProductoSalidaDesc")
                .<ClienteProductoSalida, ClienteProductoSalida> chunk(batchStep.getChunkSize())
                .reader(reader)
                .processor(processor)
                .writer(writer)
                .taskExecutor(taskExecutor)
                .throttleLimit(batchStep.getThrottleLimit())
                .build();
```

Para que ambos procesos se lleven a cabo de forma paralela, se ha generado un *flow* que integra ambos pasos:

```
    @Bean("flowClientesProductoSalidaOrdenadoParalelo")
    public Flow flowClientesProductoSalidaOrdenadoParalelo(
            @Qualifier("stepClientesProductoSalidaAsc") Step stepClientesProductoSalidaAsc,
            @Qualifier("stepClientesProductoSalidaDesc") Step stepClientesProductoSalidaDesc,
            @Qualifier("taskExecutor") TaskExecutor taskExecutor) {
        Flow flowOutputMongoAsc = new FlowBuilder<Flow>("flow-stepClientesProductoSalidaAsc")
                .from(stepClientesProductoSalidaAsc).end();
        Flow flowOutputMongoDesc = new FlowBuilder<Flow>("flow-stepClientesProductoSalidaDesc")
                .from(stepClientesProductoSalidaDesc).end();
        return new FlowBuilder<Flow>("flow-stepClientesProductoSalidaOrdenada").split(taskExecutor)
                .add(flowOutputMongoAsc, flowOutputMongoDesc).build();
```

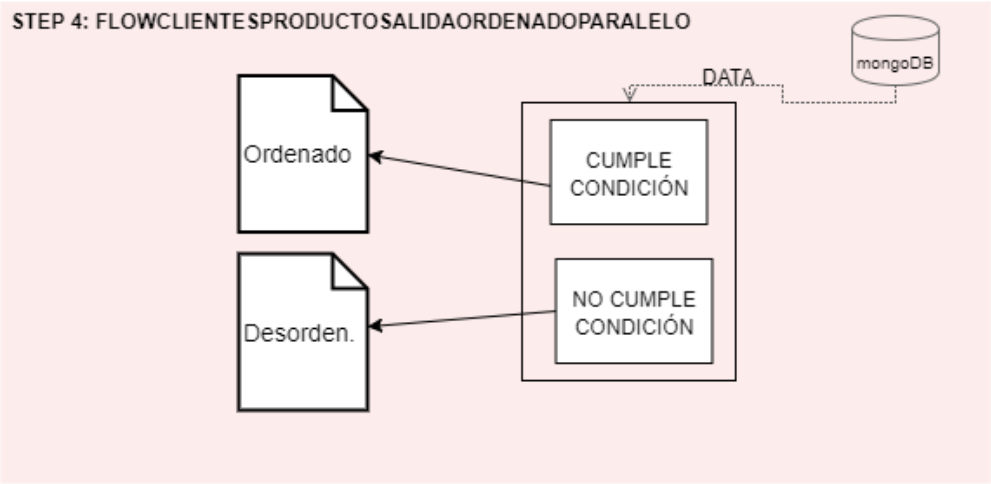
La configuración del `application.yml` para la obtención de los datos desde mongoDB es:

```
        cliente-producto-salida-asc:
          json-query: "{idCliente : /?0/}"
          parameters:
            - "-f"
          sorts:
            idProducto: ASC

        cliente-producto-salida-desc:
          json-query: "{idCliente : { $regex : /^((?!?0).)*$/}}"
          sorts:
            idProducto: DESC
```

El tipo de salida de los ficheros están configurados a su vez en el `application.yml`, en la sección perteneciente a los `output`:

```
bankia:
  batch:
   output:
    cliente-producto-salida-sorted-asc-delimited:
              file-type: plain-text
              format-type: delimited
              delimiter: ";"
              location-type: local
              local-location: ${OUTPUT_LOCATION}/cliente-producto-salida-sorted-asc.csv
              fields:
                - idCliente
                - nombre
                - primerApellido
                - segundoApellido
                - direccion
```

```
      - ssaCliente
      - idProducto
      - descripcionProducto
      - ssaProducto

  cliente-producto-salida-sorted-desc-json:
    file-type: json
    location-type: local
    local-location: ${OUTPUT_LOCATION}/cliente-producto-salida-sorted-desc.json
    fields:
      - idCliente
      - nombre
      - primerApellido
      - segundoApellido
      - direccion
      - ssaCliente
      - idProducto
      - descripcionProducto
      - ssaProducto
```



STEP 5: FLOW ENDING

- Eliminación de los archivos provisionalmente almacenados en la base de datos.

2.2. Caso de uso I (UseCase1) v-2.0

Los principales cambios respecto a la versión 1.0 son:

- Eliminación del uso de MongoDB para la ordenación de los ficheros, realizándose mediante fraccionamiento en subficheros de los documentos a ordenar.
- Eliminación de la expresión multicondicional de filtrado en el *merge* de fichero grande y pequeño, realizando dos expresiones simples de manera secuencial.
- Creación de un *bean* de `ExpressionParser` en el `BatchConfiguration` que permita un solo objeto con múltiples instancias, aligerando carga de trabajo para el procesador.

Nota

Estos cambios reducen de manera notoria el tiempo de ejecución de la aplicación, y requieren de algunos cambios tanto a nivel de código como a nivel de configuración (en el `application.yml`).

Cambios en la configuración y procesos para la Versión 2.0

El código que conforma el *job* ha sido modificado a fin de poder ordenar los ficheros mediante un nuevo *step* con un sistema de fraccionamiento, teniendo:

```
 @Bean
   public Job job(@Qualifier("bankiaJobCompletionNotificationListener") JobExecutionListenerSupport
listener,
           @Qualifier("flow-startUp") Flow flowStartUp, @Qualifier("stepClientes") Step stepClientes,
           @Qualifier("stepClientesProducto") Step stepClientesProducto, @Qualifier("stepOrdenar") Step
stepOrdenar, //* Inyección del step
           @Qualifier("flowClientesProductoSalidaOrdenadoParalelo") Flow
flowClientesProductoSalidaOrdenadoParalelo,
           @Qualifier("flow-ending") Flow flowEnding) {
       // @formatter:off
       return jobBuilderFactory.get(applicationName)
               .preventRestart()
               .incrementer(new RunIdIncrementer())
               .listener(listener)
               .start(flowStartUp)
               .next(stepClientes)
               .next(stepClientesProducto)
               .next(stepOrdenar) //* Nuevo step de ordenación
               .next(flowClientesProductoSalidaOrdenadoParalelo)
               .next(flowEnding)
               .end()
               .build();
```

NEW STEP: ORDENAR

Este *step* se ha creado en el `BatchProperties`,siendo su bean:

```
  @Bean("stepOrdenar")
   public Step stepOrdenar(
           @Qualifier("clientesProductoSalidaUsuario1SorterTasklet") Tasklet
clientesProductoSalidaUsuario1SorterTasklet, ❶
           @Qualifier("clientesProductoSalidaUsuario2SorterTasklet") Tasklet
clientesProductoSalidaUsuario2SorterTasklet, ❷
           @Qualifier("taskExecutor") TaskExecutor taskExecutor) {

       return stepBuilderFactory.get("stepOrdenar") ❶
           .tasklet(clientesProductoSalidaUsuario1SorterTasklet)
           .tasklet(clientesProductoSalidaUsuario2SorterTasklet)
           .taskExecutor(taskExecutor)
           .build();
   }
```

❶ Inyección del *tasklet* que ordena el primer fichero.
❷ Inyección del *tasklet* que ordena el segundo fichero.
❶ Utilización del `stepBuilderFactory` para crear el *step*.

Los dos bean inyectados para la ordenación hacen referencia a los siguientes:

```
  @Bean("clientesProductoSalidaUsuario1SorterTasklet")
   public Tasklet clientesProductoSalidaUsuario1SorterTasklet() {
       BatchOutputFile output = batchProperties.getOutput().getFiles().get("cliente-producto-salida-
usuario1"); ❶
       return new BankiaSortFileTasklet(output.getTemporalLocation(), output.getLocalLocation(), ❷
               (String u1, String u2) -> { ❶
                   String[] t1 = u1.split(output.getDelimiter(), -1);
                   String[] t2 = u2.split(output.getDelimiter(), -1);
                   return t1[0].compareTo(t2[0]);
               });
   }
```

En este caso, el fichero a ordenar es un fichero de texto plano separado por delimitador caracter:

1. Se integran las variables del archivo de configuración referentes al bean.
2. Se crea una nueva instancia de `BankiaSortFileTasklet` y se introducen las variables de directorio local y directorio local temporal donde se realiza el proceso. Esta clase, de la librería creada para Bankia al efecto, configura y hace uso de una librería que fracciona y ordena el fichero de texto indicado por las propiedades.
1. Se realiza un comparador de cadena de texto para el fichero de texto, especificando el delimitador (información que obtiene del `yml`, en este caso ";") e indicando por cual de los campos se desea ordenar, en este caso el primero.

Para el segundo fichero:

```
    @Bean("clientesProductoSalidaUsuario2SorterTasklet")
    public Tasklet clientesProductoSalidaUsuario2SorterTasklet() {
        BatchOutputFile output = batchProperties.getOutput().getFiles().get("cliente-producto-salida-
usuario2");
        return new BankiaSortFileTasklet(output.getTemporalLocation(), output.getLocalLocation(),
                (String u1, String u2) -> { 1
                    String t1 = u1.substring(10, 15);
                    String t2 = u2.substring(10, 15);
                    return t2.compareTo(t1);
                });
```

En este caso, el fichero a ordenar es un fichero de texto plano separado por posiciones fijas:

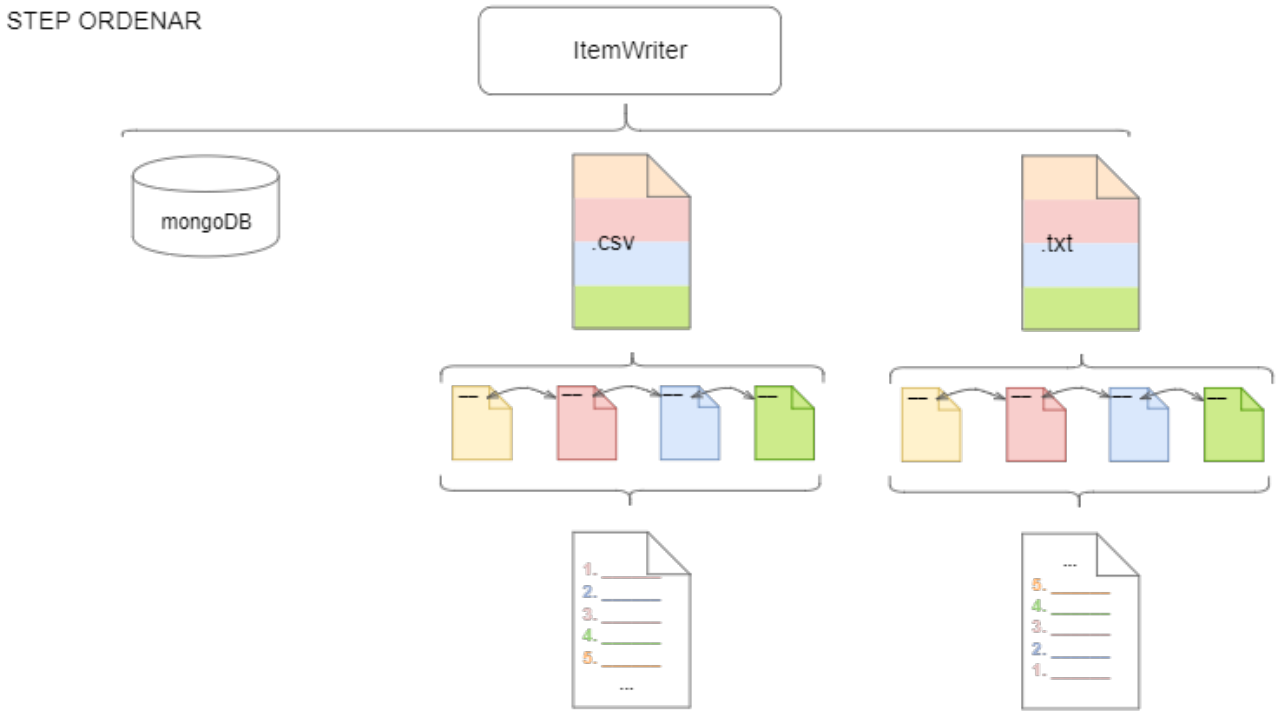
1. Como única diferencia, el comparador de cadena de texto se realiza mediante posiciones, indicándole el rango de los *char* que se desea ordenar, en este caso, el intervalo que coincide con las posiciones 10-14 (el 15 no es inclusivo).

Las variables de las que depende este proceso están en el `application.yml`:

```
output:

  files:

    cliente-producto-salida-usuario1:

      delimiter: ";"
      location-type: local
      local-location: ${OUTPUT_LOCATION}/cliente-producto-salida-usuario1.csv
      temporal-location: ${OUTPUT_LOCATION}/cliente-producto-salida-usuario1-temp.csv
      remove-temporal: true


    cliente-producto-salida-usuario2:

      location-type: local
      local-location: ${OUTPUT_LOCATION}/cliente-producto-salida-usuario2.txt
      temporal-location: ${OUTPUT_LOCATION}/cliente-producto-salida-usuario2-temp.txt
      remove-temporal: true
```

- `cliente-producto-salida-usuario:` identificador para la llamada desde el BatchConfiguration.
  - `delimiter:` indica la cadena utilizada como separador para el fichero de csv.
  - `location-type:` indica el tipo de directorio donde trabaja el proceso. En este caso se ha trabajado en local.

- `local-location:` indica la ruta donde se situa del archivo tras realizar la ordenación. El archivo procesado se guarda en nuestro caso en la carpeta Data/Output con el nombre especificado.
- `temporal-location:` indica la ruta donde se emplazarán los archivos temporales creados para el proceso así como el nombre de los mismos.
- `remove-temporal:` condicional para activar o desactivar el borrado de los archivos temporales tras el proceso.



Otras variaciones del código

Para la creación del intérprete de las expresiones *SpEL* como un único *bean* inyectado se ha creado, en el `BatchProperties`:

```
@Bean
    ExpressionParser expressionParser() {
        return new SpelExpressionParser();
```

En la creación de los distintos *bean* que hacen uso de expresiones regulares se ha inyectado este bean.

La expresión regular que establecía múltiples condiciones para el filtrado en el archivo de texto grande (en este caso ordenado por posiciones fijas) ha sido sustituida por dos expresiones regulares monocondicionales y separadas, quedando:

```
input:
      files:
          clientes-producto:
          filter-expression: "ssa == 'A184762'"
          key-fields-expression: "idCliente"
```

2.3. Caso de uso II (UseCase2)

Configuración del archivo .yml

Para la prueba actual, la configuración del `application.yml` ha sido la siguiente:

```
bankia:
  batch:
    thread-pool: 1
      core-pool-size: 1
#      max-pool-size:
#      queue-capacity:
    step: 2
      ftpinput: 1
        chunk-size: 10000 4
        throttle-limit: 32 5

    cache: 6
      clientes: ficheroClientesFtp
```

1 `thread-pool:` controla el número de hilos de la aplicación. Debido a los contadores que implementa la aplicación, se ha configurado con un valor de 1. Tanto para la cantidad máxima de hilos (`max-pool-size`) como la capacidad de la cola (`queue-capacity`) se ha adoptado su valor por defecto al estar comentados.

2 `step`: contiene la configuración de los distintos *steps* de procesos *Batch* que contiene la UseCase.

1 `<identificadorYML>`: identifica el *step* al que se referencian las variables.

4 `chunk-size:` determina el tamaño de los *chunks* en cada uno de los *steps*. En este caso, para el *step* encargado del archivo Json se ha establecido en 1000.

5 `throttle-limit:` determina el número de hilos de ejecución simultáneos. En este caso está definido a 32.

6 `cache`: Indica, para cada uno de los procesos que lo requieran, el nombre de la caché donde se almacenarán los datos procesados. En este caso, solo el *step* ftpInput que trata el Json graba datos temporalmente en caché.

Configuración y procesos específicos de la UseCase2

Para el caso de uso 2 (UseCase2) se explican a continuación los distintos pasos que ejecuta la aplicación, que se componen, en orden de ejecución, en el siguiente `job`:

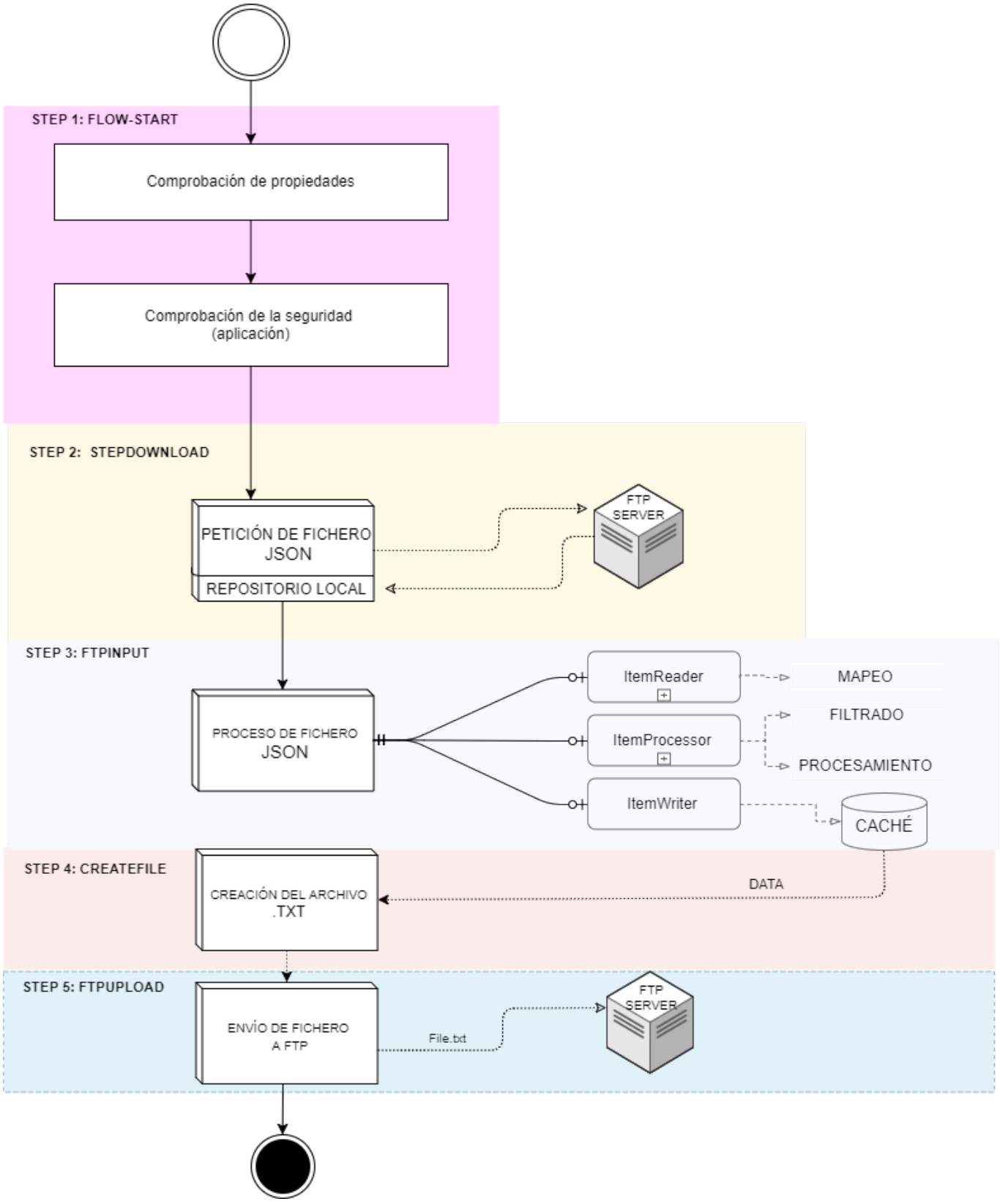
```
@Bean
public Job jobPoc2(@Qualifier("bankiaJobCompletionNotificationListener") JobExecutionListenerSupport
 listener,
          @Qualifier("flow-startUp") Flow flowStartUp,
    @Qualifier("stepDownload") Step stepDownload,
          @Qualifier("stepFtpInput") Step stepFtpInput,
          @Qualifier("stepCreateFile") Step stepCreateFile,
          @Qualifier("stepUpload") Step stepUpload,
          @Qualifier("flow-ending") Flow flowEnding)
          {
    // @formatter:off
    return this.jobBuilderFactory.get(applicationName)
              .preventRestart()
              .incrementer(new RunIdIncrementer())
              .listener(listener)
              .start(flowStartUp) 1
              .next(stepDownload) 2
              .next(stepFtpInput) 1
              .next(stepCreateFile) 4
              .next(stepUpload) 5
```

```
                .end()
            .build();
}
```

Cada uno de estos pasos cumple con alguno de los criterios funcionales solicitados. Éstos son:

1. `start(flow-startup)`: uso de una librería proporcionada por Bankia, comprobación de si el usuario tiene los permisos adecuados para ejecutar la funcionalidad en ese entorno.
2. `next(stepDownload)`: configuración del servidor ftp y petición de descarga de archivo Json. Grabado en directorio local.
1. `next(stepFtpInput)`: mapeo de fichero Json a POJO y grabado en caché de los contadores de coincidencias por campo.
4. `next(stepCreateFile)`: generación de fichero de texto plano con el identificador de coincidencia y el número de las mismas.
5. `next(stepUpload)`: subida al servidor del fichero procesado que contiene el número de coincidencias de la variable del Json especificada en el archivo de configuración.

El diagrama de la UseCase2 es el siguiente:



STEP 1: FLOW-START

Inicio de la aplicación:

- Comprobación de la necesidad de ejecución de la aplicación y de sus propiedades.

- Comprobación de la configuración introducida en la aplicación.

STEP 2: STEPDOWNLOAD

- Creación del canal de comunicación con el servidor FTP.
- Petición del archivo y descarga a respositorio local.

El `step` generado es:

```
    @Bean("stepDownload")
    public Step stepDownload(@Qualifier("downLoadFileTasklet") DownLoadFileTasklet downLoadFileTasklet)
{
        return this.stepBuilderFactory.get("stepDownload")
                                .tasklet(downLoadFileTasklet) 1
                                .build();
    }
```

1 El step se basa en un tasklet que, tras la configuración de los parámetros del servidor FTP, llama a éste y obtiene el fichero indicado.

La configuración del servidor y del fichero se realizan mediante llamadas a los campos contenidos en el `application. yml`. Éste, a su vez, contiene las siguientes variables:

```
bankia:
    batch:
        ftp:
          session:
            host: 127.0.0.1
            clientMode: 0
            fileType: 0
            port: 21
            username: root
            password: 1234

          download:
            retryIfNotFound: true
            downloadFileAttempts: 1
            rtryIntervalMilliseconds: 10000
            fileNamePattern: Clientes.json
            remoteDirectory: /
            localDirectory: ${INPUT_LOCATION}/
            sftp: false
```

- `session:` contiene la lista de parámetros para la configuración de acceso al servidor.
  - `host:` nombre del host donde se ubica el servidor.
  - `clientMode:` constante que en su valor 0 indica que es el servidor es el que debe conectarse al puerto de datos del cliente.
  - `fileType:` constante que indica el tipo de compilación del archivo, en este caso el 0 corresponde con `ASCII_FILE_TYPE`.
  - `port:` indica el puerto que establece la conexión con el servidor FTP, para la UseCase, el puerto 21.
  - `username:` usuario de la sesión al que accede el servicio.
  - `password:` contraseña de la sesión.

- `download:` contiene la lista de las variables para la configuración de la descarga desde servidor FTP.
  - `retryIfNotFound:` especifica si debe reenviarse la petición al servidor si el archivo no se ha encontrado. En este caso está activado mediante el valor `true`.
  - `downloadFileAttempts:` indica el número de intentos de descarga del archivo. En el caso actual, 1 intento.
  - `rtryIntervalMilliseconds:` contiene el tiempo de espera hasta la siguiente petición de descarga del archivo, en caso de configurar más de un intento. El valor en este caso es de 10000 ms.
  - `fileNamePattern:` contiene el nombre del archivo que se desea descargar el servidor. Para la prueba actual, `clientes.json`.
  - `remoteDirectory:` especifica la ruta en la cual se encuentra el archivo dentro del servidor remoto.
  - `sftp:` indica si el la transferencia se realiza mediante el protocolo sftp. En esta prueba está deshabilitado mediante el indicador `false`.

- El *bean* que forma la estructura de su *step* es un tasklet configurado en el archivo `BatchProperties`:

```
@Bean("downLoadFileTasklet")
    public DownLoadFileTasklet downLoadFileTasklet(@Qualifier("bankiaFtpSessionFactory") SessionFactory
 bankiaFtpSessionFactory) {

        DownLoadFileTasklet  ftpTasklet = new DownLoadFileTasklet(); 1
        ftpTasklet.setRetryIfNotFound(batchProperties.getFtp().getDownload().getRetryIfNotFound());

 ftpTasklet.setDownloadFileAttempts(batchProperties.getFtp().getDownload().getDownloadFileAttempts());

 ftpTasklet.setRetryIntervalMilliseconds(batchProperties.getFtp().getDownload().getRtryIntervalMilliseconds());
        ftpTasklet.setFileNamePattern(batchProperties.getFtp().getDownload().getFileNamePattern());
        ftpTasklet.setRemoteDirectory(batchProperties.getFtp().getDownload().getRemoteDirectory());
        ftpTasklet.setLocalDirectory(new
 File(batchProperties.getFtp().getDownload().getLocalDirectory()));
        ftpTasklet.setSftp(batchProperties.getFtp().getDownload().getSftp());
        ftpTasklet.setSessionFactory(bankiaFtpSessionFactory);

        return ftpTasklet;

    }
```

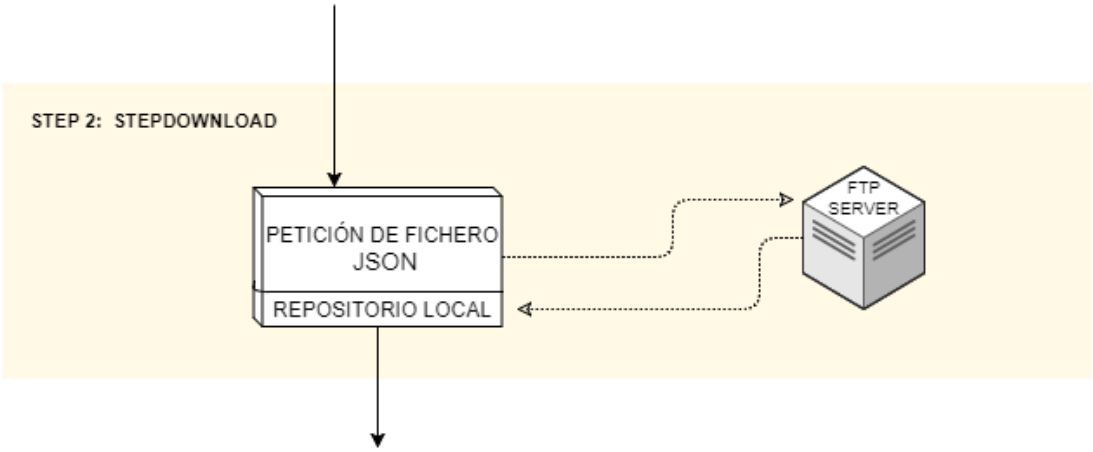
1 Se introducen las variables referentes a la petición del fichero una vez creado el acceso al servidor, indicando a su vez el repositorio local donde se graba.

En el constructor se le inyecta la configuración de la sesión en el constructor, siendo éste:

```
    @Bean("bankiaFtpSessionFactory")
    public SessionFactory bankiaFtpSessionFactory()
    {
        DefaultFtpSessionFactory ftpSessionFactory = new DefaultFtpSessionFactory();
        ftpSessionFactory.setHost(batchProperties.getFtp().getSession().getHost());
        ftpSessionFactory.setClientMode(batchProperties.getFtp().getSession().getClientMode());
        ftpSessionFactory.setFileType(batchProperties.getFtp().getSession().getFileType());
        ftpSessionFactory.setPort(batchProperties.getFtp().getSession().getPort());
        ftpSessionFactory.setUsername(batchProperties.getFtp().getSession().getUsername());
        ftpSessionFactory.setPassword(batchProperties.getFtp().getSession().getPassword());

        return ftpSessionFactory;
```

```
    }
```

Con este *bean* se configura la sesión de acceso, que recoge las variables contenidas en el `application.yml`.



STEP 3: STEPFTPINPUT

- Lectura del fichero pequeño y escritura sobre caché en memoria de conteo de coincidencias en campo.

  1. ItemReader: Lectura del archivo Json y mapeo a POJO.
  2. ItemProcessor:
     a. Filtrado según campos preconfigurados.
     b. Filtrado según parámetros de seguridad.
  3. ItemWriter: conteo de coincidencias de los datos leídos y escritura de los mismos en caché según variables de campo.

El `step` generado es:

```
@Bean("stepFtpInput")
    public Step stepFtpInput(@Qualifier("ftpInputReader") ItemReader<Cliente> reader,
      @Qualifier("ftpInputProcessor") ItemProcessor<Cliente,Cliente> processor,
      @Qualifier("ftpInputWriter") ItemWriter<Cliente> writer,
            @Qualifier("taskExecutor") TaskExecutor taskExecutor) {
         BatchStep batchStep = batchProperties.getBatchStep("ftpinput");

     return this.stepBuilderFactory.get("stepFtpInput")
       .<Cliente,Cliente> chunk(batchStep.getChunkSize())
       .reader(reader)
       .processor(processor)
       .writer(writer)
       .exceptionHandler(new BankiaExceptionHandler())
                .listener(new BankiaItemReaderListener<>())
                .listener(new BankiaChunkCountListener("ftpinput", 1000))
                .taskExecutor(taskExecutor)
                .throttleLimit(batchStep.getThrottleLimit())
                .build();
    }
```

El *step* sigue la configuración estándar de un proceso *Batch*, teniendo *reader*, *processor* y *writer* creados mediante sus respectivos *builders* e inyectados en el *step*.

La configuración del proceso se establece en el `application.yml` de la carpeta `profiles/bankia-aq-batch-pocs-poc1`, y para un archivo Json:

```
bankia:
    batch:
        input:
          files:

            clientes:
              file-type: json
              location-type: local
              local-location: ${INPUT_LOCATION}/Clientes.json
              filter-expression: "indbaja == 'N'"
              key-fields-expression: "idCliente"
```

- `clientes:` identificador para la configuración.
  - `file-type:` especifica el tipo de fichero que se va a procesar, en este caso Json.
  - `local-location:` ubica el archivo dentro del repositorio local.
  - `filter-expression:` filtrado que se realiza durante el *processor*, antes del trabajo de conteo de coincidencias de la UseCase. `indbaja == 'N'` ha sido la expresión utilizada esta vez, eliminando aquellos campos que no posean el valor `N` en el campo `indbaja`.
  - `key-fields-expression:` indicador del campo sobre el que se realizará el conteo de coincidencias. Se ha configurado el campo `idCliente` para la prueba.

Los *bean* que forman su estructura batch (*reader*, *processor*, *writer*) son:

```
@Bean("ftpInputReader")
    public ItemReader<Cliente> ftpInputReader() {
        return new
BankiaFlatFileItemReaderBuilder<Cliente>().batchProperties(batchProperties).input("ftpinput")
                .targetType(Cliente.class).transformer(new ClienteFromMapper()).build();
    }
```

El *reader* genera un mapa del archivo Json, según lo definido en el `application.yml`, asociándolo con el POJO `Clientes.java`.

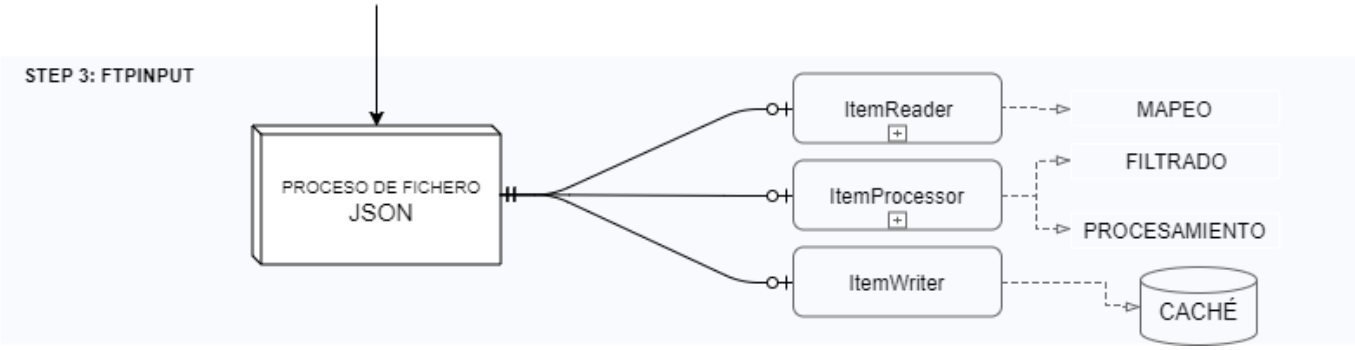
```
@Bean("ftpInputProcessor")
    public ItemProcessor<Cliente, Cliente> ftpInputProcessor(SecurityValidation securityValidation) {
        BatchInputFile batchInputFile = batchProperties.getInput().getFiles().get("ftpinput");
        return new BankiaCompositeItemProcessorBuilder<Cliente, Cliente>()
                .filterProcessor(new BankiaFilterItemProcessor<>(batchInputFile.getFilterExpression()))
                .processor(new ClienteItemProcessor()).build();
    }
```

El *processor* realiza dos filtrados según expresiones contenidas en el `application.yml`:

> Según condición, `"filter-expresion: indbaja=='N'"`.
> Según unas condiciones de seguridad, `field: "ssa" function: "FFFADADF"`.

```
@Bean("ftpInputWriter")
    public ItemWriter<Cliente> ftpInputWriter(CacheService cacheService) {
        BatchInputFile batchInputFile = batchProperties.getInput().getFiles().get("ftpinput");

        return new BankiaCacheItemCounterWriter<>(cacheService, cacheFtpClientes,
batchInputFile.getKeyFieldsExpression());
        }
```

El `writer` genera la escritura en caché de los distintos valores del campo seleccionado y sus respectivas coincidencias, datos generados por la instancia de `BankiaCacheItemCounterWriter()`.



STEP 4: STEPCREATEFILE

- Generación del fichero de texto que con los contadores de los distintos valores según el campo introducido en el *step* anterior.

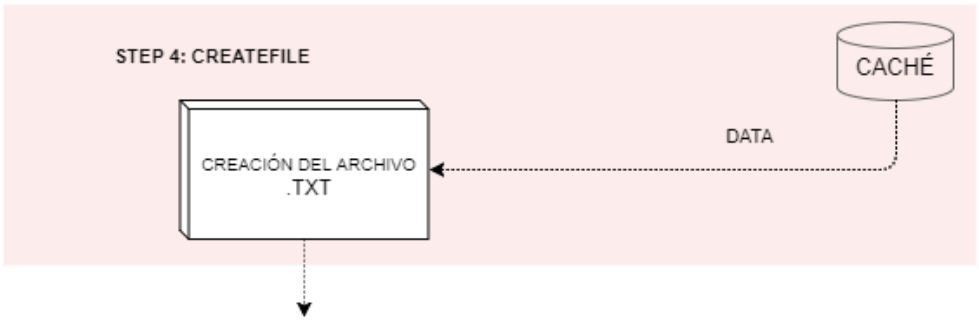
El `step` generado es:

```
    @Bean("stepCreateFile")
    public Step stepCreateFile(@Qualifier("bankiaItemCountFileTasklet")
 BankiaItemCountFileTasklet<Cliente> bankiaItemCountFileTasklet) {
        return this.stepBuilderFactory.get("stepCreateFile")
                                .tasklet(bankiaItemCountFileTasklet)
                                .build();
```

El step se basa en un tasklet incluído en la librería específica de Bankia que crea, desde caché, el fichero de texto plano con los distintos valores del campo seleccionado y el número de ocurrencias.

El nombre del fichero generado está enlazado con el nombre del fichero del siguiente *step* por practicidad, pudiendo alterarlo variando el campo `fileNamePattern` del `application.yml` :

```
bankia:
    batch:
        ftp:
          upload:
            fileNamePattern: Clientes.txt
```



STEP 5: STEPUPLOAD

- Apertura de canal de comunicación con el servidor.

- Envío del archivo a directorio elegido del servidor remoto.

El `step` generado es:

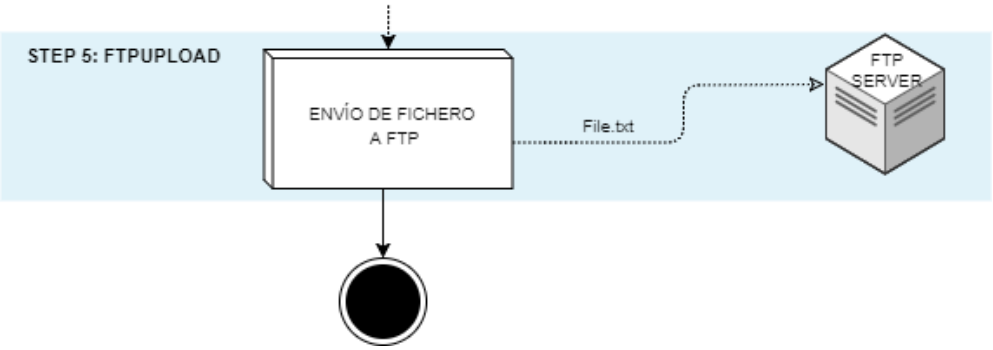
```
    @Bean("stepUpload")
    public Step stepUpload(@Qualifier("uploadFileTasklet") UpLoadFileTasklet upLoadFileTasklet) {
        return this.stepBuilderFactory.get("stepUpload")
                                .tasklet(upLoadFileTasklet)
                                .build();
    }
```

El step se basa en un tasklet que, haciendo uso de la configuración establecida previamente del servidor FTP, crea un canal de comunicación con un `MessageHandler` y envía el fichero al remoto.

La configuración se realiza mediante el `application. yml`. Éste contiene las siguientes variables:

```
upload:
        fileNamePattern: Clientes.txt
        remoteDirectory: /
        localDirectory: ${INPUT_LOCATION}/
        sftp: false
```

- `fileNamePattern:` nombre del fichero que se desea enviar al servidor. Variable compartida con el *step 4*. En este caso, el fichero `Clientes.txt ` generado sería encontrado automáticamente por la aplicación para enviarse.
- `remoteDirectory:` Repositorio destino para el fichero en el remoto. En este caso estará en el *root* del repositorio del usuario.
- `localDirectory:` Repositorio local donde se encuentra el archivo `Clientes.txt`.
- `sftp:` activa o desactiva el modo de envío mediante protocolo *sftp*, en este caso, está deshabilitado.



2.4. Caso de uso III (UseCase3)

Configuración general para la UseCase3

Para la prueba, la configuración general del `application.yml` ha sido la siguiente:

```
bankia:
  batch:
    thread-pool: 1
      core-pool-size: 20
#      max-pool-size:
#      queue-capacity:
```

```
    step: 2
      clientes: 1
        chunk-size: 10000 4
        throttle-limit: 32 5
```

1. `thread-pool:` Contiene la configuración referente a los hilos. En este caso, el `core-pool-size` se ha establecido en 32 hilos. Tanto para la cantidad máxima de hilos (`max-pool-size`) como la capacidad de la cola (`queue-capacity`) se ha adoptado su valor por defecto al estar comentados.
2. `step`: contiene la configuración general de los distintos *steps* que contiene la UseCase.
1. `<identificadorYML>`: identifica el *step* al que se referencian las variables. En este caso, el `step` del proceso es único y se ha denominado para su configuración como `clientes`.
4. `chunk-size:` determina el tamaño de los *chunks* en cada uno de los *steps*. En este caso, para el *step* se ha establecido en 10000.
5. `throttle-limit`: determina el número de hilos de ejecución simultáneos. En este caso está definido a un valor de 32.

Configuración y procesos específicos:

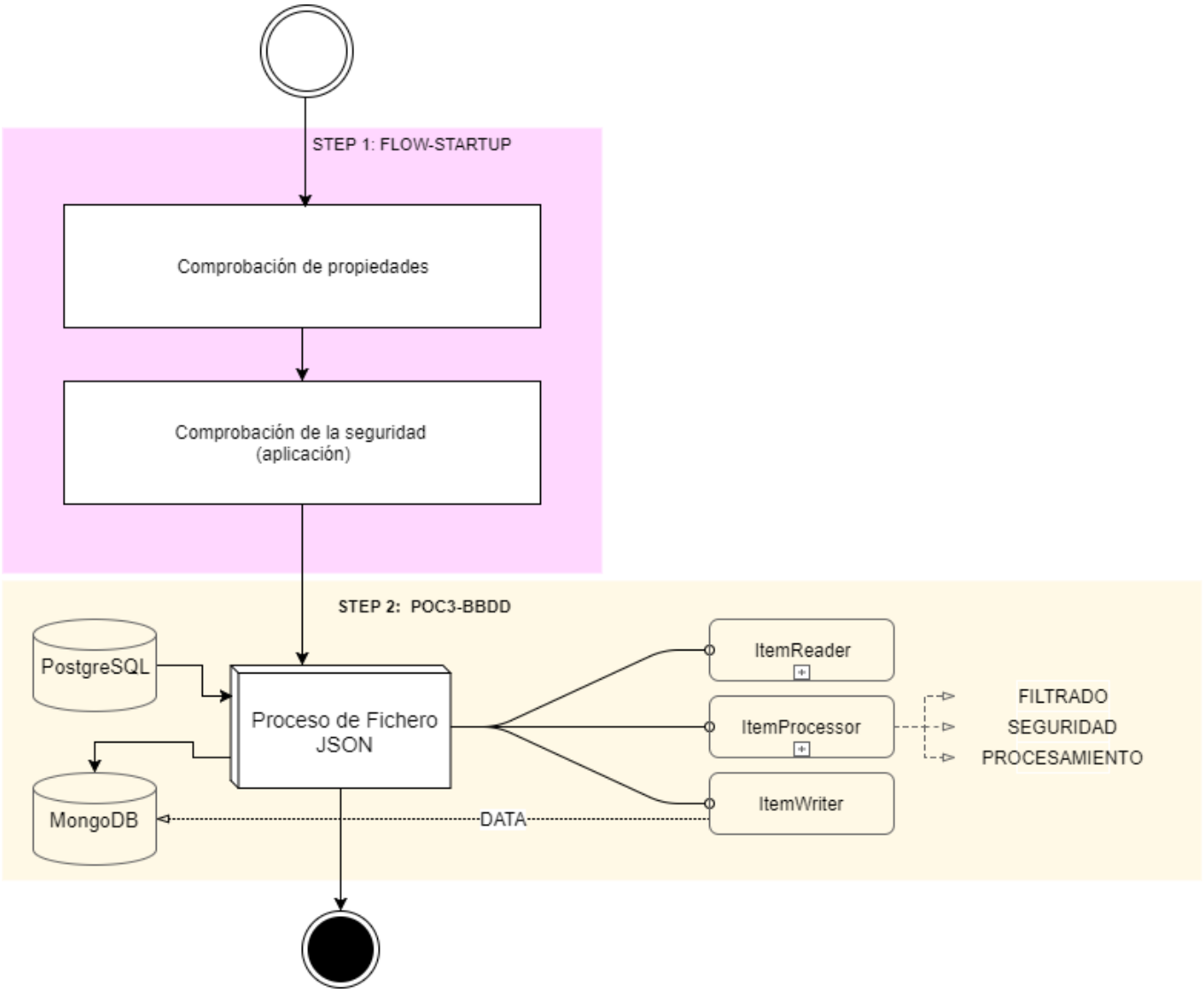
Para el caso de uso III (UseCase3) se explican a continuación los distintos pasos que ejecuta la aplicación, que se componen, en orden de ejecución, en el siguiente `job`:

```
@Bean("jobPoc3")
    public Job jobPoc3(
        @Qualifier("bankiaJobCompletionNotificationListener") JobExecutionListenerSupport listener,
        @Qualifier("flow-startUp") Flow flowStartUp,
        @Qualifier("stepPoc3BBDD") Step stepPoc3BBDD) {

        return this.jobBuilderFactory.get(applicationName)
                .preventRestart()
                .incrementer(new RunIdIncrementer())
                .listener(listener)
                .start(flowStartUp) 1
                .next(stepPoc3BBDD) 2
                .end()
                .build();
```

Cada uno de estos pasos cumple con alguno de los criterios funcionales solicitados. Éstos son:

1. `start(flow-startup)`: uso de una librería proporcionada por Bankia, comprobación de si el usuario tiene los permisos adecuados para ejecutar la funcionalidad en ese entorno.
2. `next(stepPoc3BBD)`: contiene la lectura desde Base de Datos *Postgres* de la información en el *reader*, el procesamiento de las mismas con filtros por el *processor* y la escritura en MongoDB mediante el *Writer*.

El diagrama del UseCase3 es el siguiente:



STEP 1: FLOW-STARTUP

Inicio de la aplicación:

- Comprobación de la necesidad de ejecución de la aplicación y de sus propiedades.
- Comprobación de la configuración introducida en la aplicación.

La caso de uso se inicia con la ejecución del `flow` encargado de comprobar la existencia de las condiciones necesarias para iniciar la cadena de procesos.

STEP 2: Poc3BBDD

- Lectura desde Base de Datos PostgresSQL, filtrado y subida a Base de Datos Mongo.

1. ItemReader: Lectura de los datos desde *BD PostgresSQL*.
2. ItemProcessor:

   a. Filtrado según campos preconfigurados.

   b. Procesamiento de los datos.

3. ItemWriter: escritura de los archivos procesados en *MongoDB*.

El `step` generado es:

```
@Bean("stepPoc3BBDD")
    public Step stepPoc3BBDD(
            @Qualifier("poc3InputProcessor") ItemProcessor<Cliente, Cliente> processor,
            @Qualifier("poc3MongoWriter") ItemWriter<Cliente> writer,
            @Qualifier("taskExecutor") TaskExecutor taskExecutor) {

        BatchStep batchStep = batchProperties.getBatchStep("clientes");

        return this.stepBuilderFactory.get("stepPoc3BBDD").<Cliente, Cliente>
 chunk(batchStep.getChunkSize())
                .reader(jdbcReader(contractDataSource()))
                .processor(processor)
                .writer(writer)
                .exceptionHandler(new BankiaExceptionHandler())
                .listener(new BankiaItemReaderListener<>())
                .listener(new BankiaChunkCountListener("clientes", 1000))
                .taskExecutor(taskExecutor)
                .throttleLimit(batchStep.getThrottleLimit())
                .build();
```

La configuración del proceso se establece en el `application.yml` de la carpeta `profiles/bankia-aq-batch-pocs-poc3`:

```
bankia:
    batch:
        input:

          files:
            clientes:
              file-type: json
              location-type: local
              local-location: ${INPUT_LOCATION}/Clientes.json //SE NECESITA???
              filter-expression: "ssa != ''"
              key-fields-expression: "ssa"

          database:
            select-clientes:
                query: select c.id, c.nombre, c.apellido1, c.apellido2, c.direccion, c.indBaja, c.ssa
 from CLIENTE c
```

Los *bean* que forman su estructura batch (*reader*, *processor*, *writer*) son:

```
@Bean(destroyMethod = "")
public ItemReader<Cliente> jdbcReader(@Qualifier("postgresdatasource") final DataSource dataSource) {

        return new JdbcCursorItemReaderBuilder<Cliente>() 1
                .dataSource(dataSource).name("jdbcReader")
                .sql(batchProperties.getInput().getDatabase().get("select-clientes").getQuery()) 2
                .rowMapper(new ClienteMapper()).saveState(false).build(); 1
    }
```

1. El *ItemReader* es configurado mediante un *Jdbc builder* propio de Spring.
2. Se le indexa la consulta *SQL* contenida en el `application.yml`. En este caso, la *query* introducida es: `select c.id, c.nombre, c.apellido1, c.apellido2, c.direccion, c.indBaja, c.ssa from CLIENTE c`, consulta que realiza la petición de cada uno de los datos que se desea contener en el POJO `Cliente`.
1. Se devuelve el `ItemReader` con el método `build()`, tras denegar la persistencia de datos del item usado por el *reader* con `saveState(false)`.

Para la configuración de este *bean*, es obligatorio la definición de una fuente de datos o *datasource*. Se define:

```
@Bean("postgresdatasource")
    @ConfigurationProperties(prefix = "spring.data.postgres") (1)
    public DataSource contractDataSource() { (2)
        return DataSourceBuilder.create().build();
    }
```

(1) Se le indica la ruta donde se han configurado los distintos parámetros de conexión, ubicados en el archivo `yml` de configuración.
(2) Haciendo uso de `DataSourceBuilder` se genera la fuente de datos.

Tras la configuración del `ItemReader`, se debe configurar el *processor*:

```
@Bean("poc3InputProcessor")
    public ItemProcessor<Cliente, Cliente> poc3InputProcessor(SecurityValidation securityValidation,
            ExpressionParser parser) {
        BatchInputFile batchInputFile = batchProperties.getInput().getFiles().get("clientes");

        return new BankiaCompositeItemProcessorBuilder<Cliente, Cliente>()
            .filterProcessor(new BankiaFilterItemProcessor<>(parser,
 batchInputFile.getFilterExpression()))
                .processor(new ClienteItemProcessor())
                .build();

    }
```

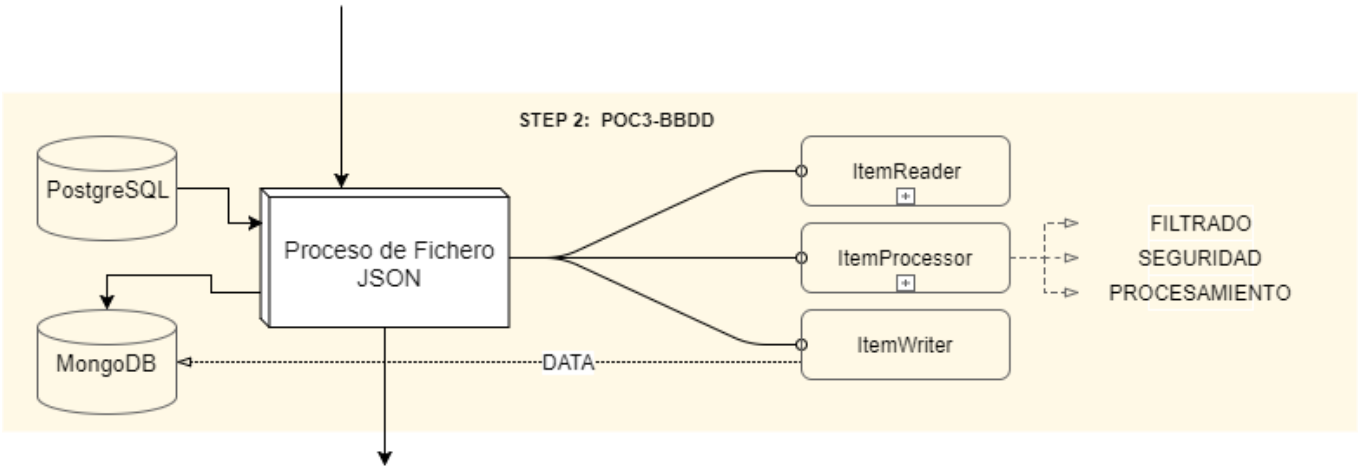
El *processor* realiza dos filtrados según expresiones contenidas en el `application.yml`:

Según condición, `"filter-expresion: indbaja=='N'"` .
Según unas condiciones de seguridad, `field: "ssa" function: "FFFADADF"`.

Por último, el *writer*:

```
@Bean("poc3MongoWriter")
    public ItemWriter<Cliente> mongoWriter(MongoOperations template) {
        return new MongoItemWriterBuilder<Cliente>() (1)
            .collection(mongoProperties.getCollection()) (2)
            .template(template) (1)
            .build();
```

(1) Se hace uso de una clase *builder* de la librería de Spring.
(2) Se introduce la colección que genera una librería específica de Bankia.
(1) La *template* introducida es generada también por la mencionada librería, y configura la conexión a la base de datos Mongo.



2.5. Caso de uso IV (UseCase4)

Configuración del archivo .yml

Para la prueba, la configuración general del `application.yml` ha sido la siguiente:

```
bankia:

  batch:
    ftp-enabled: false 1
    thread-pool: 2
      core-pool-size: 20
#      max-pool-size:
#      queue-capacity:
    step: 1
      clientes:
        chunk-size: 10000 4
        throttle-limit: 32 5

    input:
      files:
        clientes:
          file-type: json
          location-type: local
          local-location: ${INPUT_LOCATION}/Clientes.json
          filter-expression: "indbaja == 'N'"
          key-fields-expression: "ssa"

    output:
      database:
        postgres:
          query: "insert into clientes (idclientes, nombre, primerapellido,
 segundoapellido,direccion,indbaja,ssa) values(?,?,?,?,?,?,?)"
```

1. `ftp-enabled:` si el valor es *false*, se desactiva el protocolo FTP, ya que no se requiere para la UseCase4. CAMBIAR EN UN FUTURO
2. `thread-pool:` Contiene la configuración referente a los hilos. En este caso, el `core-pool-size` se ha establecido en 20 hilos.
1. `step`: contiene la configuración de los distintos *steps*. Para esta UseCase, solamente se realiza un *step* identificado como *clientes*.
4. `chunk-size:` el tamaño de los *chunks* para este único *step* se ha establecido en 10000.
5. `throttle-limit`: determina el número de hilos de ejecución simultáneos. En este caso, definido a 32.

Configuración y procesos específicos:

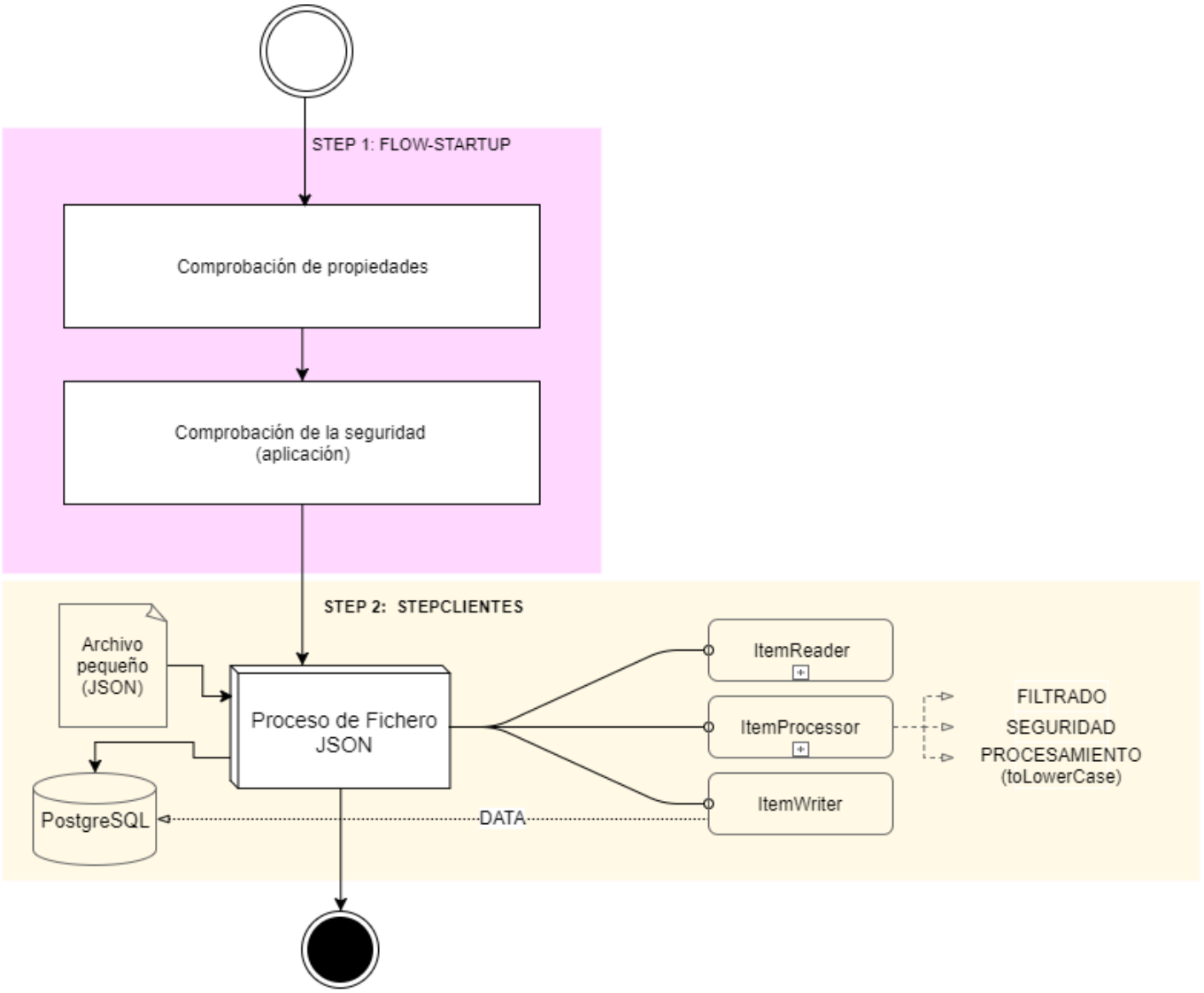
Para el caso de uso 4 (UseCase4) se explican a continuación los distintos pasos que ejecuta la aplicación, que se componen, en orden de ejecución, en el siguiente `job`:

```
   @Bean
   public Job job(@Qualifier("bankiaJobCompletionNotificationListener") JobExecutionListenerSupport
listener,
           @Qualifier("flow-startUp") Flow flowStartUp,
           @Qualifier("stepClientes") Step stepClientes) {
       // @formatter:off
       return jobBuilderFactory.get(applicationName)
               .preventRestart()
               .incrementer(new RunIdIncrementer())
               .listener(listener) 1
               .start(flowStartUp) 2
               .next(stepClientes) 1
               .end()
               .build();
       // @formatter:on
   }
```

Cada uno de estos pasos cumple con alguno de los criterios funcionales solicitados. Éstos son:

1. `listener(listener)`: *listener* encargado de la monitorización de los procesos.
2. `start(flow-startup)`: uso de una librería proporcionada por Bankia, comprobación de si el usuario tiene los permisos adecuados para ejecutar la funcionalidad en ese entorno.
1. `next(stepClientes)`: lectura de archivo Json, filtrado de la información por condición y grabado en caché de memoria.

El diagrama de la UseCase4 es el siguiente:



STEP 1: FLOW-STARTUP

Inicio de la aplicación:

- Comprobación de la necesidad de ejecución de la aplicación y de sus propiedades.
- Comprobación de la configuración introducida en la aplicación.

STEP 2: CLIENTE

- Lectura del fichero pequeño y escritura sobre base de datos relacional.

1. ItemReader: Lectura del archivo Json.
2. ItemProcessor:
   a. Filtrado según campos preconfigurados.
   b. Filtrado según parámetros de seguridad (desactivado).
   c. Procesamiento de los datos.
3. ItemWriter: escritura de los archivos procesados en base de datos.

El `step` generado es:

```
@Bean("stepClientes")
    public Step stepClientes (@Qualifier("clientesReader") ItemReader<Cliente> reader,
      @Qualifier("clientesProcessor") ItemProcessor<Cliente,Cliente> processor,
      @Qualifier("clientesWriter") ItemWriter<Cliente> writer,
         @Qualifier("taskExecutor") TaskExecutor taskExecutor) {

        BatchStep batchStep = batchProperties.getBatchStep("clientes");

        return this.stepBuilderFactory.get("stepClientes")
                .<Cliente,Cliente> chunk(batchStep.getChunkSize())
                .reader(reader)
                .processor(processor)
                .writer(writer)
                .listener(new BankiaItemReaderListener<>())
                .listener(new BankiaChunkCountListener("clientes", 1000))
                .taskExecutor(taskExecutor)
                .throttleLimit(batchStep.getThrottleLimit())
                .build();

    }
```

La configuración del proceso se establece en el `application.yml` de la carpeta `profiles/bankia-aq-batch-pocs-poc1`, y para un archivo Json:

```
bankia:
    batch:
        input:
            files:
                clientes:
                file-type: json
                location-type: local
                local-location: ${INPUT_LOCATION}/Clientes.json
                filter-expression: "indbaja == 'N'"
                key-fields-expression: "ssa"
```

Los *bean* que forman su estructura batch (*reader*, *processor*, *writer*) son:

```
@Bean("clientesReader")
    public ItemReader<Cliente> clientesReader() {
        return new
 BankiaFlatFileItemReaderBuilder<Cliente>().batchProperties(batchProperties).input("clientes")
                .targetType(Cliente.class).transformer(new ClienteFromMapper()).build();
    }
```

El *reader* es análogo al explicado anteriormente para el *reader* del *step clientes* de la UseCase1.

```
@Bean("clientesProcessor")
public ItemProcessor<Cliente, Cliente> clientesProcessor(SecurityValidation securityValidation,
        ExpressionParser parser) {
    BatchInputFile batchInputFile = batchProperties.getInput().getFiles().get("clientes");
    return new BankiaCompositeItemProcessorBuilder<Cliente, Cliente>()
            .filterProcessor(new BankiaFilterItemProcessor<>(parser,
 batchInputFile.getFilterExpression())) (1)
            .processor(new ClienteItemProcessor()).build(); (2)
}
```

(1) El *processor* realiza un filtrado según la condición `"filter-expresion: indbaja=='N'"`.

(2) Se procesan las variables según los pasos definidos en el archivo `ClienteItemProcessor.java`.

Nota

El *processor* en este caso realiza:

```
public Cliente process(Cliente item) throws Exception {

    item.setIdCliente(item.getIdCliente().toLowerCase());
    item.setPrimerApellido(item.getPrimerApellido().toLowerCase());
    item.setSegundoApellido(item.getSegundoApellido().toLowerCase());
    item.setDireccion(item.getDireccion().toLowerCase());
    item.setNombre(item.getNombre().toLowerCase());;
    item.setSsa(item.getSsa().toLowerCase());

    return item;
}
```

donde simplemente se reescriben los valores de todos los atributos a minúsculas (salvo el atributo *indbaja*).

Por último, el *writer*:

```
@Bean("clientesWriter")
public ItemWriter<Cliente> clientesWriter(@Qualifier("postgresdatasource") DataSource datasource) {


    return new JdbcBatchItemWriterBuilder<Cliente>().dataSource(datasource)
            .itemPreparedStatementSetter(itemPreparedStatementSetter())
            .sql(batchProperties.getOutput().getDatabase().get("postgres").getQuery())
            .build();
}
```

Genera la escritura en base de datos relacional a través de *PostgreSQL*, creando una estructura de mapa según la expresión `key-fields-expression:"ssa"` y según la *query* contenida en el `application.yml`.

Nota

En este caso, la *query* es : `"insert into clientes (idclientes, nombre, primerapellido, segundoapellido,direccion,indbaja,ssa) values(?,?,?,?,?,?,?)"`, escribiendo todos los atributos contenidos en el POJO en la base de datos.