## Memoria

Para la resolución del problema creo una solución en visual studio 19 con un proyecto de consola en .net core 2.2.

Para preparación del proyecto, le agrego los analizadores de código de StyleCop y sonaranalycer, para ayudar en el uso de buenas prácticas y estandarización del código.

Configuro el proyecto con el contenedor de dependencias de .net core (ServiceCollection).

Le cargo en ConfigurationBuilder para ficheros json, para sacar las configuraciones a un fichero appsettings.json.

Le configuro un serilog para el tratamiento de logs, y lo agrego en el contenedor de dependencias.

Creo 2 repositorios para la obtención y guardado de datos.

* IFileImporterRepository: con las funcionalidades de obtener el tamaño de un fichero y otra para obtener el un trozo del fichero.
  + Se implementa estas funcionalidades para un repositorio alojado en blobstorage de azure, a través de la clase FileImporterBlobStorageRepository.
* IElementRepository: Este repositorio está destinado a inicializar el destino de datos, y de guardar colecciones de objetos del tipo Element.  
  Se proponen 2 implementaciones.
  + La primera aproximación la realizo mediante la implementación a través de dapper. Pero observo que el rendimiento para tanta inserción penaliza mucho. ElementSqlServerDapperRepository
  + Me documento para saber cuál es la forma más óptima de guardar registros en sql server y observo que es a través de bulk insertions. Implemento una segunda forma de este repositorio, pero esta vez a través de SqlClient para poder usar esta funcionalidad. ElementSqlServerSqlClientRepository. Finalmente, esta es la implementación que configuro en el contenedor de dependencias para que se resuelva.

Para orquestar toda la operativa, se crea un servicio de aplicación llamado IDataImporter, con un único método Run().

Se implementa una única forma de realizarlo.

Algoritmo utilizado.

Primero, se consulta el tamaño del fichero a través del repositorio de IFileImporter.

En configuración tenemos almacenado el tamaño de los trozos con que descargaremos el fichero, y lo iremos tratando. Se calcula el número de trozos y se realiza lo siguiente.

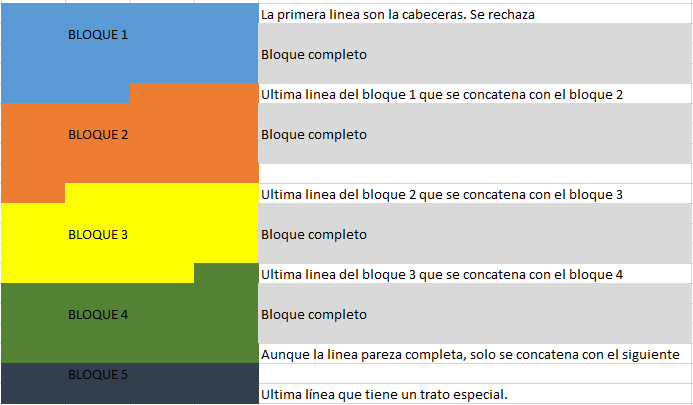
Tenemos 2 variables. Una colección de “filas completas”, es importante el matiz de completas, pues al trocear el documento podemos encontrarnos con que estamos tratando parte de una tupla. Una fila completa es la que se encuentra inequívocamente entre 2 saltos de línea.

Cogemos el primer trozo (si es el primero obviamos la primera línea pues se trata de las cabeceras), dividimos el fichero con Split de salto de línea, y agregamos todas las filas a esta colección menos la última que “Puede” estar dividida, esta información de la ultima fila, la guardamos en una cadena para concatenarla con el siguiente bloque.

Los sucesivos bloques, antes de tratarlos, concatenamos la información obtenida con el “trozo” de la última línea del bloque anterior, con lo que aseguramos que todas las líneas menos la última (esto no lo podemos asegurar) están completas.

En el último trozo, la última línea se trata de forma especial, se detecta si es una línea de datos, lo que se traduce en una nueva inserción, o de una línea vacía con lo que se obvia.

Este es el algoritmo finalmente implementado, pero se estudió la estrategia de dividir la tarea de descargar el fichero, en tantos hilos como procesadores tenga el equipo en el que corre la aplicación, e ir volcando la información en la colección antes descrita, y cada vez que se alterara esta colección, ir tratando en paralelo la información obtenida, siempre teniendo en cuenta que hay registros incompletos y que la única forma de evitar insertar información corrupta es esperar a tener bloques consecutivos descargados antes de poder tratar estos registros fronterizos.



Pero se detecto que el descargar el fichero de internet no tiene cuello de botella en procesamiento sino de red, y el descargar en paralelo no incrementaba la velocidad significativamente, pero si la complejidad del algoritmo, y lo mismo para el guardado. Con lo que se simplificó.

Para el guardado también se barajaron varias posibilidades de implementación, dejando como definitiva la de bulk insertions, que ofrece un rendimiento muy por encima de hacer bulk insertion de forma manual en bloques de 1000 registros, o insertando bloques a través de dapper.

Se proponer una solución donde se aplica el principio de inversión de control. Se utiliza una arquitectura n-capas, con capa de servicio y de repositorio. Si formara parte de un proyecto mayor, habría dividido cada capa en un proyecto, para publicarlo y enlazarlo al resto de la aplicación de formas mas limpia y arrastrando el mínimo de dependencias de nugets posibles.

## Propuestas de mejora

Se propondría un proyecto para los objetos de dominio, otro para los repositorios, y otro para los servicios de aplicación. Y finalmente un proyecto para albergar la aplicación de consola (o lo que sea, web, servicio local, escritorio…).

Se proponen más salvaguardas para un desarrollo mas completo, donde se verifique la conexión con la base de datos, que la tabla es justo como se quiera, conexión con internet, existencia del fichero blobstorage, verificación que se trata de un csv (por ejemplo, leer las primeras líneas y ver si el formato concuerda.).

Se proponen test de integración con los servicios de terceros (sql server, y azure blob storage.)

Se propone una mejor traza de lo sucedido, con mensajes que indiquen que parte va el proceso y el tiempo estimado de finalización.

Se proponer una política de reintentos por si salta la conexión a internet o a la base de datos.

Se propone una política de rollback por si en algún momento el proceso produce un error, se mantenga consistente la base de datos.

Se propone introducir todas las cadenas de texto en ficheros de recursos, unificados y fácilmente internacionalizables.

Se propone el despliegue de la aplicación como un servicio local subscrito a una cola Azure Service Bus (u otro manejador de colas) que trate los mensajes de la cola de uno en uno, y que en el payload del mensaje se especifique la URI del fichero, almacenado en un blobstorage, a tratar.

## Pruebas de rendimiento

Finalmente, una vez comprobada que la funcionalidad es completa, se prueban distintas configuraciones de tamaño de bloque a descargar y tamaño de la inserción bulk.

Variando estos 2 parámetros se obtienen resultados significativos en el rendimiento del desempeño de la tarea.

Pruebo un par de ellas y me quedo con la que considero una buena solución.

|  |  |
| --- | --- |
| Prueba | 1 |
| Tamaño bloque a descargar: | 1048576 |
| Tamaño del batch del bulk insertion: | 10000 |
| Inserciones en un minuto: | 3676388 |
| Memoria utilizada: | 80 MiB aprox. |
|  | |

|  |  |
| --- | --- |
| Prueba | 2 |
| Tamaño bloque a descargar: | 5242880 |
| Tamaño del batch del bulk insertion: | 15000 |
| Inserciones en un minuto: | 3989782 |
| Memoria utilizada: | 233 MiB aprox. |
|  | |

|  |  |
| --- | --- |
| Prueba | 3 |
| Tamaño bloque a descargar: | 5242880 |
| Tamaño del batch del bulk insertion: | 25000 |
| Inserciones en un minuto: | 3989782 |
| Memoria utilizada: | 266 MiB aprox. |
|  | |

|  |  |
| --- | --- |
| Prueba | 4 |
| Tamaño bloque a descargar: | 10485760 |
| Tamaño del batch del bulk insertion: | 50000 |
| Inserciones en un minuto: | 3989782 |
| Memoria utilizada: | 414 MiB aprox. |
|  | |

Después de estas pruebas, me decanto por la primera opción que, aunque es un poco más lenta, ofrece mucha estabilidad de uso de recursos, y un rendimiento aceptable.