ACCESO Y PROCESAMIENTO DE DATOS

Roberto Blázquez y Francisco Toribio

Contenido

Diseño y propuesta de solución.

Proyecto que importa y exporta diferentes archivos en distintos formatos. La aplicación genera un archivo jar que contiene todas las dependencias y favorece su uso en cualquier SDK.

Al ejecutar el jar con la opción parser, el programa muestra en la consola lo que va ejecutando.

Primero lee del .csv el modelo Residuo, después el modelo Contenedor para después parsearlos a .json, .csv, .xml y escribir la bitácora en .xml.

Con la opción resumen la aplicación exporta el informe a .html generando las gráficas. Si el archivo existiera y se volviera a generar, se añade al nombre uno entre paréntesis a modo de índice.

Para el .css se ha usado Bootstrap mejorando la vista de las consultas. Al final de los informes donde quedan reflejados los datos filtrados y sus gráficos, viene el tiempo de generación de la consulta.

Si no existiera la carpeta de destino de los informes, el programa la crea. La opción resumen se puede usar con cualquiera de los distritos de los archivos y con cualquier formato que admite la aplicación. Si no existe el distrito nos informa con un mensaje, creando la carpeta sin escribir la consulta.

Al ejecutarlo sin ningún argumento, nos muestra un mensaje con las indicaciones correctas de uso y los parámetros opcionales para poder indicar los archivos que se quieran leer, con un ejemplo de cómo realizar las consultas.

Si el archivo tuviera errores en los datos, la aplicación lanzaría un mensaje de error porque no podría leer el documento en cualquiera de sus formatos.

Clases y elementos usados. Justificación tecnológica.

Hemos utilizado:

Lenguaje de programación: Kotlin.

IDE: IntelliJ.

Control de versiones: Github.

Sistema de control de versiones: GitFlow.

Sistema de automatización de construcción de código: Gradle.

El programa funciona con dependencias para serializar el .xml y el .json. Usa el DSL para generar el .html de las consultas y construirlo de manera dinámica reutilizando más código.

Para las gráficas utilizamos Lets Plot y para el Data Science, Data Frames cargándolo todo en memoria. También usamos el Logger de Kotlin (wrapper de la fachada de Java).

Usamos corrutinas para leer y escribir de manera asíncrona.

Typealias para nombrar tipos de datos y estructuras ya existentes en nuestro código de una forma más visual.

Secuencias para optimizar las operaciones sobre colecciones, evitando la creación de objetos temporales entre los pasos de la cadena de computaciones.

DTO's para transportar datos.

También utilizamos el principio de segregación de interfaces para que ninguna clase dependa de métodos que no usa. Además, fomentamos la funcionalidad que nos aporta el uso de interfaces para evitar acoplar nuestro código.

Programación con clases genéricas para generalizar las funciones y poder reutilizarlas en más de una ocasión.

Las opciones heredan de clases selladas para aprovechar la potencia que nos da que cada subtipo (subclase) de una clase sellada sea otra clase y todos los subtipos se saben en tiempo de ejecución.

El programa lanza excepciones si las opciones no son correctas.

Estructura del programa: está dividido en lectores, escritores, importadores y exportadores.

Los controladores están abstraídos a interfaces con composición.

Transformación de formatos de la información.

Para la transformación de los datos de formato .csv a formato .json o a formato .xml usamos Kotlin serialization.

Realización de las consultas.

Para las consultas usamos DataFrame.

Gráficos.

Para la realización de los gráficos hemos usado los datos filtrados de los csv's con las consultas propuestas en la práctica.

Utilizamos las librerías que Kotlin nos proporciona, Lets-Plot, API de Kotlin para crear gráficos de código abierto para datos estadísticos y que tiene ggplot para dibujar los gráficos.

Aplicación de otras técnicas interesantes.

Inclusión de corrutinas y técnicas de programación asíncrona, usando runBloking y runCatching para logar el mensaje en consola y controlar los errores y excepciones.

Creación de flujo de trabajo de integración continua (CI) en acciones de GitHub para la comprobación y construcción del proyecto JVM con Gradle.