# Market Risk Data Cleanser

Dear reviewer,

Before you start review my code, I’d like to mention, that homework task statement have a lot of ambiguous points. It’s like a Suisse cheese with a lot of holes. In a real life, I wouldn’t accept that task from BAs or as a plain explanation in a task tracker. Let me describe those points and my assumptions for implementation. I’m not a nerd, all of this thing can be negotiated in a few mins personally =)

## Here is task description:

*In this excercise, we would like to emulate the scenario described above, by asking you to develop a simple library that is going to read time series data from the specified input files and then applying a simple algorithm to them.*

### Assumptions:

1. *simple library*  - library provides very simple public API by “facade”.
2. *read time series data from the specified input files* – I’m going to treat it as a path to folder with those files that woudn’t change over time. No file tracking, no change tracking. Only files presented at time of scan going to be processed.

## Data type description:

*The data is a series of data points composed of:*

* *Instrument (containing InstrumentId and Name);*
* *DataPointId;*
* *DateTime;*
* *Value*

### Assumptions:

1. *Instrument –* is a type.
2. *InstrumentId ­­*– integer positive (>0) value. All other values treated as error.
3. *Name –* string non-null and non-empty value. All other values treated as error.
4. *DataPointId –* integer positive (>0) value. All other values treated as error.
5. *DateTime –* Date-time value thatrepresent date not earlier than 2011, 1st Jan. Any other value treated as error. 2010, 31st Dec, 23:59:59.999 – is not valid entry.
6. *Value –* string value. There is no data how it should be treated, so I’ve decided to leave it as it is from human perspective – string. Other services can process it on they own. With additional information it can be implemented as type [Money](https://martinfowler.com/eaaCatalog/money.html), or any other [measurement type](http://www.step-10.com/SoftwareDesign/General/QuanitiesAndUnitsOfMeasure.html) with specific precision.

## Algorithm description

*and the algorithm to be applied ot the data sourced is the following:*

* *if a value is duplicated several times in a row in one week, set the value of its last occurrence to 0*
* *merge the data from all the input files and sort according to the DateTime.*
* *save the result as a JSON file.*

*Both input and output files should be formatted in JSON.*

### Assumptions:

1. *if a* ***value*** *is duplicated several times –* As soon as there no additional examples or notes, I’m going to treat at as literal matching with field name without taking into account any other information   
   (i.e. Instrument).
2. *if a value is duplicated* ***several*** *times –* “several” should be read as “more than once”
3. *duplicated several times* ***in a row in one week –*** Oneweekcanbe treated in several ways. It can be either “next/last 7 days” or “calendar week”. I’ve implemented both variants and one can change meaning of that statement.
4. *set the value of its last occurrence to 0 –* According to assumption 2, it means that any other occurrence of the same value within “one week” (see p.3) should be changed to “0”.
5. *merge the data from all the input files and sort according to the DateTime –* as soon as I don’t know hardware specifications I’m going to use as less memory as possible.

# Technical specification and ideas:

* Solution implemented using C# 7.0 (.Net 4.7).
* Target architecture intentionally set to x86, to consume less memory for general scenarios. And at the same time it provide restrictions to the maximum size of processing file – around 1.7 Gb, the rest memory consumed by runtime and co.   
  Processing bigger files requires dedicated splitters and in general moving to x64 wouldn’t solve issue, because it will mask higher level issues in architecture.
* Logging implemented with Structured Logs that can preserve meta information about logged event.
* JSON serialization intentionally implemented with Newtonsoft.JSON and JIL. The first one more flexible and used for processing external data, but slow. The second one using for fast processing and rely on very strict structure of data. It’s reasonable, because the app controls that (de)serialization process.
* Overall idea follows map-reduce strategy that allows consume less memory. There is always tradeoffs between resource consuming and performance.   
  At *map* step the app maps all data to “date-and-hour” folders, that stored as separate files for each source file.   
  At *reduce* step all files from “date and hour” sorted folders read and sorted within “date-and-hour”. Then one by one sorted set saved into final refined output json manually. That allows write down big file without memory pressure.