# crisp-mini-wrangler

Crisp Take Home Assignment - Mini CSV Wrangler

## TL;DR

To test, build the library and its example consumer and run it, simply run `./build-and-run.sh` from the project root.

It runs the configuration outlined in the specification document against the sample CSV from GitHub. It outputs the parsed and transformed values along with their type in the format `['<output field 1> (<output field 1 type>)','<output field 2> (<output field 2 type>)', . . .]`.

⚠ You'll need JDK11 installed an on the path for this to work.

## Usage

### CLI

The CLI is just an example consumer of the wrangler library to show its usage.

```
Usage: mini-wrangle-cli [-ahV] -c=<configuration file> -i=<CSV file>
Crisp Mini Wrangler - CLI Demo
  -a, --async     Process the file in an asynchronous fashion
  -c, --config=<configuration file>
                  Configuration file, see README for syntax
  -h, --help      Show this help message and exit.
  -i, --csv, --input=<CSV file>
                  CSV file to process
  -V, --version   Print version information and exit.
```

### Library

To be able the transformation library, you need a configuration file specifying the transformations and a data CSV. To get the transformed data, you register callbacks for produced `Records` and errors and run the transformation. The configuration can either be read either from an internal DSL, an external file or be created programatically.

*Library consumption*

```
val t = Transformer(
    configFile, ①
    { r: Map<String, Any> -> ②
        println(r) ③
    },
    { row: Array<String>, exception: Exception -> ④
        System.err.println("${exception.message} in ${row}") ⑤
    })
t.transform(inputFile, false) ⑥
```

① Pass in the configuration file

② Callback for produced records …

③ …, simply print them to `STDOUT`

④ Callback for errors …

⑤ …, print them to `STDERR`

⑥ Run `inputFile` through the transformation synchronously

## Configuration

Transformation configurations define …

- … the columns in the input file and their expected order,
- … the output records consisting of multiple fields,
- … the transformation from one or more input columns to an output field

Configurations are best edited within an IDE that supports Kotlin, as you get nice autocompletion and validation that way. However, this is not a strict requirement.

*Configuration Example as from the specification*

```
import net.wolfgangwerner.miniwrangler.model.config.* ①

wrangler {
    input { ②
        column("Order Number")
        column("Year")
        column("Month")
        column("Day")
        column("Product Number")
        column("Product Name")
        column("Count")
        column("Extra Col1")
        column("Extra Col2")
        column("Empty Column")
    }

    record { ③
        field("OrderID").integerFrom("Order Number") ④
        field("OrderDate").dateFrom("Year", "Month", "Day") ⑤
        field("ProductId").stringFrom("Product Number")
        field("ProductName").productNameFrom("Product Name")
        field("Quantity").decimalFrom("Count", "#,##0.0#")
        field("Unit").staticStringFrom("kg")
    }
}
```

① Import the configuration DSL

② Specify the format of the input CSV. Columns must appear in the order they appear in the CSV.

③ Output record specification, containing the field types and from which columns they are retrieved.

④ Example for a field of type `Integer` parsed from column `Order Number`.

⑤ Example for a field of type `Date` parsed from the columns `Year`, `Month` and `Day`.

# Build

**Requirements**

- Java/JDK >= 11

The project contains two modules, one for the library (`mini-wrangler-lib`) and one for the example consumer (`mini-wrangler-cli`).

To build the whole project, run `./gradlew build` in the project root.

To get the performance measurements, run `./gradlew clean measurePerformance` in the project root.

The README `pdf` rendition are created manually using `asciidoctor-pdf`.

# Architecture

The library and the CLI client are implemented in Kotlin. The transformation is able to run in a synchronous or asynchronous fashion using coroutines. The async way is faster if single row transformation durations exceed 10ms on average.

The transformer produces a stream of `Rows`, parses its `Columns` and produces a single `Record` for each row (or an error). A `Record` is made up of multiple `Fields` and their typed values.

Consumers subscribe to outputted `Records` and errors in a callback mimic and are free to handle them as they like.

For details, see Architecture Decision Records

## Terminology

- `Column` refers to a column in the input CSV
- `Row` is a a single line from the CSV as an array of Strings
- `Record` refers to to the typed and transformed representation of of a `Row`
- A `column ref` of a `Field` conceptually points to a `Column` from the input CSV
- `Field` represents a typed and named value in a `Record` [1]
- The `TransformationConfig` holds information about the input `Columns` to process and how to aggregate them into `Fields`
- `unmarshalling` refers to the process of getting a typed `Field` value from a `Row`

## Assumptions

1. Every input CSV file contains only records of one type, i.e. record-based text formats are not supported.
2. Every input CSV file contains exactly one header row designating the contained record’s fields.
3. The order of records in the output may differ from that in the input. Assuming an analytics data ingestion context, this should be fine.
4. Field types do not need to be specified within the external DSL, we can provide a pool of field types and transformations and configure the system using these.
5. We don’t need a rich domain model for the CSV data, as it is bound to differ per input format
6. We can’t use an internal (Kotlin) DSL, as the requirements explicitly state otherwise. I assume that the mappings are not created by core developers but rather analysts/consultants, potentially from customer’s staff.
7. While ideally the CSV parser used supports different encodings transparently, this application assumes the input to be in UTF-8

8. The target data format field types do not need to be specified externally, as the requirements state that

   > ```
   > [...] use case is taking a delimited data file from a customer and massaging
   > it to fit with a standardized schema [...]
   > ```

9. For proper casing of product names, we assume that each word is capitalized. Truecasing product/brand names would require a dictionary containing properly cased names. Truecasing of is an interesting NLP problem in itself, but I consider it outside the scope of this assignment. [2]
10. I assume all input dates in UTC. Supporting additional timezones would require extending the configuration by the timezone of dates represented as `String` and the corresponding type unmarshalling mechanism. I consider this out of scope for now.
11. The transformation to an output record does not require data from multiple input rows.

# Architecture Decision Records

## ADR 1: Stream Based Architecture

**Decision**

Since the requirements state that the input files may be potentially very large, the application should be able to deal with potentially unbounded streams of records.

## ADR 2: Transformations are can be run asynchronously

Transformations can be simple and fast for basic text wrangling, but can also grow complex and even have the need to access external systems. E.g. if an output field is required to contain the date of the maximum shelf live, this information could be required to be retrieved from a master data system of some sort. Another example would be to take shipping times or opening hours of a store into consideration for date calculations.

**Decision**

Since the order of records does not matter for our purposes (see [a-3]), we can run transformations in an async fashion. It must be able to turn off async behavior, since, if the transformation is cheap, serial processing may well be faster.

**Consequences**

- Costly transformations can be performed in parallel.
- The system is able to transform rows either in parallel or sequentially
- The order of output records is not guaranteed if processed in parallel (see [a-3])

Several (though not very sophisticated) test runs w/ 1000, 10_000 and 100_000 rows and different (mocked) transformation durations on a 8 core i7 2015 MBP indicated that:

- If transformations are instantaneous, sequential processing is significantly faster
- For transformations requiring 10ms and more, are roughly 8 times faster. This is consistent with the number of cores in the test machine.

The following table contains the rough average from tables generated by `LearningTests.compare sync and async processing`

*Table 1. Measurements for different transformation durations*

| Rows | Transformation ms | Duration sync | Duration async | async/sync |
|---|---|---|---|---|
| 100 | 0 | 7ms | 121ms | 17 |
| 100 | 10 | 1150ms | 150ms | 0.13 |
| 100 | 100 | 10s | 1.4s | 0.14 |
| 1000 | 0 | 15ms | 118ms | 0.12 |
| 1000 | 10 | 11s | 1.4s | 0.12 |
| 1000 | 100 | 102s | 13s | 7.8 |
| 10000 | 0 | 98ms | 542ms | 5.5 |
| 10000 | 10 | 1.8min | 15s | 0.11 |
| 10000 | 100 | 17min | 2.15min | 0.14 |
| 100000 | 0 | 796ms | 4262ms | 5.3 |
| 100000 | 10 | 19.7min | 2.7min | 0.13 |
| 100000 | 100 | 2.85h | 21.5min | 0.12 |

For more detailed analysis, I'd set up a JMH benchmark, but I'll skip that for now.

### Addendum after implementation

Actual measurements w/ generated test data show that for the example transformation from the instructions, the performance does not benefit from parallelization but use significantly more CPU cycles. See `TransformerPerformanceTests` for how the measurement was run.

*Table 2. Measurements for example transformation*

| Rows | Duration sync ms | Duration async ms | async/sync |
|---|---|---|---|
| 100 | 68 | 62 | 0.9117647058823529 |
| 1000 | 50 | 228 | 4.56 |
| 10000 | 185 | 509 | 2.7513513513513512 |
| 100000 | 283 | 2775 | 9.80565371024735 |
| 1000000 | 2640 | 30180 | 11.431818181818182 |

Simple `time`-ed invocations of the CLI yield the following example results for the format described in the specification:

*Table 3. Example measurements using* `time mini-wrangler-cli`

| Rows | Mode | User | System | CPU | Total |
|---|---|---|---|---|---|
| 100k | sync | 12.22s | 0.77s | 273% | 4.753 |
| 100k | async | 33.66s | 1.58s | 458% | 7.687 |
| 1M | sync | 19.67s | 2.49s | 131% | 16.826 |
| 1M | async | 207.00s | 8.98s | 483% | 44.685 |

## ADR 3: CSV Parser

While implementing a CSV parser by simply splitting rows at a delimiter character seems simple at the first glance, there are a lot of things that actually need to be taken into consideration (escaping delimiters in text columns, text delimiting, line breaks in texts, different line separators etc.).

For the JVM, a lot of CSV parser libraries are available, though some of which are quite dated. Univocity, a supplier of commercial data ingestion products, provides a performance comparison.

When selecting a parser, we need to make sure that it can perform in a streaming fashion as not to break [adr-1].

We don't need advanced mapping to objects (as we'll deal with multiple formats as opposed to having a rich domain model), as we will provide and run our own transformations on the parsed data, only robust and fast parsing of CSV records.

The requirements could probably be fulfilled using this library alone, however, this defeats the purpose of the exercise.

**Decision**

We're using SimpleFlatMapper.

The SimpleFlatMapper CSV module is the fastest OSS parser in the comparison mentioned above. It is actively being developed, with ~20 releases in 2019 so far and 300 stars on github.

SFM supports callback, iterator and stream based parsing.

Detailed performance stats by the SFM team here.

We're using the raw parser flavor as not to …

- … tie our implementation to much into a parser implementation
- … introduce runtime overhead for object mapping

## ADR 4: Decouple configuration data and configuration DSL

The current implementation uses a Kotlin DSL for defining transformations. We could want to support additional formats in the future and not prevent clients from creating configurations programatically.

**Decision**

We specify the DSL and the actual configuration used by the transformation in separate classes.

The configuration DSL can give return a configuration object, but the transformer does not know about the DSL.

This adds some code but decreases coupling.

## ADR 5: Don't put transformation code into DSL

The specification of the actual transformation logic could also be in the DSL.

**Decision**

As executing `.kts` scripts via `javax.script.ScriptEngine` is reportedly slow, we just evaluate scripts to retrieve configuration definitions. After having parsed the DSL, nothing else is executed as script. The available transformations are hardcoded in the library, see [a-4]

## ADR 6: Don't use infix functions in the record definition DSL

Infix functions for record definitions would allow for writing sth. like `field "foo" from "foo col" asType string` which would be quite readable.

However, since infix functions can only have a single parameter, we'd have to jump through some hoops to make that happen.

**Decision**

As of [adr-4] we can provide additional configuration definition formats should need be. For now, we got with a syntax that is idiomatic for Kotlin DSL, i.e. `field("OrderID").integerFrom("Order Number")` instead of `field "OrderID" from "Order Number" asInteger`.

## ADR 7: Minimize coupling w/ CSV parser

The CSV parser libraries reviewed in [adr-3] provide many options for mapping CSV rows to options. We could implement the complete wrangler on top of one of these.

**Decision**

As the exercise is about coming up with an own solution and we might perhaps want to switch parsing libraries later on, we strive to minimize coupling between the solution and the CSV parser used.

**Addendum after implementation**

There are only two places where the CSV Parser is used:

- in `net.wolfgangwerner.miniwrangler.transformer.Transformer.produceCsvRowsFromFile` to read rows and feed them to the transformer
- in `net.wolfgangwerner.miniwrangler.model.config.TransformationConfig.ensureCsvMatches` to get the header row for validation.

The second reference could be omitted, but I think it is beneficial to use the same parser for getting the rows and the headers for validation. Otherwise, there could be inconsistencies that are hard to track down.

# Next Steps

- I'd implement the possibility of consuming streams instead of using callbacks for consumers.
- The DSL validation could be improved.
- Refactor Tests to use parameterized tests, e.g. for field validations
- Support additional transformations for `Records`
- Factor out `StringField` concatenations, product name casing and `StaticStringValueField` (support all types of static values)
- Introduce own exception hierarchy instead of using (only) stock exceptions.
- Support better parsing for dates, perhaps merge `DateField` and `FormattedDateField`
- Generally make a better distinction between field types and transformations
- Improve the `Transformator` API to be easily callable from Java code. I'd use interfaces for the listener callbacks and refactor accordingly.

[1] So a String from a `Column` is to `Row` as `Field` is to `Record`

[2] I once built a (pretty specific) true casing tool that scraped existing data from a product catalog website, put the words into an Aspell dictionary and checked/corrected all uppercase product names against it.