Individual Data Import Tool Manual

Franz Brummer

2010-11-23

Contents

Individual Data Import Tool Manual....1
Introduction....1
Installation of the Individual Data Import Tool (IDIT)....2
Requirements....2
Installation....2
Creating an Import Project....2
Creating the File rules.txt....3
Creating the Files keys.data and columnlength.txt....5
Other Files....6
Importing the Data....6
Creating A Virtual Stock....6
Data Transformation....7
Import Into the Database....8
Update of the Database Metadata....8
Appendix....9
Description of Some Rules....9
Rule Add....9
Rule Age....9
Rule Count....9
Rule Splitup....9
Rule Exchange....10
Rule Enter....10
Rule Grouping....10
Rule GroupingSingle....11
Rule Year....11
Rule Multiple....11
Regel Multiply....11
Rule Accept....11
Rule Manifold....12
Rule If....12
Rule IfNot....12

Introduction

The Import Tool For Individual Data reads data from text files with fixed length fields and applies transformation rules to the fields.

Possible transformations are for example the conversion of a field, the creation of a new field by aggregation or the split up of a line in several new lines.

The transformed data is written to a CSV file (comma separated values) that can be imported into the database by the tool.

In the output data one line stands for exactly one individual case like one student or one exam etc.

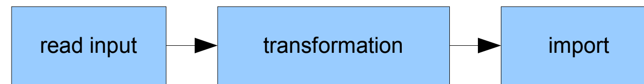

Figure 1: Data flow for the import

Installation of the Individual Data Import Tool (IDIT)

Requirements

The IDIT needs a Java Runtime Environment (JRE) version 1.5 or higher installed on the system to run. It runs on all systems which are supported by Java (Windows, Linux etc.). For the database connection JDBC drivers are used. Drivers for MySQL and Oracle DBMs are included with the tool.

Installation

There is no special installation procedure needed. The tool consists of only one file (ICEImport.jar) which can be copied to any directory in the file system. To make calling the tool from the command line easier it is advisable to copy it to a directory that isn't located very deep in the file system hierarchy.

Creating an Import Project

An import project contains all input and output files as well as a file with the transformation rules, database connection parameters and the database metadata of the stock.

A project has a strict folder structure:

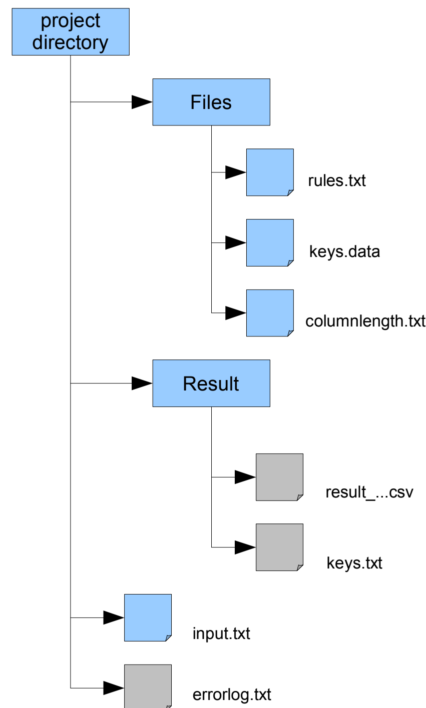

Figure 2: file structure of an import project

The project directory can have any name. It has to contain two subdirectories:

- Files/ contains
    - rules.txt with the transformation rules and the database connection parameters,
    - keys.data with the characteristics numbers for the columns in the output (result) file,
    - columnlength.txt with the field lengths for the database table
    - and all other text files which are needed for the conversion, like, for example, conversion tables.
- In Result/ the output (result) files are created at the end of the conversion step
    - result_dd-mm-yyyy_hh_mm_ss.csv the CSV file with the result of the transformation
    - and keys.txt with a list of all column attributes of the result of the transformation.
- in the top level project directory are the input file as well as errorlog.txt, that gets created after the transformation and contains a log of the transformation and any errors that may have occurred during the transformation.

Creating the File rules.txt

First of all the format of the input file has to be described. At the moment text files with fixed lengths per input field are supported.

The first line in `rules.txt` contains a comma separated list of all indexes where fields begin in the input lines.

Example:

The input file contains lines in which the first field has 4 characters length, the second field 2 characters, the third field 1 character and the fourth field 5 characters. That would translate to the following line in `rules.txt`:

```
1,5,7,8,13
```

This would result in every input line being split up into these four fields and beeing available for the transformation.

The next line describes the virtual stock in the ICE and the parameters for the database connection. This line has to begin with the keyword "Metadata". All the parameters have to be on one line.

Example:

```
Metadata 701 604 20062 1 jdbc:mysql://icends.his.de/ user password
ice com.mysql.jdbc.Driver edatentab_1011 1011 716 1 201 1 701 2
```

This line contains in fixed order:

- number of the virtual stock (701)
- characteristic number of the point in time (604 – semester)
- attribute of the point in time (20062 – winter term 2006)
- topic area (1 – study demand)
- JDBC URL for the database connection (jdbc:mysql://icends.his.de/)
- username for the database connection (user)
- password for the database connection (password)
- name of the database (ice)
- JDBC driver class (com.mysql.jdbc.Driver)
- prefix for the physical data table (edatentab_1011)

  If a complete table-id is used (e.g. edatentab_10110000), the import program tries to import exactly this table. If the table already exists, the program stops. If only a part of a name is used (e.g. edatentab_101100), the program searches for the next free table in the area of the missing numeric characters (in this example: 10110000 - 10110099).
- area of responsibility (1011)
- administrative keys, as an arbitrary count of characteristic/attribute pairs (716/1 – higher education institutions, ICE Lower Saxony; 201/1 – notation, count and 701/2 – quality of data, final)

Next come the transformation rules with every rule on its own line.

Every rule line begins with the name of the rule followed by any parameters the rule expects.

Examples:

Indexes used as parameters for rules usually relate to fields in the input line. The index count starts at 1. Taking the previous example for the splitting up of an input line the indexes would reference the following field positions:

index	character position	length

1	1 - 4	4
2	5 - 6	2
3	7	1
4	8 - 12	5

`Concatenate 99999 3,2`

> Adds a new field to the output line that consists of the concatenation of the contents of the fields 3 and 2 (in this order) in the input line. If the input fields should be empty 99999 is added to the output line instead.

`Pass 9 1`

> Adds the contents of field 1 of the input line unchanged to the output line. If the field is empty 9 gets added to the output line instead.

`Convert 99 conversions.txt 4`

> Opens the file `conversions.txt` in the directory `Files/` in the project directory. This file has to consist of two columns of numbers. In the first column are the possible values of the field in the input line and in the second column the converted values for the output field. In this case the values in field 4 in the input line are searched in the first column and converted to the corresponding values in the second column for the output line. If the input field is empty 99 is written to the output field.

The rules are applied to every line of the input file in the order in which they appear in the file `rules.txt`. The outputs of all rules are written to the output lines in the order of their application.

In the above example the output line would look like this:

```
[concatenation of fields 3 and 2];[field 1];[conversion of field
4]
```

Creating the Files keys.data and columnlength.txt

The file `keys.data` contains the definition of the characteristic in the database which is used for every field in the output line. This file is also used during the transformation step to validate the correctness of the attributes of all output fields against the database contents (every pair of characteristic (column) and attribute (field) is checked if it exists in the database, if any errors occur they are written to the file `errorlog.txt` – see further down).

Every line in this file represents one field position in the output line. This means that `keys.data` has to have exactly as many lines as the output line contains fields.

<u>Example:</u>

```
101 # study demand

604 # point in time semester

501 # gender
```

This means that when importing the output file into the database the first field in each line gets the characteristic 101, the second the characteristic 604 and the third the characteristic 501.

The corresponding field lengths for the database table fields are in the file `columnlength.txt`. This information is needed for the table definition in the database:

```
2  # field 1 study demand => integer(2) in database table
5  # field 2 point in time semester => integer(5)
1  # field 3 gender => integer(1)
```

So the files `keys.data` and `columnlength.txt` always have to have the same line count which equals the field count of the output lines.

Other Files

The directory `Files/` also contains all other files which are used by the rules (for example for conversion of fields – see above). More information about these files can be found in the documentation for individual rules.

Importing the Data

Creating A Virtual Stock

After the directory structure for the import project together with all the necessary files for the import are created the import can be started. But before starting the import, make sure that the virtual stock is listed in database-table `ice.virtuellebestaende`:

```
select * from virtuellebestaende where bestandnr=XXX
```

Otherwise you must make an entry to the database:

```
insert into virtuellebestaende values (<bestandnr>,
<bereich>,<zeitmerkmal>,<jahrstart>, <bezeichnung>,
<bestandsart>);
```

- The `<bestandnr>` (stock number) must be defined clearly (the table field `bestandnr` is unique).
- The `<bereich>` (area) derives from the topic defining characteristic that was used as key to encode the table data. For example if student data is encoded with characteristic 101, the virtual stock into which the data is imported must be related to area “1”.
- For `<zeitmerkmal>` (characteristic for time) use the id-number of the characteristic that is used to define the point in time of the table data.
- Field `<jahrstart>` (start of year) refers to the entry in data table `ice.jahrstart`. This entry characterizes the nature of dates (a year concerning students, for example, consists of a summer semester and the following winter semester.)
- Field `<bezeichnung>` (label) contains the part of a string. The data file that is to be imported to the virtual stock should begin with that string, otherwise the import gets canceled.
- When dealing with individual data import, for `<bestandsart>` (type of stock) always use “1”.

If the virtual stock is existent, copy the data file that you want to import into the project directory. The name for the data file is determined by the database entry for the virtual

stock and the point in time which is denoted on the line “Metadaten...” (metadata) in file regeln.txt (rules.txt).

Example:

Data has to be imported into the virtual stock with number 701 in the database of ICE Lower Saxony.

In database table ice.virtuellebestaende (virtual stocks) in the line with bestandnr=701 (stock number) the field bezeichnung (label) has value “HSS14”. In file regeln.txt the metadata line has a value of

```
Metadata 701 604 20062
```

This results in the data file name HSS14_20062.txt. So the name of the data file is constructed by concatenating the label of the virtual stock, an underscore (_) and the attribute of the characteristic point in time, followed by “.txt”.

Data Transformation

In a command line window one changes to the project directory. The import tool is then started with:

```
\path\to\java -jar \path\to\ICEimport.jar cli
```

This starts an interactive session with the import tool on the command line:

At first you have to give the path to the directory of the import project and push enter. If you are already in the project directory (like advised above) you only push enter.

The tool then checks if the project directory is valid. Several checks are performed for this validation:

- Does the virtuel stock number in rules.txt exist in the database? (that means there has to be an entry for the stock number in the table ice.virtuellebestaende).
- Do the directories Files/ and Result/ exist in the project directory and do they contain the necessary files depicted above? (rules.txt, keys.data, columnlength.txt)? (see figure 2)
- Does the data file for the import exist in the top level project directory and is its name valid? (see rule for the import file name above)
- If there already exists a data table in the database for the stock and point of time, the user gets the possibilities to delete the data table from the database before the import, to execute only the transformation step or to exit the import tool.

  Should the user decide to delete existing data table, a file named deleted_keys.txt will be created by the program in the project directory. The file contains a list of all keys, deleted from the meta data of the virtual stock concerned (table virtbestschlüssel), as they are not available in any other data table of the virtual stock anymore. Please act with caution at this point! The Summation – Characteristics will be deleted as well and have to be added manually afterward.

After the successful validation the transformation of the input file to a CSV file in the directory Result/ is started. During the transformation all rules get executed in their order in the file rules.txt on all input file lines in turn and the result is written to the output file. In case there occurs any serious error during the transformation (like a rule

trying to access a field in the input line that doesn't exist or wrong arguments to a rule) the program execution will stop and a description of the error will be output to the command line.

After the transformation is finished the tool outputs a message to the command line and waits for the user to check the file `errorlog.txt` in the project directory for any errors that occurred during the transformation.

This file contains information like the count of lines read and error messages like e. g. keys that don't exist in the database (every pair of characteristic and attribute that gets written to the output file is checked for existence in the database).

Additionally a file `keys.txt` gets created in the directory `Result/`, which contains a sorted list of all keys in the columns of the output file. This can be used to check the success of the transformation.

If the files `errorlog.txt` and `keys.txt` show that there have still occurred errors during the transformation the user cancels the import by responding "no" to the question if the import should be continued. In this case the information in the files `errorlog.txt` and `keys.txt` should be used to check and correct `rules.txt` and other files used by the transformation rules and restart the transformation.

Import Into the Database

If the tranformation runs without errors, the import of the output file into the database can be started.

For the import a new data table for the virual stock is created. For the name of this table the prefix in the line `Metadata` in `rules.txt` is used to find the next free table number.

In case an already existing data table has been deleted from the database at the start of the transformation the number of this table is reused for the new import.

Examples:

value in line Metadata	name for the new data table
`edatentab_1011`	the first free table name from edatentab_10110000 to 10119999 is used.
`edatentab_101100`	the first free table name from edatentab_10110000 to 10110099 is used.
`edatentab_10110022`	the table name for the import is edatentab_10110022. (There should not exist a table with this name!)

Once the name for the data table has been found all lines from the output (CSV) file `Result/result....csv` get read into this table.

Update of the Database Metadata

After the import of the data is complete all characteristics and their attributes from the file `keys.txt` get written to the database table `ice.virtbestschluessel`.

Additionally the administrative keys at the end of the line `Metadata` in `rules.txt` get written to the table `ice.virtbestschluessel` (they get numbered in order

verwaltung=1,2 and3).

Information about the virtual stock gets added to the tables `ice.virteinzelbestand` and `ice.icetabellen`.

If there exists a file `hints.csv` in the project directory the hints for the data table get read from it. (This file is created automatically from the database contents if the stock gets deleted from the database before the transformation but it can also be created with any text editor; it contains all fields from the database table `ice.hinweise`, separated by semicolons)

Appendix

Description of Some Rules

Rule Add

Adds an integer number to a value in the input line. To subtract a number, negative values (like -1) can be given.

Example for this rule in regeln.txt:

```
Add 9 1 2
```

Adds to the value at index 2 in the input line the integer 1 and returns the result, if the value at index 2 is empty or not a number, 9 is returned.

Rule Age

Rule to calculate the age of a person from the birth date and the current year in the input data.

Example for this rule in regeln.txt:

```
Alter 1 2
```

Gets the current year from index 1 (format YYYY) and the birth date from index 2 (format MMYYYY) in the input line. Returns the monthy and year of the birth date and the age in years.

Rule Count

Reads the count of copies that have to be made of a output line from a value in the input line. *This rule creates copies, so there must not be other rules who also create copies in the file rules.txt.*

Example for this rule in regeln.txt:

```
Count 1
```

Reads the number of copies for the output line from index 1 in the input line.

Rule Splitup

Makes copies of a line depending on the contents of multiple consecutive groups of fields in the output line. The output line is copied for every group and every copy is appended

with the values in the output line at the indexes in the relevant group. *This rule creates copies, so there must not be other rules who also create copies in the file rules.txt. The rule Splitup itself can occur multiple times in rules.txt to split up a line on multiple groups. All indexes refer to fields in the output line!*

For every group there has to be a pair of index (refering to the group: index 1 refers to the first element in the group) and value. This pair describes the condition (value at index in group) under which no copy of the line gets created for the group. If the index is 0 a copy of the line always gets created for the group regardless of its content and the value.

Example for this rule in regeln.txt:

```
Splitup 1 10 [2;3|4;5|6;7] 0 0 1 99 2 99
```

In this example a maximum of three copies of the ouput line is created. The values from index 1 to index 10 in the output line are cut out and for every group of indexes between 1 and 10 (groups 2/3, 4/5 and 6/7) a new copy is created by combining the values at the indexes 1, 8, 9 and 10 (the rest of indexes which are not part of any group) with the values of all groups and appending them to the output line.

The following rules are applied:

- for group 1 (indexes 2 and 3) a copy is always created, no matter what values there are in the output line at indexes 2 and 3 (0 0 means, always create a copy)
- for group 2 (indexes 4 and 5) a copy is created if the ouput line does not contain the value 99 at the first index of the group (index 4).
- for group 3 (indexes 6 and 7) a copy is created if the ouput line does not contain the value 99 at the second index of the group (index 7).

Rule Exchange

Exchanges the positions of two values in the input line.

Example for this rule in regeln.txt:

```
Exchange 1 2
```

Exchanges the values at indexes 1 and 2 in the input line and outputs the result to the output line.

Rule Enter

Enters one or more fixed values into the output line.

Examples for this rule in regeln.txt:

```
Enter 1
```

Appends the value 1 to the output line.

```
Enter 1 2 3
```

Appends the values 1, 2 and 3 to the output line.

Rule Grouping

Transforms discrete intervals of values in the input line in values for the output line. This rule reads the mappings from input to output values from a text file. The text file contains lines with either a pair of input and output values (e.g. “1 2”: the value 1 in the input line

gets translated into 2 in the output line) or an interval of input values and a output value (e.g. “1-10 2”: the values 1 to 10 inclusive in the input line get tranlated into 2 in the output line).

Example for this rule in regeln.txt:

```
Grouping 99 1 grouping.txt
```

Reads the value at index 1 in the input line and tries to translate it for the output line using the mappings described in the file grouping.txt. If there's no mapping found 99 gets written to the output line.

Rule GroupingSingle

Works like the rule Grouping. The difference is, that values for which there exists no mapping in grouping.txt get written to the output line unchanged.

Rule Year

Tranforms a two digit year value into a four digit year value. Years smaller than or equal to 20 get transformed into 20xx, years greater than 20 into 19xx.

Example for this rule in regeln.txt:

```
Year 1
```

Transforms the two digit year value at index 1 in the input line in a four digit year value for the output line.

Rule Multiple

Rule to create copies of an output line by combining the output line multiple times with some already transformed characteristics. The rule expects three parameters: the start index and end index of the interval in the output line where the attributes for the characteristics are and the length of one characteristic group in this interval. The group length has to be an integral divisor of the length of the interval from start index to end index. *This rule creates copies, so there must not be other rules who also create copies in the file rules.txt.*

Example for this rule in regeln.txt:

```
Multiple 0 9 3
```

This creates a rule Multiple for the interval 0 to 9 in the output line. The interval is cut out of the output line and divided into three characteristic groups of length 3 (indexes 0 to 2, 3 to 5 and 6 to 9). The result is that for every output line three new output lines are created. The first copy gets appended the values of the first characteristic group (0 – 2), the second copy the values of the second group (3 – 5) and the third copy the values of the third group (6 – 9).

Regel Multiply

Multiplies a value in the input line with an integer value. For division the reciproke value has to be utilised.

Example for this rule in regeln.txt:

```
Multiply 9 2 1
```

Multiplies the value at index 1 in the input line with the integer number 2 and writes the result to the output line. If the value at index 1 is empty or not a number, 9 is written to the output line.

Rule Accept

Takes a value from the input line and writes it to the output line without modification. Also a special value can be given that gets written to the output line if the input value is empty.

Example for this rule in regeln.txt:

```
Accept 99 1
```

Writes the value at index 1 in the input line to the output line. If the value is empty 99 is written to the output line.

Rule Manifold

Creates copies of the output line combined with certain attributes of a characteristic. The copy counts are read from fields in the input line. The arguments for this rule are one or more pairs of indexes and attributes of a characteristic. The copy count for each attribute is read from the field at its index in the input line. *This rule creates copies, so there must not be other rules who also create copies in the file rules.txt.*

Example for this rule in regeln.txt:

```
Manifold 1 3 2 4
```

This example gets the first copy count from index 1 in the input line and creates copy count copies of the output line with 3 appended as attribute. The second copy count is read from index 2 of the input line and copy count copies of the output line are created with the attribute 4 appended to each copy. The attributs 3 and 4 have to have a common characteristic.

Rule If

Checks if a field in the input line contains one of a number of values. If this is true then the output line is written, if not the output line is skipped.

Example for this rule in regeln.txt:

```
If 1 1 2 3
```

If the value at index 1 in the input line is 1,2 or 3 the output line will be written, if not it is skipped.

Rule IfNot

Checks if a field in the input line contains one of a number of values. If this is true then the output line is skipped, if not the output line is written.

Example for this rule in regeln.txt:

```
IfNot 1 1 2 3
```

If the value at index 1 in the input line is 1,2 or 3 the output line will be skipped, if not it is written.