@import - Importing data

Only in DbVisualizer Pro

This feature is only available in the DbVisualizer Pro edition.

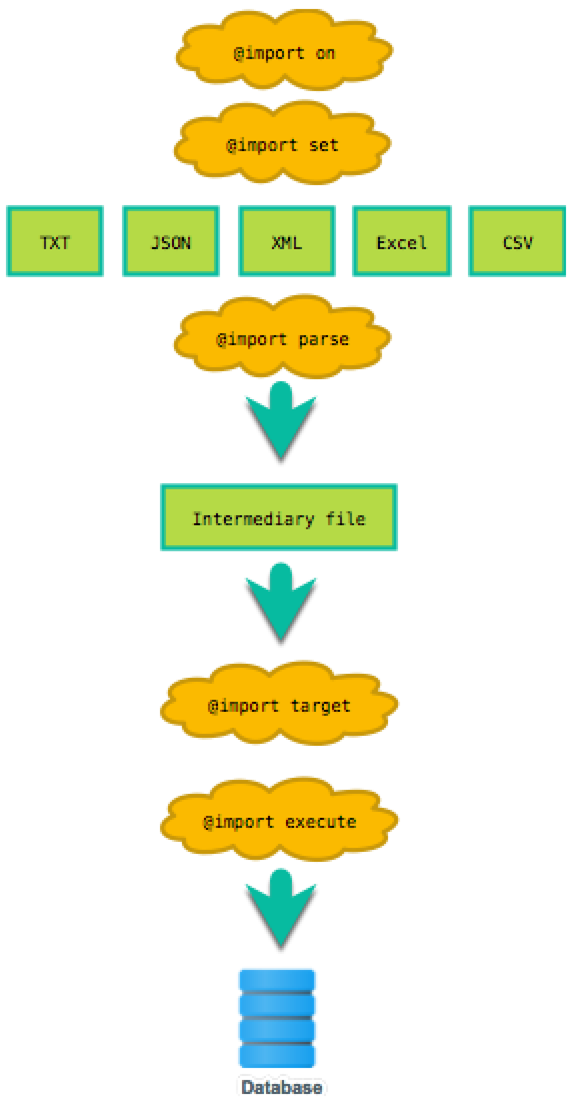
Instead of using the GUI to import data you can use client-side commands to import data, i.e **@import**. This enables you to use the DbVisualizer command-line interface to automate your imports and utilise other client-side commands such as @export, @mail, among others. Import data using the @import command supports the following formats:

- **Excel**
  xlsx or the legacy xls format
- **XML**
  The same XML formats that can be exported with DbVisualizer
- **JSON**
  The same JSON formats that can be exported with DbVisualizer
- **CSV**
  Importing CSV files supports a lot of configurations such as multi-symbol column separator, multi-line values, etc.
- **TXT**
  Importing fixed width text files

These are client-side commands for @import are:

`@import on`	The command starts an import session
`@import set`	Set parameters for the import
`@import parse`	Parse and convert the source data to an intermediary format, and analyze the data.
`@import target`	Identify the target table and what target columns should be used
`@import execute`	Runs the export

The import process is explained by the following figure.



The **@import parse** step **(1)** lays the foundation for the database import as it based on the source file format (csv, json, xlsx, etc), parses the data to an intermediary and internal file. This file is then analyzed in order to detect widths, data types and their sizes based on the data, row and cell references back to the source file, and a lot more.

The intermediary format stores the input data as **Records** associated with information from where the data **originates**.

Once the data has been parsed some basic tests are performed to see if the properties of the data is compatible with the **target table** properties. E.g. if the size of the input data will fit the targeted table columns. This done when when the **@import target** command is run **(2)**.

Final step of an import is to **execute** (**3-4**) the actual operations towards the database to import the data. I.e. **execute** the INSERT statements.
Any failures in the import will include specification of where in the source file the invalid data originates.

Following is a complete example where a simple CSV file is imported to a Target table Expenses.

Importing data - Example

The example shows how data about fruits included in a CSV file **fruits.csv** is imported into a database table **fruits**.

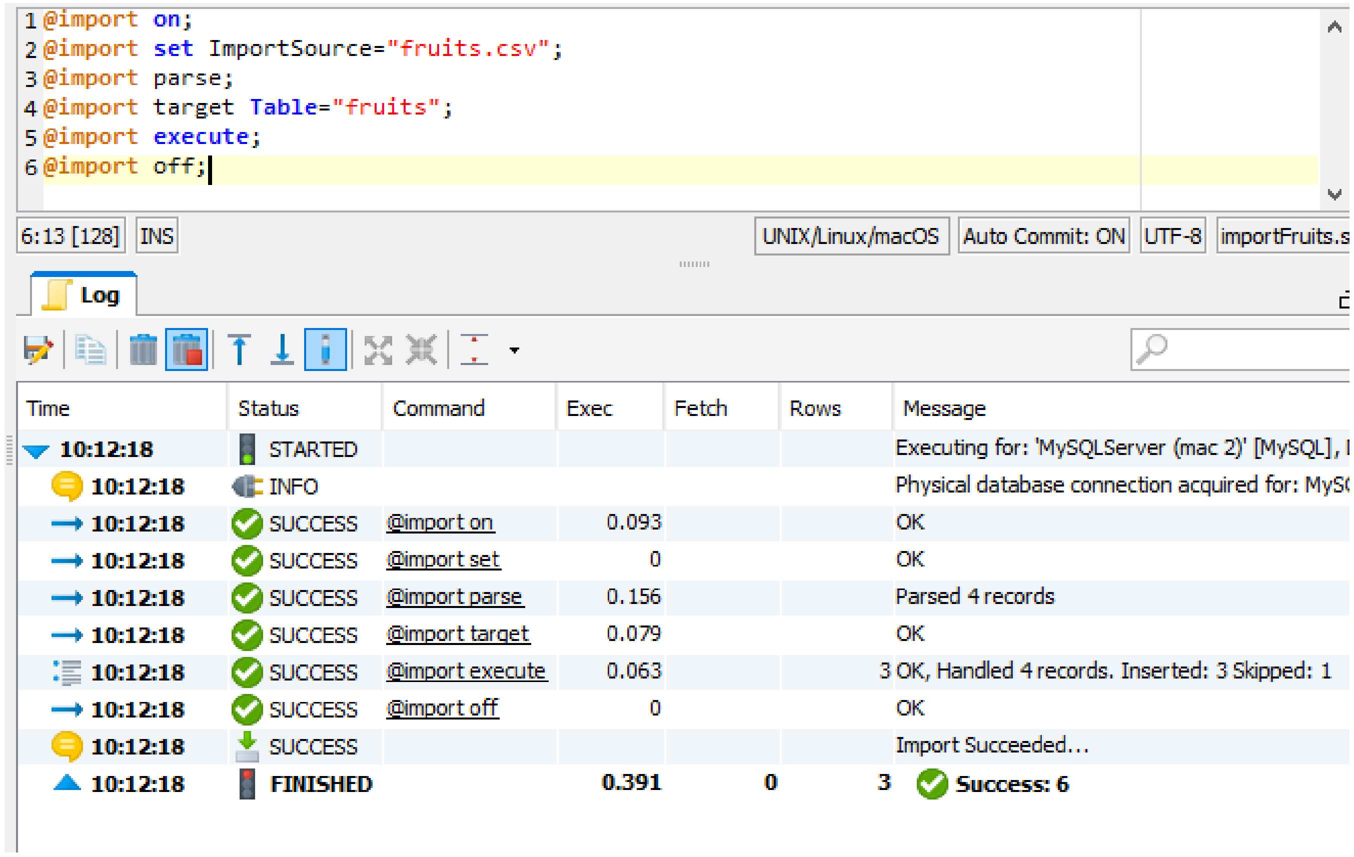
The example shows a minimal example where the columns of the CSV file is mapped directly to the table columns.

```
ID,FRUIT,PRICE
1,Banana,1.22
2,Apple,0.35
3,Orange,0.55
```

The SQL needed to import the file

```
@import on;
@import set ImportSource="fruits.csv";
@import parse;
@import target Table="fruits";
@import execute;
@import off;
```

Running the import script in DbVisualizer



```
@import on;
@import set ImportSource="fruits.csv";
@import parse;
@import target Table="fruits";
@import execute;
@import off;
```

```
6:13 [128]  INS                    UNIX/Linux/macOS  Auto Commit: ON  UTF-8  importFruits.s
```

```
Time      Status    Command          Exec   Fetch  Rows  Message
10:12:18  STARTED                                         Executing for: 'MySQLServer (mac 2)' [MySQL], I
10:12:18  INFO                                            Physical database connection acquired for: MySC
10:12:18  SUCCESS   @import on       0.093                OK
10:12:18  SUCCESS   @import set          0                OK
10:12:18  SUCCESS   @import parse    0.156                Parsed 4 records
10:12:18  SUCCESS   @import target   0.079                OK
10:12:18  SUCCESS   @import execute  0.063             3  OK, Handled 4 records. Inserted: 3 Skipped: 1
10:12:18  SUCCESS   @import off          0                OK
10:12:18  SUCCESS                                         Import Succeeded...
10:12:18  FINISHED                   0.391      0      3  Success: 6
```

As seen in the screen-shot above 3 records were inserted and and 1 was skipped (the header).

Read further on in the guide for how you can tailor your import using the different parameters of the import commands.

The @import commands

@import on

The **@import on** command initialises the import session. When a client-side command import session is started an **output folder** is created where DbVisualizer generates data representing intermediate results of the import. The output folder is created under a root folder (importresults) in the DbVisualizer preferences directory. The name of the folder is generated to be unique.

These folders are automatically cleaned by DbVisualizer on regular basis.

Note: Normally there is no need to specify any parameters to the **@import on** command. Default values should be sufficient for most uses.

Parameter	Default	Valid Values
ImportResult Root	*PREFSDIR*/importresults	The root folder for parsing results and other temporary files produced by the import session. **Note**: Any results produced in this directory will automatically be removed on regular basis.

ImportResultDir	Automatically generated as a sub folder to *PREFSDIR*/importresults	The path to put the DbVisualizer import results. If the path is an absolute path the results are stored in this folder. If it's not an absolute path it is assumed that it's path relative to the ImportResultRoot The folder is created when import is started. If the folder already exists, the import fails unless the **Clean** parameter is also used. If none of the ImportResultRoot/ImportResultDir is specified DbVisualizer will create a new uniquely named directory for every import session.
Clean	false	If true, the ImportResultDir is cleaned before import. If false import fails if directory exists. No need of specifying this parameter if **ImportResultDir** is not specified.
ContinueImportFile		Specified if you are continuing an import. See chapter about Continuing an Import.

Example 1

```
@import on ImportResultDir=/tmp/myimport Clean="true";
```

Import results are stored in the folder /tmp/myimport. Any existing folder /tmp/myimport will be deleted/cleaned.

Example 2

```
@import on ImportResultDir="newimport";
```

Import results are stored in *PREFSDIR*/importresults/newimport. If this directory exists the import will fail.

@import set

The **@import set** command takes a parameter name followed by an equal sign and a value, e.g. `parameter="value"`. You can use the following parameters, where **ImportSource** is the only required parameter. All names are case-insensitive. **Note** that you may use multiple @import set commands but the first one must include the parameter **ImportSource** setting the input data to import.

Parameter	Default	Valid Values
BooleanFalseFormats	false, no, 0, off	Comma separated string of values that should be interpreted as boolean false.
BooleanTrueFormats	true, yes, 1, on	Comma separated string of values that should be interpreted as boolean true
CsvColumnDelimiter		Can be used to override the auto detection by specifying a column delimiter. `CsvColumnDelimiter=";"` The most common column delimiters are auto detected.
CsvColumnDelimiterType	Auto Detect	Valid values are: **String** or **Auto Detect**. If only `CsvColumnDelimiter` is specified in the command `CsvColumnDelimiterType` is set to String

CsvTextQuotedBetween	None	Having data quoted is needed if the data itself contains the delimiter use to separate columns, special characters such as tabs, multiline separators, etc.
		
		The `CsvTextQuotedBetween` parameter is used to specify the identifier which is used to enclose this type of data. Please note that there may not be multiple text quote identifiers in the same import source.
		
		Any character is valid to use. Some additional aliases can be used to make the values more clear.
		
		• **`Single`** Equal to specifying the value as `"'"`
		• **`Double`** Equal to specifying the value as `""""`. Note that a " character need to be escaped with an additional ".
		• **`None`** Equal to specifying the value as `""` (The default)
		
		Example 1: For the following `CsvTextQuotedBetween="Single"` is used:
		
		`1       'Apple' 'Round fruit'`
		
		Example 2: For the following `CsvTextQuotedBetween="Double"` is used:
		
		`2       "Lemon" "Yellow fruit"`
		
		Example 3: For the following `CsvTextQuotedBetween="|"` is used:
		
		`3       |Lemon| |Yellow fruit|`
		
		If you have quoted data in your import source and do not specify `CsvTextQuotedBetween`, the data will be detected as text data.

DateFormat	yyyy-MM-dd	See valid formats in Changing the Data Display Format document.
DecimalSeparator	.	The decimal separator to use when parsing decimal numbers
Encoding	As set in Tool Properties	The file encoding to use when parsing the file to import
ErrorIncludeStackTrace	false	true/false If true the Java stack trace of any exception will be included in error messages.
ExcelCellPolicyError	SKIP	Defines what to do for a formula cell where the cached value indicates an error. Possible values: `EMPTY:` Set the cell to blank `SKIP_ROW:` Skip the complete row `ERROR:` Produce an error for the cell/row `TO_STRING:` Include the error as data.
ExcelSheetId	0	An id specifying the sheet id of a workbook when importing from xls/xlsx files. First sheet has id = 0 Either this parameter or ExcelSheetName parameter can be used. Not both.
ExcelSheetName		A name specifying the sheet name of a workbook when importing from xls/xlsx files.
FailOnConvertFailure	false	If true, the Import will fail if data conversion fails.
FailOnNoColumnsFoundFailure	false	If true, the Import will fail if we found no columns during continue import
FailOnParseFailure	false	If true, the Import will fail if we got a failure during parsing of the source data.
HeaderStartRow	`0`	The row index of the row where the Header starts. If set to a number **x** the StartRowOfData parameter is automatically set to **x +1.** The default value 0 indicates that the source data has no header information.
ImportSource		A path to the file to import. Must be included in the first **@import set** command. The path is an absolute path or a relative path to the script location. If a @cd command has been run before the **@import set** command a relative path is relative to the @cd directory
MaxRows	-1	The Maximum row to parse/import
ShowNullAs		The value that should be considered as NULL. E.g. **(null)**
SkipEmptyRows	true	true/false
SkipHeader	true	true/false If set to false the header is also imported.

SkipRowsStartingWith		String
StartRowOfData	1	The row index of the row where data starts. See also HeaderStartRow.
ThousandSeparator	,	The thousand separator to use
TxtColumns		Used when fixed columns text files are imported. Example **0, 4**. For detailed syntax of the TxtTrim parameter please see the example Importing fixed column width input data
TxtTrim	true	If true, the column data retrieved when importing fixed column text files is trimmed.
TimeFormat	HH:mm:ss	See valid formats in Changing the Data Display Format document.
TimeStampFormat	yyyy-MM-dd HH:mm:ss.SSSSS	See valid formats in Changing the Data Display Format document.

Example 1

```
@import set StartRowOfData="5" SkipRowsStartingWith="//";
```

We are starting to import data from row 5 of the source data file and skipping rows starting with "//".

Example 2

```
@import set HeaderStatRow="1";
```

The input data has header information at row 1. We are starting to import data from row 2. As `StartRowOfData` is not explicitly set is automatically set to 2.

@import parse

This command does all the parsing and analysing of input data.

Example

```
@import parse;
```

The source data file is parsed. As a result result are stored in the location pinpointed by ImportResultDir.

Example output from the `@input parse` command:

```
Parsed 5 records. Columns in source: 2 using ',' delimiter

INDEX NAME     TYPE       NULLABLE FROM ROW
----- -------- ---------- -------- --------
0     NAME     String(6)  No       2
1     BIRTHDAY String(16) No       4

Total bytes: 73 B
```

The information shows the number of parsed records along with the number of columns found. If the parsed file was a CSV file the used delimiter is printed.

For each column the following information is printed:

- **INDEX:** The index of the column
- **NAME**: If **`@import set`** parameter **`HeaderStartRow`** was specified and header information was extracted the extracted column name is printed.
- **TYPE**: The type of data found. The size declaration (E.g. **16)** represents the longest string found.
- **NULLABLE:** If the column is nullable or not.
- **FROM_ROW**: From which row in the source file the data type (e.g. String) was determined. This number serves as a hint to investigate source data when an unexpected type is analyzed.
  E.g. The column name **BIRTHDAY** in the source data indicates that this data should be a date. By investigating the source data at row 4 you may find the reason why the column was analyzed as String column.

@import target

This command is responsible of all preparation of the target table prior to import. This includes dropping, truncating, deleting from and creating the table.

When this is executed a check is done if the input data will actually fit the table. This is done by comparing of the analyze result with the specified target and column mapping**.** Depending on the parameters the check is performed at different occasions. Parameters for this command are:

Parameter	Default	Valid Values
Catalog		The target table Catalog
CleanData		Specifies if data should be cleared before import. Values **Drop** The table is dropped before import. If this is used either the CreateTableSQL or CreateTableSQLFile must be specified **Clear** The table is cleared. Either through the use of TRUNCATE or DELETE. Default method is TRUNCATE. Override the default **Clear** method by specifying the parameter **ClearTableMethod**. Before the table is cleared the check of input data towards the target table is performed.
ClearTableMethod		**Truncate** or **Delete**.
ColumnMapping		Specifies mapping of source columns to target columns. The default is to import the source columns to the target table column by index. First column in source is import in first column in table. To specify another order or to ignore certain columns use ColumnMapping parameter. Syntax: `ColumnMapping="<src col>=<tgt col>, <src col=tgt col>, ..."` Source column can be identified by index starting with 0 or by its name. Target column can be identified by index starting with 1 or by its name. Example: `ColumnMapping="0=2, 1=3"` or `ColumnMapping="id=no, color=col"` **Overriding type** The mapping also supports overriding of the **type** information of the input data column. I.e overriding the type information deducted by DbVisualizer when parsing/analysing the data. The syntax in this case, for a single column mapping is: `<src col>(<type spec>)=<tgt col>` Where `<type spec>` is one of: `String, Date, Time, Timestamp, Number, Decimal Number, Boolean, BLOB` and `CLOB` Example: `ColumnMapping="id(Number)=no, color=col"`
ColumnMappingFile		A reference to a file containing the column mapping.
CreateTableSQL		The SQL needed to create the table to import too. This parameter or the `CreateTableSQLFile` parameter is required if `CleanData="Drop"` is specified.
CreateTableSQLFile		A file reference to a file containing the SQL to create the table.
DropTableSQL		The SQL for dropping the table
FailOnDropFailure	false	If true, the Import will fail if the DROP table statement fails.
Schema		The target table schema to import to
SkipValidateColumns		A comma separated list of target column names for which validation shall not be done. Example: `SkipValidateColumns="Acol,Bcol"`
SkipValidateColumnNumbers		A comma separated list of target column numbers for which validation shall not be done. Example: `SkipValidateColumnNumbers="1,5"`
SkipValidateJdbcTypes		A comma separated list of JDBC type names for which validation shall not be done. Example: `SkipValidateJdbcTypes="INTEGER,TINYINT,VARCHAR"`
Table		The target table to import to
UseDelimitedIdentifiers	false	true or false If true object identifiers will be delimited.

Some examples

Example 1

```
@import target Table="MyTable" ColumnMapping="0=ID,2=NAME" CleanData="Drop"
               CreateTableSQL="CREATE TABLE MyTable (id SMALLINT, name VARCHAR(45))";
```

Target table is the "MyTable" table. The table is dropped and recreated before import. We are mapping the first column of the input data to the target column "ID" and the third column to the target column "NAME".

As the table is dropped we need to supply the DDL/SQL for creation of the table.

Example 2

```
@import target Table="MyTable" ColumnMapping="2=NAME" CleanData="Clear"
```

The table is cleared before the import. The clear method is determined by DbVisualizer. For Databases supporting this Truncate is used.

@import execute

Run the actual import.

Parameter	Default	Valid Values
BatchImport	true	**true** or **false** Using batch import will significantly improve the import speed. Note though that batch import may not be supported by all databases or JDBC drivers. In error situations it is also a good idea to switch off batch import.
BatchSize	100	For every 100 (or specified) number of rows being inserted, DbVisualizer will run a commit.
FailOnInsert Failure	false	If true, the Import will fail if an INSERT statement towards the database fails

Examples

```
@import execute BatchImport="false";
```

Run the import. Don't import using batch import

Examples

Selecting data to import and mapping columns

CSV File delimited by exclamation mark "!".

```
Volvo!XC90
BMW!F32 4 Series
Volvo!XC60
Mercedes!C197 SLS AMG
```

Note that the first column of the CSV file is the brand name (Volvo) of the car. The table we are importing to have the columns in opposite order model, brand. I.e. we need to Map the columns.

```
CREATE TABLE carmodel (model VARCHAR(50), brand VARCHAR(50));

@import on;
@import set ImportSource="cars.csv" CsvColumnDelimiter="!" SkipRowsStartingWith="BMW";
@import parse;
@import target Table="carmodel" ColumnMapping="0=brand,1=model";
@import execute;
```

Parameters used

CsvColumnDelimiter="!" specifying that the data is delimited by the character '!'.

SkipRowsStartingWith="BMW" We are not importing BMW cars

ColumnMapping="0=brand,1=model" Column **0** of the CSV file is mapped to the **brand** column of the table. Column **1** is mapped to the **model** column.

The table carmodel content after import:

```
model        brand
------------ --------
XC90         Volvo
XC60         Volvo
C197 SLS AMG Mercedes
```

Overriding analysed type information

When an input file is parsed DbVisualizer analyses the data to determine data types of the input data. The algorithm for this is quite coarse. DbVisualizer does offer a way to override the analysed data type.

CSV data

```
NAME,BIRTHDAY
August, "Sat, 21 Jul 1962"
Sven, "Fri, 21 Jan 1972"
Lotta, "NoData"
Bert, "Sat, 21 Jul 1962"
```

Note that the birthday of "Lotta" is "NoData" which is of course not a valid date. When DbVisualizer parses/analyses the data, it will come to the conclusion that the BIRTHDAY column is a String.

The result of @import parse will contain a table describing information about the data that was parsed.

```
Parsed 5 records. Columns in source: 2 using ',' delimiter

INDEX NAME     TYPE       NULLABLE FROM ROW
----- -------- ---------- -------- --------
0     NAME     String(6)  No       2
1     BIRTHDAY String(16) No       4

Total bytes: 73 B
```

As mentioned earlier you can see that column BIRTHDAY has been interpreted as a String. This was found examining row 4 (FROM ROW column is 4).

In connection with inserting this column in the database DbVisualizer would insert/set this as a string. This would result in total import failure and no rows would be inserted in the database.

This may be addressed by overriding the analysed type for BIRTHDAY (String) and set the type to date.

The SQL

```
CREATE TABLE birthdays (name VARCHAR(40), birthday DATE);

@import on;
@import set ImportSource="birthdays.csv" CsvTextQuotedBetween="Double" DateFormat="EEE, d MMM yyyy"
HeaderStartRow="1";
@import parse;
@import target Table="birthdays" ColumnMapping="0=name,1(date)=birthday";
@import execute;
@import off;
```

Parameters

- DateFormat="EEE, d MMM yyyy"
  Defining the format to be able to interpret the dates in the CSV file.
- ColumnMapping="0=name,1(date)=birthday"
  Note the **1(date)=birthday** where we are mapping the column with index 1 to the target column birthday. The **(date) part specifies that column 1 should be interpreted as a date.**

The result of the import using the script above is that 3 rows are imported (August, Sven and Bert). The row representing Lotta is reported as a failure as Indicated below.

```
1 Record affected, Record: 0 originating at row: 4
DataRecordException: Convert error, Column: BIRTHDAY at index: 1
DataTypeConversionException: Value is 'NoData'. Not a valid date format. Valid format: 'EEE, d MMM yyyy'
```

Importing fixed column width input data (TxtColumns parameter)

Text File

```
001   APPLE
002     LEMON
003   ORANGE
```

The SQL Script

```
@import on;
@import set ImportSource="fruitlist.txt" TxtColumns="0,6";
@import parse;
@import target Table="fruitslist" Catalog="test";
@import execute;
```

The parameter **TxtColumns** parameter specifies the column character positions. In this case first column starts at character position 0 and the second column starts at character position 6.

The resulting imported table is

```
id name
-- ------
1  APPLE
2  LEMON
3  ORANGE
```

Note how "LEMON" is imported without proceeding blanks. This is because column values are trimmed (TxtTrim parameter default is true).

The TxtColumns parameter

The TxtColumns parameter supports a number of syntaxes as explained in the examples below.

An example when parsing a row "AAA BBB CCC"

TxtColums parameter	Yields extracted columns
0-2, 4-5	"AAA" "BB"
1-3, 8-9	"AA" "C"

Omitting the end index (as in the SQL script above)

TxtColums parameter	Yields extracted columns
0, 4, 8 (same as 0-3,4-7, 8-end of line)	"AAA" "BBB" "CCC"
1, 8-9 (same as 1-7,8-9)	"AA BBB" "C"

Using the "+" sign

TxtColums parameter	Yields extracted columns
0+3, 4+3, 8 (Same as 0-3,4-7, 8-end of line)	"AAA" "BBB" "CCC"

0+7, 8+1 (Same as 0-7, 8-9)	"AAA BBB" "CC"

Importing Excel data

Excel file

```
A   B
--  --
1        A
2   #DIV/0!
3        C
```

Note how row 2 column B has the value **#DIV/0!.** This value represents a case where a cell is a calculated using formula where the calculation is producing an error.

SQL Script to import

```
CREATE TABLE exceldata (id INT, value VARCHAR(50));

@import on;
@import set ImportSource="excelData.xlsx" ExcelSheetName="mydata" ExcelCellPolicyError="SKIP_ROW";
@import parse;
@import target Table="exceldata";
@import execute;
@import off;
```

Parameters used

ExcelSheetName="mydata"
Specifying that the sheet named mydata is is the sheet to import.

ExcelCellPolicyError="SKIP_ROW";
Specifies that if we find a formula that produced an error we should skip the complete row.

Resulting Table

```
id value
-- -----
1  A
3  C
```

Note that row 2 is not imported as we instructed by ExcelCellPolicyError="SKIP_ROW";

Continuing an export that has failed

If any of the input data cannot be imported DbVisualizer will keep track of this. This is done by storing the failed data in a specific errorRecords.drec file in the directory where the import process stores its intermediate results (See ImportResultRoot parameter).

Following is an example were the export fails. It also shows how to import the failed data again. Specifically, the first import fails as Clementine is a name that is too long to fit in the target table column.

```
ID,FRUIT,PRICE
1,Banana,1.22
2,Clementine,0.35
3,Orange,0.55
```

First Import SQL Script

```
CREATE TABLE fruits (id SMALLINT, name VARCHAR(6), price DECIMAL(10,2));

@import on  Clean="true" ImportResultDir="/tmp/importContinueFirstImport";
@import set ImportSource="fruits.csv" HeaderStartRow="1";
@import parse;
@import target Table="fruits" SkipValidateJdbcTypes="VARCHAR";
@import execute;
@import off;
```

Note: the table definition for the fruit table defines the column name to be **VARCHAR(6).** The input data **"Clementine" will not fit there.**

Parameters used

ImportResultDir="/tmp/importContinueFirstImport" We get our results in this directory. Makes it easy to refer in the second import script.

SkipValidateJdbcTypes="VARCHAR" Tells DbVisualizer not to validate VARCHAR columns. This is done for the purpose of this example. If not specified, the import would stop before any data has been imported.

When running this import the Database used in the example (MySQL) will fail when the import tries to insert the data row 3 as Clementine will not fit the column. The Failure printed in the DbVisualizer Log for this looks something like:

```
1 Record affected, Record: id = 0 originating at row: 3
DataRecordException: Error when importing data.
MysqlDataTruncation: Data truncation: Data too long for column 'name' at row 1
```

Note that the source data is pinpointed as **originating at row: 3**.

The Table fruits content after import.

```
id name   price
-- ------ -----
1  Banana 1.22
3  Orange 0.55
```

Second import SQL Script

```
ALTER TABLE fruits MODIFY COLUMN name VARCHAR(20);

@import on ImportResultDir="/tmp/importContinueSecond" UseImportFile="/tmp/importContinueFirstImport
/errorRecords.drec" Clean="true";
@import execute;
@import off;
```

The ALTER statement is dealing with the root cause why the import failed. The name column was too small.

Note that when continuing an import, the commands **@import set** and @import target is not specified. The settings and target from the old import is used.

Parameters used

ImportResultDir="/tmp/importContinueSecond": Specifying a separate directory for the results

UseImportFile="/tmp/importContinueFirstImport/Results/errorRecords.drec" Pinpointing the file containing the data that contains the data that could not be imported.

The Table fruits content after second import.

```
id name       price
-- ---------- -----
1  Banana     1.22
3  Orange     0.55
2  Clementine 0.35
```

Testing an import - Dry Run

The client-side import offers a way to run the import script to perform all client side data validation without changing anything in the database. This is done using the **@set dryrun** command.

Note that when running the import, without dry run, the import may fail nevertheless due to checks on the database side. E.g. primary- or unique key constraint checks.

```
@import set ImportSource="fruits.csv";
@import parse;
@set dryrun;
@import target Table="fruits" CleanData="Clear";
@import execute;
@set dryrun off;
@import off;
```

When running the script, no actual clearing of the table will be done as the parameter **CleanData** indicates. Nor does the **@import execute** lead to any rows being inserted in the database.

Since there is a **@set dryrun** command prior to the commands no changes to the database table will be performed.