Loadsheets

December 5, 2020

Overview

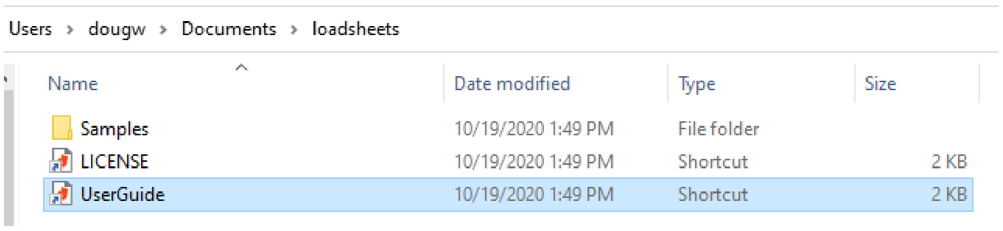
The `loadsheets` program appends rows to an existing Excel "template" file from one or more Excel "source" files as directed by a configuration file. This approach provides a simple, automated way to create "loadsheets" of new product information as required by several industry retailers and distributors.

Installation

The setup program is a Windows Installer package stored in a zip file and downloaded from our website. The installer is digitally signed and should not be used if "Unknown" or has a signer other than "Winsby Group LLC". It is a 64bit console app that installs under the local user (so admin rights are not required). The name of the file is:

```
loadsheets_setup_(version).msi
```

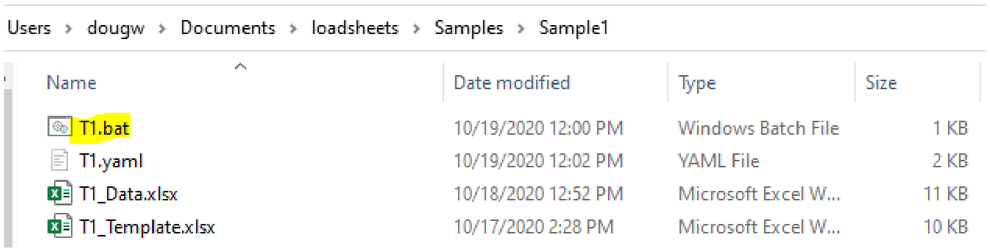
Once installed, file explorer will open the new "loadsheets" directory in your Documents. For example:



Users › dougw › Documents › loadsheets

Name	Date modified	Type	Size
Samples	10/19/2020 1:49 PM	File folder	
LICENSE	10/19/2020 1:49 PM	Shortcut	2 KB
UserGuide	10/19/2020 1:49 PM	Shortcut	2 KB

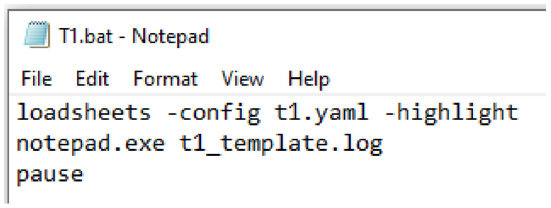
Here you will find a "Samples" folder which contains a few example projects for you to explore. Unlike other Windows applications you may be familiar with, `loadsheets` does not have a user interface. This means that everything is run from the command line (either cmd or powershell). But, in order to make it easier to run, you can create "batch" files which will run the project and present the results in notepad.



Users › dougw › Documents › loadsheets › Samples › Sample1

Name	Date modified	Type	Size
T1.bat	10/19/2020 12:00 PM	Windows Batch File	1 KB
T1.yaml	10/19/2020 12:02 PM	YAML File	2 KB
T1_Data.xlsx	10/18/2020 12:52 PM	Microsoft Excel W...	11 KB
T1_Template.xlsx	10/17/2020 2:28 PM	Microsoft Excel W...	10 KB

For example, here are the contents of T1.bat:



```
loadsheets -config t1.yaml -highlight
notepad.exe t1_template.log
pause
```

The first line is the command that runs the `loadsheets` program. Note that we're telling it which configuration file to use, and instructing it to "highlight" any cells in the output Excel file that have problems.

The second line uses "notepad" to display the results of the run to the screen. This "log file" always uses the name of the template file as its base name (with a .log extension).

The "pause" simply keeps the command window from closing until you press enter.

Features

1. Excel (xlsx) and CSV files are used for source data.
2. Excel (xlsx) files are used for template layout (and final output).
3. Multiple source files can be used (joined together by part number).
4. Coded ("stacked") data sources are supported (e.g. multiple rows per part number: Part, Code, Description).
5. Console application allows automated processing.
6. Data validation at the field level.
7. Data validation between fields (including conditional requirements)
8. One or more fields may be used for filtering rows with full Boolean expressions:

```
$LINE_CODE == "XYZ" && $POP_CODE in ["A","B"]
```

9. Option to include **runtime parameters** for filtering or other conditional validation:

```
Loadsheets -config oreilly.yaml -params "p1=ABC"

$LINE_CODE == p1
```

10. Built-in functions which can be used in conditional expressions:

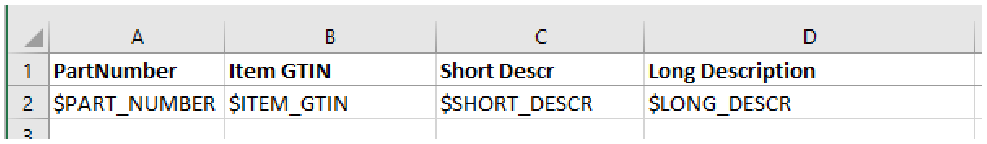
```
$EFF_DATE < Today()
$EFF_DATE < Date(p1)
```

11. Log file includes complete error descriptions with cell references (for both input and output identification).
12. Option to highlight any cells with errors in the output file.
13. All output values default to string values, with the ability to force Excel native data types if required.

Configurations

All processing jobs are controlled by a configuration file specifically designed for each customer's requirements. These files are written in a markup language (called YAML) using any text editor (e.g. Notepad++, VS Code, etc.). It is usually easiest to start with an existing config file and make changes as necessary to fit your current requirements.

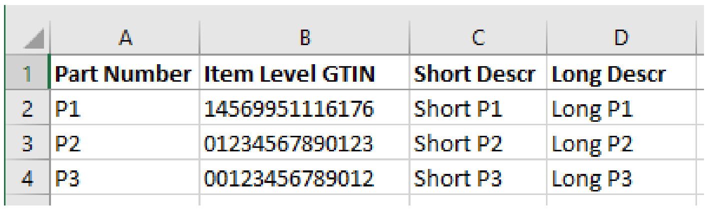
Besides the configuration file (aka "config" file), we also need a "template" file and at least one "source" file. The template file is simply an Excel file with placeholders for the source data. For example, here is a very simple template file with just four template placeholders (or "variables"):



	A	B	C	D
1	PartNumber	Item GTIN	Short Descr	Long Description
2	$PART_NUMBER	$ITEM_GTIN	$SHORT_DESCR	$LONG_DESCR
3				

Note that template variables start with a dollar sign and are all included on a single row. In this example, we've started on row 2, but you can start on any row needed (e.g. some templates have several rows of explanation at the top and so you'd want to start your data after those lines). There is a "DataRow" setting in the config file to indicate which row contains the template variables.

Let's now look at a source file that we might use to fill this template.



	A	B	C	D
1	Part Number	Item Level GTIN	Short Descr	Long Descr
2	P1	14569951116176	Short P1	Long P1
3	P2	01234567890123	Short P2	Long P2
4	P3	00123456789012	Short P3	Long P3

Note that the headings in the source file do not need to match the headings in the template file. Instead, we use the config file to help us match them up.

There are two main sections to a config file, “Template” and “Sources”. The **Template** section defines the output requirements (for filling in an existing Excel file). The **Sources** section defines the data inputs that will be used to fill the template.

Minimal Example

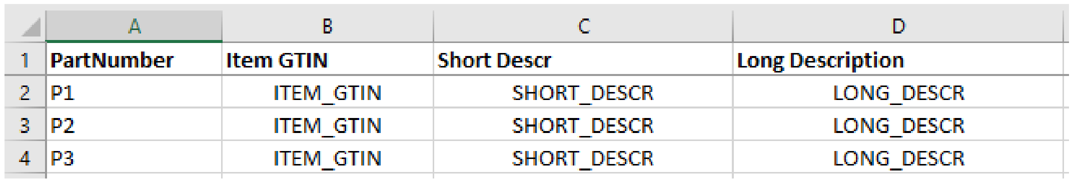
Sometimes it is easiest to look at a simple example. Here is a config file that fills a single column (the part number) in the above template file.

```
Description: Minimal Example

Template:
  FileName: "Template.xlsx"
  WorkSheet: "Data"
  DataRow: 2
  Variables:
    - {Name: $PART_NUMBER, Type: "string", Required: true}

Sources:
  - Name: Primary
    Description: Main Parts file
    FileName: "Data.xlsx"
    WorkSheet: "Data"
    HeaderRow: 1
    Columns:
      - { Heading: "Part Number", Format: "part", TemplateVariable: $PART_NUMBER }
```

If we tried to run with this config against the above data and template files, though, we’d get an error because there are three variables defined in the template file that are not defined in the Template section of the config file. If we removed the "$" from those extra variables, however, we’d get something like the following:



	A	B	C	D
1	PartNumber	Item GTIN	Short Descr	Long Description
2	P1	ITEM_GTIN	SHORT_DESCR	LONG_DESCR
3	P2	ITEM_GTIN	SHORT_DESCR	LONG_DESCR
4	P3	ITEM_GTIN	SHORT_DESCR	LONG_DESCR

Note that the part number was properly populated, but why was the other data copied down on each row? This example shows how you can provide “default” values that never change (and are not provided by a data source).

Samples

As mentioned above, sample projects are provided to get you started. It is usually easier to copy an existing config file and make changes than to start from scratch.

Required Fields

There are two methods to require that a template variable be filled from a data source. The easiest way is simply to add “`Required: true`” to a variable definition (all fields default to false).

The second way is to add required logic using the “`RequiredIf`” condition. This allows you to set the required flag based on other variables on the record. For example, you might want to require an inner pack length whenever an inner pack quantity is provided. An example of this is shown below (and in Sample2):

```
 - {Name: $IN_PK_LEN, Type: "number", RequiredIf: "$IN_PK_QTY != nil && $IN_PK_QTY > 1"}
```

Filtering

There are times when you might want to exclude some part numbers from the results. This can be done by a "Filter" expression added to one or more variables in the Template section. For example:

```
  - {Name: $PARTNUMBER, Type: "date", Required: true, Filter: "$POP_CODE in ['A','B']"}
```

If more than one Filter is defined, all must be satisfied in order for the row to be included. If there is a chance that the source data is empty, you must add a check for "nil" to keep the process from attempting to compare a nil value. For example:

```
Filter: "$POP_CODE != nil && $POP_CODE in ['A','B']"
```

This tells the processor to first check that the pop code is not empty, before attempting the "in" operation. The "&&" means AND which will exit ("sort-circuit") the evaluation if evaluating to false.

Built-In Functions

There are server functions provided for special use when creating conditional expressions. Two of the more useful functions concern dates. The Today() function allows you to compare a date against today. For example:

```
Filter: "$EFF_DATE != nil && $EFF_DATE < Today()"
```

The second function converts a string date to the internal Excel date format.

```
Filter: "$EFF_DATE != nil && $EFF_DATE < Date('2020-01-01')"
```

Because date comparisons are so common, we included three additional functions to make that easier. For example, the following will only include rows where $EFF_DATE is on or before '2020-01-01':

```
Filter: "DateOnOrBefore($EFF_DATE, '2020-01-01')"
```

Note that you do not need the test for "nil" because that is handled by the function.

Runtime Parameters

It's sometimes helpful to provide a value for use in one of the conditional expressions at runtime rather than needing to edit the config file each time. For example, instead of hard-coding the date in the above filter, you might want to write something like:

```
Filter: "$EFF_DATE != nil && $EFF_DATE < Date(p1)"
```

Here, we've defined a parameter called "p1" which will hold the value we include on the run command. That might look something like:

```
loadsheets -config t2.yaml -highlight -params "p1=2020-01-01"
```

If you need to supply more than one parameter, simple separate them by a comma. For example:

```
loadsheets -config t2.yaml -highlight -params "p1=2020-01-01,p2='XYZ'"
```

See the section on "Writing Expressions" for more information.

SET /p Command (for user prompts):

Here is how to prompt the user for a value and pass it into the loadsheets program (for use in a filter expression).

```
set /p EXPDATE="Cutoff Date (yyyy-mm-dd): "
loadsheets -config t1.yaml -highlight -params "cutoff=%EXPDATE%"
```

The "EXPDATE" above could be any name (that is not already used as a variable). Note the use of it with %% around it in the `-params` phrase.

The actual variable, in this case, that is passed in is "cutoff" (you could make it p1, p2 or whatever you like). You would use the variable "cutoff" in the config file like this:

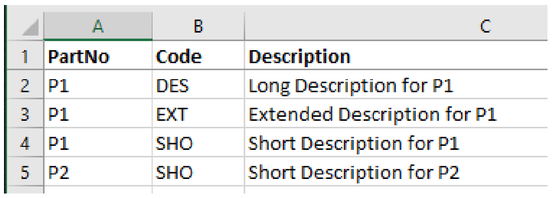
```
Filter: "DateBefore($EFF_DATE, cutoff)"
```

Resulting log file:

```
loadsheets (v1.0.5)
...
Run Parameters: [cutoff=2021-01-01]
...
Warnings: 1, Elapsed time: 496ms
```

Coded Files (Stacked Columns)

It's common to have source files with coded values which need to represent different columns in the output template. For example,



	A	B	C
1	**PartNo**	**Code**	**Description**
2	P1	DES	Long Description for P1
3	P1	EXT	Extended Description for P1
4	P1	SHO	Short Description for P1
5	P2	SHO	Short Description for P2

The biggest difference is that we need a repeating part number column and some kind of code to indicate the type of data represented.

In the above example, "SHO" represents the "Short Description" and must be "pivoted" to the correct column in the output. Presumably, there would be one template variable (column) for each code provided.

The program supports this type of data source with two special template variable values, `lookup:code` and `lookup:value`. For example:

```
    Columns:
      - { Heading: "PartNo",      Format: "part" }
      - { Heading: "Code",        Format: "text", TemplateVariable: "lookup:code" }
      - { Heading: "Description", Format: "text", TemplateVariable: "lookup:value"}
```

We then need a Code table to associate each lookup:code value to an actual template variable:

```
    Codes:
      - { Code: "DES", TemplateVariable: $LONG_DESCR }
      - { Code: "EXT", TemplateVariable: $EXT_DESCR }
      - { Code: "SHO", TemplateVariable: $SHORT_DESCR }
```

If you include more than one "lookup:code" column, they will be **combined to form the code** by concatenating the data in order separated by a **hyphen**. So, for example, DES-EN to create an English Long Description. If a lookup code column is empty, the hyphen separator will not be included (so you will never have a code with `--`).

```
    Columns:
      - { Heading: "PartNumber",   Format: "part" }
      - { Heading: "TypeCode",     Format: "text", TemplateVariable: "lookup:code" }
      - { Heading: "RecordNumber", Format: "text", TemplateVariable: "lookup:code" }
      - { Heading: "LanguageCode", Format: "text", TemplateVariable: "lookup:code" }
      - { Heading: "Content",      Format: "text", TemplateVariable: "lookup:value"}
    Codes:
      - { Code: "SHO-EN",   TemplateVariable: $SHORT_DESCR_EN }
      - { Code: "SHO-ES",   TemplateVariable: $SHORT_DESCR_ES }
      - { Code: "FAB-1-EN", TemplateVariable: $FAB_1_EN }
      - { Code: "FAB-2-EN", TemplateVariable: $FAB_2_EN }
      - { Code: "FAB-3-EN", TemplateVariable: $FAB_3_EN }
```

See Sample1 for a working example of a coded data source.

Source Formats

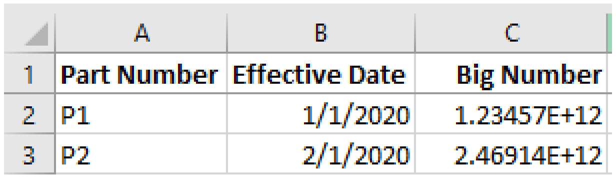
Each source column must have a "Format" which defines how to interpret the information found there.

Format	Type	Special Handling
text	alpha-numeric plus any special characters	spaces trimmed from front and back
number	no decimal point	validate as 64bit integer
decimal	with decimal point	extraneous decimal places removed
date	yyyy-mm-dd string (or Excel native date)	validate between 1/1/1900 and 1/1/2200
price	0.0000	string will always be 4 decimal places
gtin14	number	front zero filled to 14 characters, check-digit validated
part	alpha-numeric plus any special characters	used to identify part number column

Initial Data Conversion

Each data element (i.e. source column) undergoes two separate data conversion steps, once when reading and a second time when writing.

All data is read from Excel as a string regardless of how they were entered. The following source data, for example, shows three types of data (text, dates and numbers):



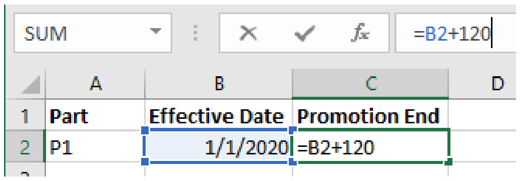
	A	B	C
1	Part Number	Effective Date	Big Number
2	P1	1/1/2020	1.23457E+12
3	P2	2/1/2020	2.46914E+12

The "Effective Date" column was entered (and interpreted) as a native Excel "Date" formatted number. So, when the program reads B2, for example, it sees "43831" (which represents the number of days since 12/31/1899).

The "Big Number" column was entered as a "General" formatted number. As seen here, General format displays large numbers in scientific notation. But C2, for example, is read as "1234567890123".

It is important to note that both of these numbers are now stored internally by the program as strings (with validations/conversions as described in the above table).

Calculated Values



	A	B	C	D
1	Part	Effective Date	Promotion End	
2	P1	1/1/2020	=B2+120	

The program also supports calculated values as shown above (here showing a date calculation, but any formula will work).

Conditional Expression Variables

Any template variable found in a condition (i.e. `RequiredIf` and `Filter`) uses the "typed" (native) value, not the "string" value. If a native value is missing (or when a conversion error occurs) the variable contains "nil".

Template Types

Each template variable requires a "Type" which defines how the information is displayed in the output Excel file.

Type	How Stored	Style
string	string value	left aligned
number	string value	format integer number as text, right aligned
number!	native Excel value	general integer number format, right aligned
float	string value	format floating point number as text, right aligned
float!	native Excel value	general floating point number format, right aligned
float2	string value	format number as text, right aligned
float2!	native Excel value	number format "0.00"
float4	string value	format number as text, right aligned
float4!	native Excel value	number format "0.0000"
date	string value	format as text, left aligned
date!	native Excel value	formatted as "yyyy-mm-dd", left aligned

Final Data Conversion

When the variable data is written to the template file, the second data conversion (and presentation) is performed. Note that the "!" styles should not generally be used because native values might not be presented properly. For example, a price variable would not have trailing zeros (4.000 is stored as 4) and so depends on the receiver to process that data properly. There are cases, however, where the receiver may request this format.

Command (Batch) File

Rather than typing `loadsheet` commands from a terminal window each time you want to process a job, you can use a `.bat` (command) file to automate the process. For example

```
@echo off
cls
pushd %~dp0
set /p EXPDATE="Cutoff Date (yyyy-mm-dd): "
loadsheets -config t1.yaml -highlight -params "cutoff=%EXPDATE%"
start /b notepad++.exe "%CD%\t1_template.log"
start /b excel.exe "%CD%\t1_template_%date:~10,4%-%date:~4,2%-%date:~7,2%.xlsx"
pause
popd
```

The first line (`@echo off`) just keeps each line from displaying when executing. The "`pushd`" and "popd" commands are a workaround required if you have the command file on a network drive and you do not have a drive letter mapped for it. This is because CMD does not support UNC paths as current directories. `pushd` creates a temporary mapped drive to the current working directory and popd removes that temporary mapping (and so must be after the "pause" statement). The %CD% variable allows substituting the current working directory (caused by the pushd command).

Path Names

All (non-absolute) filenames are relative to the directory **where the configuration file is located**. This includes all relative paths (or files without a path) found in the yaml file. Under Windows, there are two absolute path formats supported, drive letter (D:\path) or UNC (\\server\share). Under Linux, absolute paths start with a slash ("/").

A "BasePath" option can be added to the config file to **override** the current working directory.

```
# loadsheets config
Description: OReilly Loadsheet
BasePath:   "d:/loadsheets/OReilly"
OutputFile: "../Output/Walker_{date}.xlsx"
LogFile:    "../Logs/Walker_{date}.txt"
```

The "`OutputFile`" and "`LogFile`" options are also available for overriding default filenames and you can include a special **`{date}`** macro which is replaced by today's date. In the example above, OutputFile would be, for example, `d:\loadsheets\Output\Walker_2020-12-04.xlsx` (`./` represents the current directory, and `../` the parent directory).

Important Note: If you are including path separators in the config file or command line, use **forward** slashes as shown above in these examples.

Writing Expressions

The program includes a very powerful expression processor for validation and filtering. Here are some of the features and syntax used by that processor.

Supported Literals

- **strings** - single and double quotes (e.g. "hello", 'hello')
- **numbers** - e.g. 103, 2.5, .5
- **dates** - "2020-10-09" (yyyy-mm-dd)

Comparison Operators

- **==** (equal)
- **!=** (not equal)
- **<** (less than)
- **>** (greater than)
- **<=** (less than or equal to)
- **>=** (greater than or equal to)

Logical Operators

- **not** or **!**
- **and** or **&&**
- **or** or **||**

String Operators

- **+** (concatenation)
- **matches** (regex match)
- **contains** (string contains)
- **startsWith** (has prefix)
- **endsWith** (has suffix)

Membership Operators

- **in** (contain)
- **not in** (does not contain)

Example:

$GROUP **in** ["human_resources", "marketing"]

Numeric Operators

- **..** (range) The range is inclusive: 1..3 == [1, 2, 3]

Example:

$AGE in 18**..**45

Ternary Operators

- $FOO **?** 'yes' **:** 'no'

Example:

$AGE > 30 **?** "mature" **:** "immature"

Builtin functions

- Today() - returns today's date for comparisons (e.g. "$EFF_DATE < Today()")
- Date(d) - encodes a date string (yyyy-mm-dd) for comparisons (e.g. "$EFF_DATE < Date(p1)")
- DateEqual(d1, d2) – true if the two dates are the same
- DateBefore(d1, d2) – true if “d1” is before “d2”
- DateOnOrBefore(d1, d2) – true if “d1” is equal to, or before “d2”
- Len() (length of array, map or string)
- all() (will return `true` if all elements satisfy the predicate)
- none() (will return `true` if all elements do NOT satisfy the predicate)
- any() (will return `true` if any element satisfies the predicate)
- one() (will return `true` if exactly ONE element satisfies the predicate)
- count() (returns number of elements what satisfies the predicate)

Notice