[DRAFT 2019.10.24] Paper 4725-2020:

SAS® packages - the way to share (a how to)

Bartosz Jabłoński
Warsaw University of Technology / Citibank Europe PLC Poland

ABSTRACT

When working on Base SAS® code, especially when it becomes complex, there is a point in time when a developer decides to break it into small pieces. The developer creates separate files for macros, formats/informats, and for functions or data too. Eventually the code is ready and tested and it is time for the deployment. The issue is that the code had been written on a local Windows machine and the deployment is on a remote Linux server. Folders[1] and files have to be created with the proper structure, code has to be run in the right order and not mixed up. Moreover it is not the developer who is deploying... Small challenge, isn't it?

How nice it would be to have it all (i.e. the code and its structure) wrapped up in a single file - a portable SAS package - which could be copied and deployed with a one-liner like: `%loadPackage(MyPackage)`?

In this article an idea of how to create such a "SAS-package" in a fast and convenient way will be proposed/shared. We will discuss:

- a concept of how to build a package,
- the tools required to do so, and
- a "how to" of the process (i.e. generating packages, loading, and using them).

The intended readers for the following document are intermediate SAS users (i.e. with good knowledge of Base SAS and practice in macro programming, see [**1**]) who want to learn how to share their code with others.

INTRODUCTION and CONTEXT

In the world of programmers, software developers, and "computer people" the concept of a package is well known and common one. To give an evidence of this statement let us consider three very popular examples the Linux, the Python, and the R software, and as an endorsement the following quotes.

According to [**5**]:

> In Linux distributions, a "package" refers to a compressed file archive containing all of the files that come with a particular application. [...] Most packages also contain installation instructions for the OS, as well as a list of any other packages that are dependencies (prerequisites required for installation).
>
> Common types of Linux packages include `.deb`, `.rpm`, and `.tgz`. Since Linux packages do not usually contain the dependencies necessary to install them, many Linux distributions use package managers that automatically read dependencies files and download the packages needed before proceeding with the installation.

According to [**4**]:

> Modular programming refers to the process of breaking a large, unwieldy programming task into separate, smaller, more manageable subtasks or modules. Individual

[1] Folders - also known as Directories

> modules can then be cobbled together like building blocks to create a larger application. Packages allow for a hierarchical structuring of the module [...].

According to [**3**]:

> In R, the fundamental unit of shareable code is the package. A package bundles together code, data, documentation, and tests, and is easy to share with others. As of January 2015, there were over 6,000 packages available on the Comprehensive R Archive Network, or CRAN, the public clearing house for R packages. This huge *variety of packages is one of the reasons that R is so successful*: the chances are that someone has already solved a problem that you're working on, and you can benefit from their work by downloading their package. [As of October 2019, there were over 15,000 packages available on the Comprehensive R Archive Network!]

The main goal of this article is to propose, describe/explain, and discuss an idea of a process and tools required to build SAS packages.

In subsequent sections we introduce the concept of a SAS package from both user and developer's point of view. What is worth to mention, and what is one of the biggest advantage of using SAS packages, is that work to be done on the user's side to use provided package is almost non.

The last section provides an example in which a package is created.

WHAT IS a SAS PACKAGE?

A **SAS package** [2] is an automatically generated, single, stand alone `zip` file containing organised and ordered code structures, created by the developer and extended with additional automatically generated "driving" files (i.e. descriptor, metadata, load, unload, and help files).

The purpose of a package is to be a simple, and easy in access, code sharing medium, which will allow: on the one hand, to separate the code complex dependencies created by the developer from the user experience with the final product and, on the other hand, reduce developer's and user's unnecessary frustration related to a remote deployment process.

To create a package the developer must prepare the code files and a description file, fit them into a structured form (see next sections for details), download and execute the `%generatePackage()` macro.

To use a package the user should download package's `zip` file into the packages' folder (containing downloaded `loadpackage.sas` file). And, in the SAS session, the user should run the following code:

```
filename packages "<directory/containing/packages>";
%include packages(loadpackage.sas);
%loadPackage(packageName)
```

[2] The idea presented in this article should not be confused with other occurrences of "package" concept which could be found in the SAS ecosystem, e.g. Proc DS2 packages, SAS/IML packages, or SAS Integration Technologies Publishing Framework packages.

THE USER: HOW TO and THE RULES

User's files and folders. Since the idea of a SAS package is to take off (from the user's shoulders) the burden of "necessity to know how it is all connected and dependent" there are only a few simple steps to be done on the user's end. The user's part of work required to use a package starts with setting up some files and folders, but is very short and in practice only the last step is repeated more than once. The work goes as follows:

- Create a folder for your packages, e.g. under Windows OS family `C:/SAS_PACKAGES` or under Linux/UNIX OS family `/home/<username>/SAS_PACKAGES`.
- Copy the `loadpackage.sas` file into packages' folder.
- Copy `zip` file with the package into packages' folder.

User's session. When all files and folders are settled the user, to enjoy a package, runs a SAS session and executes the following steps.

- To load the package user runs:
```
filename packages "<directory/containing/packages/>";
%include packages(loadpackage.sas);
%loadPackage(packageName)
```
- To get help information about the package printed in the log:
  - for general information about the package user runs:
    `%helpPackage(packageName)`
  - for all available information about the package user runs:
    `%helpPackage(packageName,*)`
  - for a particular element of the package, e.g. function or macro, user runs:
    `%helpPackage(packageName, entry)`

    where *entry* is a single word which is used for context search. "License" prints out license text.
- For removing (a.k.a. unloading) package's content:
  `%unloadPackage(packageName)`

After loading a package for the first time it is a *good practice* to read the log to find out more about packages content and the list of loaded elements.

Note. The code executed behind the scene is design to affect the user session environment in "as small as possible" way. The required minimum is compilation of the `%loadPackage()`, `%unloadPackage()`, and `%helpPackage()` macros, and a global macrovariable `SYSloadedPackages`. When the `%loadPackage()` macro is executed for the first time the `SYSloadedPackages` macrovariable is created and its value is updated with package name for each new loaded package.

Caution! There is one important *restriction* regarding the SAS session! Words "`package`" and "`packages`" are restricted as a file reference for the `FILENAME` statement and the `FILENAME()` function. These words are file references used internally by the `%loadPackage()`, `%helpPackage()`, and `%unloadPackage()` macros. Using them may cause unexpected results and may jeopardise package stability!

User's "under the hood". The above steps are all that is necessary to use and work with a package. There are also some additional things happening in the background. This section explains them in more details.

- The `%loadPackage()` macro loads all components of the package as a primary job. Additionally the package's header is printed into the log. Whenever an element of the package is loaded appropriate note is printed into the log. If there were any requirements provided they will be tested at this point.
- The `%helpPackage()` macro prints out into the log help information attached to the package's content. When no second argument is provided only package's description is printed out. When the second argument is provided, if it is an asterisk ("`*`") all help's content is printed out (for datasets also `proc contents` is run), if it is a `helpKeyword` then content search is executed based on its value

and only selected parts of help's content are printed out. If `helpKeyword` value is "License" then package's license is printed out.

- The `%unloadPackage()` macro cleans up the session. All objects created by package (except `execs`) are deleted. If `clean` files were provided their content is executed too.

THE DEVELOPER: HOW TO and THE RULES

Developer's files and folders. The developer's part of work to build a package starts with preparing a set of files and folders. This part goes as follows:

- Create a folder for your package a.k.a. package's folder (hint: name it the same name as your package's name).
- Create a descriptor file, named `description.sas`, and copy it into the package's folder. The file is mandatory, has simple structure, and it contains package's metadata and (short) description (compare [**3**]). The simple structure of the `description.sas` file can be seen in the Figure 1.

Figure 1. Package descriptor structure.

```
/* This is the description file for the package.          */
/* The collon (:) is a field separator and is restricted */
/* in lines of the header part.                           */

/* **HEADER** */ ❶
Type: Package                                                : ❷
Package: Xxxxxxx                                             : ❸
Title: Xxx xxxx xx xxxxxxxx xxx                              : ❹
Version: x.x                                                 : ❺
Author: Fname1 Lname1 (xxx1@yyy.zz), Fname2 Lname2 (xxx2@yyy.zz) : ❻
Maintainer: Fname3 Lname3 (xxx3@yyy.zz)                      : ❼
License: XXX                                                 : ❽
Encoding: XXXX                                               : ❾

Required: "Base SAS Software", "SAS/Xxx", "SAS/ACCESS Interface to Yyyy" : ①
ReqPackages: "somePackage (3.14)", "otherPackage (42)"                   : ②

/* **DESCRIPTION** */ ❿
/* All the text below will be used in help */
DESCRIPTION START:

Xxxxxxxxxxx xxxxxxx xxxxxx xxxxxxxx xxxxxxxx. Xxxxxxx
xxxx xxxxxxxxxxxx xx xxxxxxxxxxx xxxxxx. Xxxxxxx xxx
xxxx xxxxxx. Xxxxxxxxxxxxx xxxxxxxxxx xxxxxxx.

DESCRIPTION END:
```

The meaning of entries (a.k.a. tags) inside the `description.sas` file content are the following (the dark bullet marks an element which is *mandatory*):

(❶) marks the header's start. Mind the structure of the tag, i.e. slash, asterisk, *space*, and double asterisk (`/* **`). Each of the following lines is a `key:value` pair and such pair must be a *single line of text*. The colon (`:`) is a field separator and is restricted in lines of the header.

(❷) is a constant (i.e. "`Package`"), required, and not null value for the key `type`.

(❸) is a packages name. It is required, not null, up to 24 characters long, and shares naming restrictions like those for a SAS dataset name.

(❹) is short title of a package (i.e. one phrase). It is required and not null.

(❺) is a package version, it is required and not null. (preferred form is: an integer value for a stable version, a decimal value for non-stale one)

(❻) and (❼) are comma separated lists of package author(s) and maintainer(s). Elements of lists are of the form: "`Firstname Lastname (email@address.com)`".

(❽) is a license under which the package is distributed. It is required, not null, possible values could be: MIT, GPL2, BSD, PROPRIETARY, etc. The license text itself should be inserted into a `license.sas` file (see further steps).

(❾) is the information about SAS sessions encoding the package files were created. It is required and not null. Possible values could be: UTF8, WLATIN1, LATIN2, etc. and the values should satisfy requirements for the `encoding=` option of the `filename` statement.

(❿) is the last required part. It is the package description. It is a free text bounded between the "`DESCRIPTION START:`" and the "`DESCRIPTION END:`" tags. It could be multi-line. It should elaborate about the package and its components (e.g. macros, functions, datasets, etc.)

(①) is a quoted and comma separated list of licensed SAS products required for the SAS session under which the package will be used. Possible values inserted into the list should be the same as these the `proc setinit` prints in the log, e.g. `"Base SAS Software", "SAS/IML", "SAS/ACCESS Interface to Teradata"`. The "`Required`" tag is optional, when it is empty or not provided in the description the testing code is not generated. Though, it is recommended to add this one.

(②) is a quoted and comma separated list of names and versions (in brackets) of other SAS packages required for the package to work. Possible values inserted into the list should be formated like e.g. `"SQLinDS (0.1)"`. The "`ReqPackages`" tag is optional, when it is empty or not provided in the description the testing code is not generated.

Based on the header's information, the following internal macrovariables are generated: `packageName`, `packageVersion`, `packageTitle`, `packageAuthor`, `packageMaintainer`, `packageEncoding`, `packageLicense`.

- Inside the package folder create subfolders for the code files. A subfolder name have to be structured as follows:
  a) it contains only lower case letters, digits, and underscore ("_")
  b) it is composed of two parts separated by and underscore ("_"), i.e.
     - the first part is a series of digits (with leading zeros, e.g. `001`, `002`, ..., `123`, `124`, ...); its purpose is to keep execution sequence (e.g. in case the code must be ordered to run properly);
     - the second part, called folder's *type*, indicates subfolder's content. The *type* have to be one of the following:
       `libname` (for libraries assignments),
       `macro` (for macros),
       `function` (for `proc fcmp` functions),
       `format` (for formats and informats),
       `data` (for the code generating datasets),
       `exec` (for so called "free code") or
       `clean` (for the code cleaning up the session after `exec`s).

  An example of a package subfolders structure can be found in the Figure 2. In case the order of code execution is irrelevant the first part (i.e. digits and underscore) may be skipped.
  In case when the order of code execution is important, e.g. format `$efg.` must be defined before function `abc()`, two folders of *type* `format` and `function` with two different sequences of digits have to be created in a way that digits indicate execution order, e.g. `017_format` for the code of the format `$efg.` and `042_function` for the code of the function `abc()`.
  The list of types may be extended in the future if need be.
- Copy the files with the code into package subfolders in accordance with *types* and the following set of rules:
  - One-file-one-object, e.g. macro `%abc()` definition have to be contained in a single file without any definitions of other objects. The only exception are formats/informats, in this case one file may contains all four formats/informats sharing the same name, e.g. numeric format `abc.`, character

format `$abc.`, numeric informat `abc.`, and character informat `$abc.` all have to be kept in one file.

- An object name is a file name, e.g. a definition of a macro named `%abc()` have to be contained in a file named `abc.sas`.
- The definition of a function have to be enclosed in the following template of the `FCMP` procedure:

```
proc fcmp
  inlib  = work.&packageName.fcmp /* optional */
  outlib = work.&packageName.fcmp.package
  <... other options ...>
;
    <... function's or subroutine's body ...>
run;
quit;
```

The `inlib=` and `outlib=` options are, literally, set to: "`work.&packageName.fcmp`" and "`work.&packageName.fcmp.package`".

- The definition of a format/informat have to be enclosed in the following template of the `FORMAT` procedure:

```
proc format
  lib = work.&packageName.format
  <... other options ...>
;
  <... numeric format's definition ...>
  <... character format's definition ...>
  <... numeric informat's definition ...>
  <... character informat's definition ...>
run;
```

The `lib=` option is, literally, set to: "`work.&packageName.format`".

- `exec` folders are for so-called "free code", i.e. if a package, to be ready and usable, requires some additional code to be run (code not fitting provided *types*) - this code have to be inserted into a file inside one of the `exec` subfolders.
- `clean` folders are for cleaning after `exec`s, i.e. if a code from one of `exec` folders creates some object (e.g. a catalog, a macro, or a dataset) the appropriate code inside a `clean` subfolder have to be developed to remove that created object.
- Parts of code files which are to be used to generate help information must be enclosed between following text tags: "`/*** HELP START ***/`" and "`/*** HELP END ***/`", e.g. from the following file containing: an example of a macro code, a help text, and other comments:

```
/*** HELP START ***/
/* >>> %ABC() macro: <<<
 *
 * Main macro which allows to do
 * this and that...
 * Recomnended for SAS 9.4 and higher.
**/
/*** HELP END ***/

/* macro definition */
/*** HELP START ***/
%MACRO ABC(
  param1 /* parameter 1 is used for ... */
 ,param2 /* parameter 2 is used for ... */
);
/*** HELP END ***/
    <... body of a macro ...>
    <...                 ...>
    <... body of a macro ...>
%MEND ABC;

/*** HELP START ***/
/* EXAMPLE 1: use in datastep

   data class;
     set sashelp.class;
     %ABC(age, weight)
   run;

**/
/*** HELP END ***/
```

only the following parts of text will be extracted for help's purpose:

```
/* >>> %ABC() macro: <<<
 *
 * Main macro which allows to do
 * this and that...
 * Recomnended for SAS 9.4 and higher.
 *
**/
%MACRO ABC(
  param1 /* parameter 1 is used for ... */
 ,param2 /* parameter 2 is used for ... */
);
/* EXAMPLE 1: use in datastep

   data class;
     set sashelp.class;
     %ABC(age, weight)
   run;

**/
```

- Create a `license.sas` file containing license information for the package. Place the file in the package folder (together with the `description.sas` and subfolders). If no file is provided the `license.sas` will be generated with standard MIT license (read "Generating package in practice - a use-case" section and Appendix A to see the MIT license text).

- Create a folder for packages, e.g. under Windows OS family `C:/SAS_PACKAGES` or under Linux/UNIX OS family `/home/<username>/SAS_PACKAGES` and copy the `generatepackage.sas` file into this folder.

Developer's session. When all files and folders are settled the developer runs SAS session and executes the following code:

```
filename packages "<directory/containing/packages/>";
%include packages(generatepackage.sas);
/*ods html;*/
%generatePackage(filesLocation=<directory/with/package/files/>)
```

When the `%generatePackage` macro ends its execution the `packagename.zip` file, containing all package's content inside it, is created inside the "`<directory/with/package/files/>`".

Developer's "under the hood". Before reading this subsection further we recommend (for a better view) to have subsections "User's files and folders" and "User's session" of the "THE USER: HOW TO and THE RULES" section read.
When the `packagename.zip` file is created, by the `%generatePackge()` macro, a lot of things is happening behind the scenes. This section explains them in more details.
The first information the developer receives after process ends is a summary report displaying basic information about the package's content. In this summary the following elements are displayed: the package's location (i.e. folder), developer's `&sysuserid.`, creation's timestamp, SAS version, the package's encoding information (based on the `description.sas` file), and current SAS session's encoding. From the `description.sas` file the package's name, version, and license type are extracted and printed. The last part of the summary is a table displaying a list of files used to build up the package.
But the summary is only the tip of an iceberg. The following steps are executed when macro runs. At the beginning the `descriprion.sas` file is tested for existence and when result is positive the file is read. The following macrovariables (descriptors): `packageName`, `packageVersion`, `packageTitle`, `packageAuthor`, `packageMaintainer`, `packageEncoding`, `packageLicense`, `packageRequired`(optional), `packageReqPackages`(optional) are created and obligatory ones (i.e. first seven) are tested for values. If at least one of the descriptors is missing the process is aborted with error. If the package name is more than *24 characters* or contains illegal symbols (non alphanumeric or underscore) the process is aborted with error too. Value of the package version should be a positive number hence this is also tested.
If `zip` file with package name exists inside the package folder the `zip` is deleted and the new fileref is generated.
In the next step the package folder is scanned and structure of files and subfolders is extracted. Since files and subfolders with code have to be named only with low case letters - it is tested, if test fails the process is aborted with error.
At this point the summary mentioned above is generated.
Further steps create so called "driving" files. The `description.sas` is copied into the `zip`. The `license.sas` is either copied or MIT license is generated. The `packagemetadata.sas` file with descriptors macrovariables is created.
The next one is the `load.sas` file. If `packageRequired` or `packageReqPackages` macrovariables are present two parts of code for testing requirements are generated, respectively. Both codes are design to set up the `packageRequiredErrors` macrovariable to one if requirements are not met. If the `packageRequiredErrors` is positive the loading of the package is aborted with the following error message "`ERROR: Loading package &packageName. will be aborted! Required SAS components are missing.`" After requirements testing all "includes" are generated. In case of functions or formats/informats, if the first one is detected, a code to update `cmplib` or `fmtsearch` is generated. If the subfolders of the `exec` type are detected a snippet code to print their content is added. As the final part of the `load.sas` code of data step for creating/updating the `SYSloadedPackages` global macrovariable is added.

The process continues with generation of the `unload.sas` code. As the first part the code to print and execute the `clean` type is assembled. As the second step the code for the macros and formats/informats deletion is generated. Deletion of functions follows after. And the last is the code for libraries unassignment and the `SYSloadedPackages` actualisation.

The third file, `help.sas`, following snippets are generated. The first to print out content of the `description.sas` file, namely the description part. The second to print out content of the `license.sas` file. The third snippet creates datastep used for content search and print out of the help text from the package files (macros, functions, etc.)

The final part of the `%generatePackage` macro is a datastep copying package's code files into the `zip`. Eventually within the package `zip` file we will find:

- Copies of all files from package subfolders but with modified names, what is needed to keep the ordering in place. Each code file's name is extended with a prefix of a form: undescore, subfolder name, and dot. For example if file name is `abc.sas` and subfolder's name is `007_macro` then the new name is `_007_macro.abc.sas`.
- The `description.sas` file (the one described earlier) and the `license.sas` file.
- The `packagemetadata.sas` file containing definitions of internal macrovariables used by the `%loadPackage()`, `%helpPackage()`, and `%unloadPackage()` macro.
- The `load.sas` file containing code executed by the `%loadPackage()` macro. The file content is built based on the subfolders and files structure provided by the developer. The file is a series of `%include`s, with additional automatic note comments in `%put` statements, and, if need be, set of options modifications e.g. inserts to `fmtsearch` option for formats/informats or appends to `cmplib` option for functions. If files of type `exec` are inside the package a code printing out their content into the log is also attached.
- The `help.sas` file containing code executed by the `%helpPackage()` macro. The file contains 1) code which displays general package description, 2) code which search for a content based on `helpKeyword`'s value and prints out the information, and 3) code which, if `helpKeyword`'s value is "License", prints out the license text.
- The `unload.sas` file containing code executed by the `%unloadPackage()` macro. The file's content is built based on the subfolders and files structure provided by the developer. Code inside the file removes macros, functions, formats, datasets and libraries created during loading process. It restores `fmtsearch` and `cmplib` options. If `clean` type subfolder is provided files from within the folder are `%include`d (they are executed as the first).

Figure 2. Example of a package's subfolders structure.

```
<packageName>
 ..
  |
  +-000_libname [one file one libname]
  |           |
  |           +-abc.sas [a file with a code creating libname ABC]
  |
  +-001_macro [one file one macro]
  |          |
  |          +-hij.sas [a file with a code creating macro HIJ]
  |          |
  |          +-klm.sas [a file with a code creating macro KLM]
  |
  +-002_function [one file one function,
  |             |  option OUTLIB= should be: work.&packageName.fcmp.package
  |             |  option INLIB=  should be: work.&packageName.fcmp
  |             |  (both literally with macrovariable name and "fcmp" sufix)]
  |             |
  |             +-efg.sas [a file with a code creating function EFG]
  |
  +-003_format [one file one format,
  |           |  option LIB= should be: work.&packageName.format
  |           |  (literally with macrovariable name and "format" sufix)]
  |           |
  |           +-efg.sas [a file with a code creating format EFG and informat EFG]
  |
  +-004_data [one file one dataset]
  |         |
  |         +-abc.efg.sas [a file with a code creating dataset EFG in library ABC]
  |
  +-005_exec [so called "free code", content of the files will be printed
  |         |  to the log before execution]
  |         |
  |         +-<no file, in this case folder may be skipped>
  |
  +-006_format [if your codes depend eachother you can order them in folders,
  |           |  e.g. code from 003_... will be executed before 006_...]
  |           |
  |           +-abc.sas [a file with a code creating format ABC,
  |                      using the definition of the format EFG]
  +-007_function
  |             |
  |             +-<no file, in this case folder may be skipped>
  |
  +-<sequential number>_<type [in lower case]>
  |
  +-...
  |
  +-00n_clean [if you need to clean something up after exec file execution,
  |          |  content of the files will be printed to the log before execution]
  |          |
  |          +-<no file, in this case folder may be skipped>
  +-...
  |
  ...
```

Generating package in practice - a use-case

The practical **example** will be build based on one of author's favorite SAS article, namely Mike Rhoads' "*Use the Full Power of SAS in Your Function-Style Macros*" [**2**], which introduces the macro-function-sandwich programming approach. The idea is to allow user to execute SQL's "select" code within a datastep, e.g.

```
data class_subset;
  set %SQL(select
             name
           , sex
           , height
           from
             sashelp.class
           where
             age > 12
          )
          (rename=(height=heightInch));

  heightCm = 2.54 * heightInch;
run;
```

Thus the package's name will be *SQLinDS* and it will be providing the `%SQL()` macro which allow users to write queries like the one above. Internally the `%SQL()` macro uses a user defined function, another macro, and stores intermediate data (views) inside a predefined library (pointing to a subdirectory of the `work`). Package will be build with 5 files: `description.sas`, two `macros`, one `function`, and one `library`. Let's assume we created the following package's folder `C:/SAS_PACKAGES/SQLinDS/` and we copied `generatepackage.sas` file into the `C:/SAS_PACKAGES/` directory. The structure of subfolders created for the package is presented in the Figure 3.

Figure 3. SQLinDS package - subfolders' structure.

```
<C:/SAS_PACKAGES/SQLinDS/>
 ..
  |
  +-description.sas ❶
  |
  +-000_libname
  |           |
  |           +-dssql.sas ❷
  |
  +-001_macro
  |         |
  |         +-dssql_inner.sas ❸
  |         |
  |         +-sql.sas ❺
  |
  +-002_function
  |            |
  |            +-dssql.sas ❹
  |
  +-license.sas ❻
```

Details of the files composing package (including license) can be found in the Appendix A or on the web (see "THE CODE" section).

When all files are placed inside proper subfolders we start a new SAS session and we execute the following code:

```
filename packages "C:/SAS_PACKAGES/";
%include packages(generatepackage.sas);
/*ods html;*/
%generatePackge(filesLocation=C:/SAS_PACKAGES/SQLinDS/)
```

As a result, extra the summary report, we receive (inside the `C:/SAS_PACKAGES/SQLinDS/` folder) the `sqlinds.zip` file. Our package is prepared. And ready for sharing!

THE CODE

If you are interested in testing approaches presented above yourself and want to play a bit with the code and data you can download SAS codes which were the motivation for this paper under the following "world wild web" address:

`http://www.mini.pw.edu.pl/~bjablons/SASpublic/SAS_PACKAGES`

or from authors GitHub:

`https://github.com/yabwon/SAS_PACKAGES`

REFERENCES

[1] Art Carpenter, "Carpenter's Guide to Innovative SAS Techniques", SAS Press

[2] Mike Rhoads, "Use the Full Power of SAS in Your Function-Style Macros", SAS Global Forum 2012 Proceedings, `https://support.sas.com/resources/papers/proceedings12/004-2012.pdf`

[3] Hadley Wickham, "R Packages: Organize, Test, Document, and Share Your Code", O'Reilly Media 2015, `http://r-pkgs.had.co.nz/description.html`

[4] `https://realpython.com/python-modules-packages/` as of October 2019

[5] `https://www.internetblog.org.uk/post/1520/what-is-a-linux-package/` as of October 2019

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at one of the following e-mail address:

`yabwon`✉`gmail.com`

`bartosz1.jablonski`✉`citi.com`

or via the following LinkedIn profile:

`www.linkedin.com/in/yabwon`

—

Appendix A

The content of files composing the `SQLinDS` package.

❶

```
/* This is the description file for the package.          */
/* The collon (:) is a field separator and is restricted */
/* in lines of the header part.                           */

/* **HEADER** */
Type: Package
Package: SQLinDS
Title: SQL queries in Data Step
Version: 1.0
Author: Mike Rhoads (RhoadsM1@Westat.com)
Maintainer: Bartosz Jablonski (yabwon@gmail.com)
License: MIT
Encoding: UTF8

/* **DESCRIPTION** */
/* All the text below will be used in help */
DESCRIPTION START:

The SQLinDS package is an implementation of
the macro-function-sandwich concept introduced in:
"Use the Full Power of SAS in Your Function-Style Macros"
the article by Mike Rhoads, Westat, Rockville, MD

Copy of the article can be found at:
https://support.sas.com/resources/papers/proceedings12/004-2012.pdf

SQLinDS package provides following components:
 1) %dsSQL_inner() macro
 2) dsSQL() function
 3) %SQL() macro

Library DSSQL is created in a subdirectory of the WORK library.

DESCRIPTION END:
```

❷

```
/*** HELP START ***/

/* >>> dsSQL library: <<<
 *
 * The dsSQL library stores temporary views
 * generated during %SQL() macro's execution.
 * If possible, created as a subdirectory of WORK:

   options dlCreateDir;
   LIBNAME dsSQL BASE "%sysfunc(pathname(WORK))/dsSQLtmp";

 * if not then redirected to WORK

   LIBNAME dsSQL BASE "%sysfunc(pathname(WORK))";

**/

/*** HELP END ***/

data WORK._%sysfunc(datetime(), hex16.)_;
  length option $ 64;
  option = getoption("dlCreateDir");
run;

options dlCreateDir;

data _null_;
  set _LAST_;
  rc1 = LIBNAME("dsSQL", "%sysfunc(pathname(work))/dsSQLtmp", "BASE");
  rc2 = LIBREF("dsSQL");
  if rc2 NE 0 then
    rc1 = LIBNAME("dsSQL", "%sysfunc(pathname(work))", "BASE");
  call execute ("options " || strip(option) || ";");
run;

proc delete data = _last_;
run;

libname dsSQL LIST;
```

❸

```
/*** HELP START ***/

/* >>> %dsSQL_Inner() macro: <<<
 *
 * Internal macro called by dsSQL() function.
 *
 * Recomnended for SAS 9.3 and higher.
 * Based on paper:
 * "Use the Full Power of SAS in Your Function-Style Macros"
 * by Mike Rhoads, Westat, Rockville, MD
 * https://support.sas.com/resources/papers/proceedings12/004-2012.pdf
 *
**/

/*** HELP END ***/

/* inner macro */
%MACRO dsSQL_Inner() / SECURE;
  %local query;
  %let query = %superq(query_arg);
  %let query = %sysfunc(dequote(&query));

  %let viewname = dsSQL.dsSQLtmpview&UNIQUE_INDEX_2.;
  proc sql;
    create view &viewname as
      &query
    ;
  quit;
%MEND dsSQL_Inner;
```

❹

```
/*** HELP START ***/

/* >>> dsSQL() function: <<<
 *
 * Internal function called by %SQL() macro.
 *
 * Recomnended for SAS 9.3 and higher.
 * Based on paper:
 * "Use the Full Power of SAS in Your Function-Style Macros"
 * by Mike Rhoads, Westat, Rockville, MD
 * https://support.sas.com/resources/papers/proceedings12/004-2012.pdf
 *
**/

/*** HELP END ***/

proc fcmp
  inlib  = work.&packageName.fcmp
  outlib = work.&packageName.fcmp.package
;
  function dsSQL(unique_index_2, query $) $ 41;
    length
      query query_arg $ 32000 /* max querry length */
      viewname $ 41
    ;
    query_arg = dequote(query);
    rc = run_macro('dsSQL_Inner' /* <-- inner macro */
                  ,unique_index_2
                  ,query_arg
                  ,viewname
                  );
    if rc = 0 then return(trim(viewname));
    else
      do;
         put 'ERROR:[function dsSQL] A problem with the dsSQL() function';
         return(" ");
      end;
  endsub;
run;
quit;
```

❺

```
/*** HELP START ***/

/* >>> %SQL() macro: <<<
 *
 * Main macro which allows to use
 * SQL's queries in the data step.
 * Recomnended for SAS 9.3 and higher.
 * Based on paper:
 * "Use the Full Power of SAS in Your Function-Style Macros"
 * by Mike Rhoads, Westat, Rockville, MD
 * https://support.sas.com/resources/papers/proceedings12/004-2012.pdf
 *
 * EXAMPLE 1: simple sql querry

   data class_subset;
     set %SQL(select name, sex, height from sashelp.class where age > 12);
   run;

 * EXAMPLE 2: with dataset options

   data renamed;
     set %SQL(select * from sashelp.class where sex = "F")(rename = (age=age2));
   run;

 * EXAMPLE 3: dictionaries in datastep

   data dictionary;
     set %SQL(select * from dictionary.macros);
   run;

**/

/*** HELP END ***/


/* outer macro */
%MACRO SQL() / PARMBUFF SECURE;
  %let SYSPBUFF = %superq(SYSPBUFF); /* macroquoting */
  %let SYSPBUFF = %substr(&SYSPBUFF, 2, %LENGTH(&SYSPBUFF) - 2); /* remove brackets */
  %let SYSPBUFF = %superq(SYSPBUFF); /* macroquoting */
  %let SYSPBUFF = %sysfunc(quote(&SYSPBUFF)); /* quotes */
  %put NOTE-***the querry***; /* print out the querry in the log */
  %put NOTE-&SYSPBUFF.;
  %put NOTE-****************;

  %local UNIQUE_INDEX; /* internal variable, a unique index for views */
    %let UNIQUE_INDEX = &SYSINDEX;
  %sysfunc(dsSQL(&UNIQUE_INDEX, &SYSPBUFF)) /* <-- call dsSQL() function,
                                                 see the WORK.SQLinDSfcmp dataset */
%MEND SQL;
```

❻ MIT license text (used by default, mind macrocode in the first line):

```
Copyright (c) %sysfunc(today(),year4.) &packageAuthor.

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```